



Introduction au langage Python 3 et au framework web Flask par l'exemple

Serge Tahé, septembre 2020

Ce site a été créé avec le convertisseur [Word ou ODT - > HTML] créé par l'IA Gemini 3 en janvier 2026.

https://tahe.developpez.com/ - Ce cours tutoriel écrit par Serge Tahé est mis à disposition du public selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Partage dans les Mêmes Conditions 3.0 non transposé.

1 Avant-propos

Le PDF de ce document est disponible |ICI|.

Ce document propose une liste de scripts Python dans différents domaines :

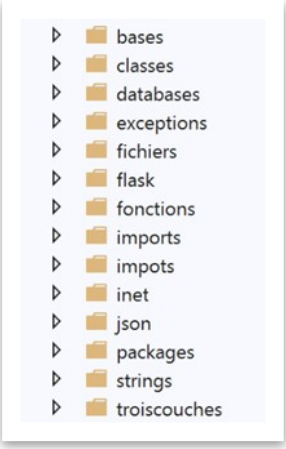
- les fondamentaux du langage ;
- la gestion de bases de données MySQL et PostgreSQL ;
- la programmation réseau TCP/ IP (protocoles HTTP, POP3, IMAP, SMTP) ;
- la programmation web MVC avec le Framework FLASK ;
- les architectures trois couches et la programmation par interfaces ;

Ce n'est pas un cours Python exhaustif mais un recueil d'exemples destinés à des développeurs ayant déjà utilisé un langage de script tel que Perl, PHP, VBScript ou des développeurs habitués aux langages typés tels que Java ou C# et qui seraient intéressés par découvrir un langage de script orienté objet. Ce document est peu approprié pour des lecteurs n'ayant jamais ou peu programmé.

Ce document n'est pas non plus un recueil de "bonnes pratiques". Le développeur expérimenté pourra ainsi trouver que certains codes pourraient être mieux écrits. Ce document a pour seul objectif de donner des exemples à une personne désireuse de s'initier au langage Python 3 et au Framework Flask. Elle approfondira ensuite son apprentissage avec d'autres documents.

Les scripts sont commentés et les résultats de leur exécution reproduits. Parfois des explications supplémentaires sont fournies. Le document nécessite une lecture active : pour comprendre un script, il faut à la fois lire son code, ses commentaires et ses résultats d'exécution.

Les exemples du document sont disponibles à l'adresse |https://tahe.developpez.com/tutoriels-cours/python-flask-2020/documents/python-flask-2020.rar| :



Le PDF du document est disponible |ICI|.

Le document est une mise à jour d'un ancien document paru en juin 2011 |https://tahe.developpez.com/tutoriels-cours/python/|. Le document de 2011 avait été construit avec l'interpréteur Python 2.7. Depuis, des versions Python 3.x sont apparues. En février 2020, on en est à la version 3.8. Les versions 3.x ont amené une discontinuité dans la portabilité entre versions : des codes s'exécutant sous Python 2.7 peuvent se révéler inutilisables sous Python 3.x. C'est notamment le cas pour l'écriture console. En 2.7 on peut écrire `[print "toto"]` alors qu'en 3.x il faut écrire `[print("toto")]` : il faut mettre des parenthèses. Cette simple évolution fait que les codes fournis avec le document de 2011 sont pour la plupart inutilisables directement avec Python 3.x. Il faut les modifier.

Ce nouveau document ne se contente pas de mettre à jour les codes de 2011 pour qu'ils soient exécutables avec Python 3.8 :

- les sections sur la programmation TCP-IP et l'usage des bases de données ont subi d'importantes évolutions ;
- la section sur la programmation web qui n'était qu'une introduction est désormais un cours complet utilisant le framework FLASK ;

Pour construire ce cours, j'ai suivi une démarche inhabituelle : j'ai fait un portage du cours PHP [Introduction au langage PHP7 par l'exemple]. Je n'ai donc pas suivi les structures traditionnelles des cours Python ou Flask. Je voulais avant tout savoir comment je

pouvais faire avec Python 3 ce que j'avais fait en PHP 7. Le résultat est que j'ai pu refaire, en Python 3 / Flask, tous les exemples du cours PHP 7.

Ce document peut comporter des erreurs ou des insuffisances. Le lecteur peut utiliser le flux de discussion |forum| pour les signaler.

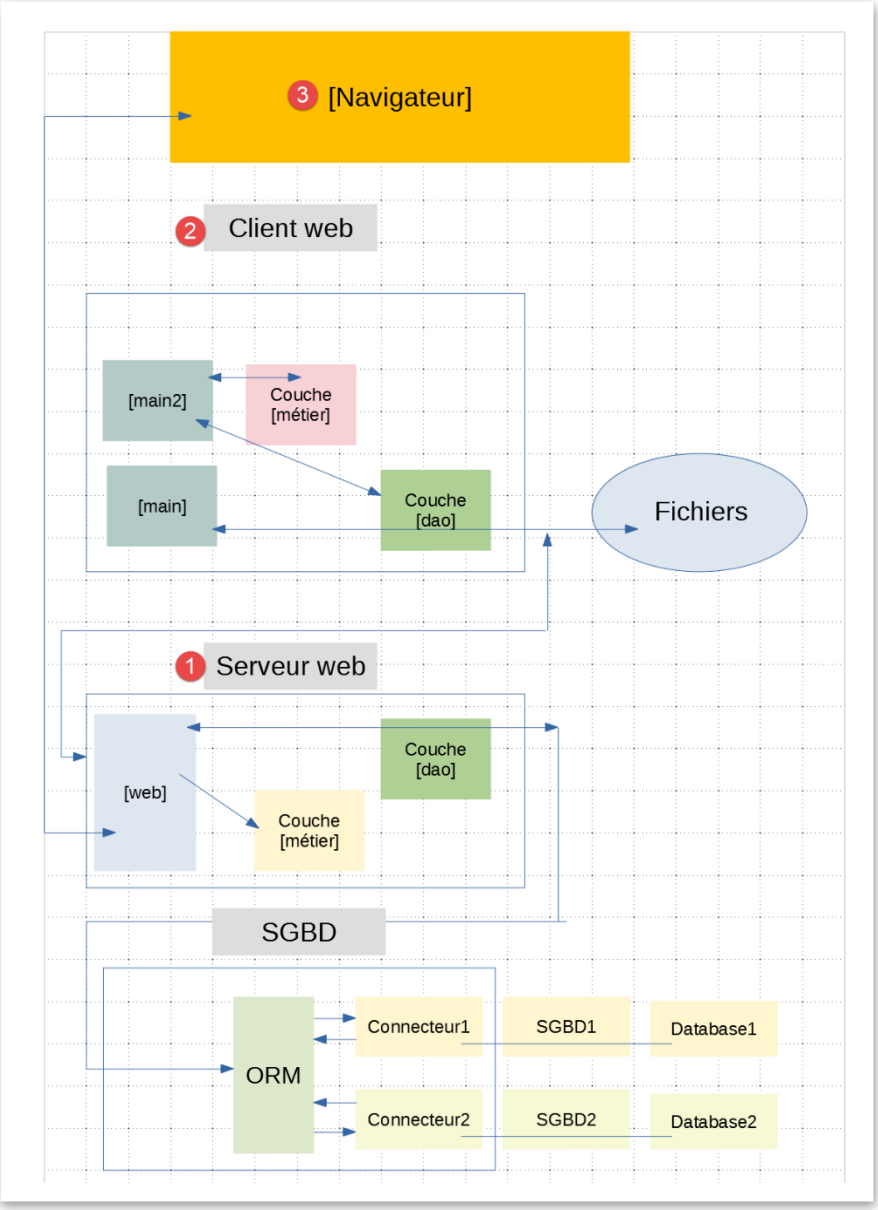
Le contenu du document est le suivant :

Chapitre	Dossier du code	Contenu
Chapitre 1		Présentation du cours
Chapitre 2		Installation d'un environnement de travail
Chapitre 3	`[bases]`	Les bases du langage Python – structures du langage – type de données – fonctions – affichage console – chaînes de formatage – changement de types – les listes – les dictionnaires – les expression srégulières
Chapitre 4	`[strings]`	Notation des chaînes de caractères – méthodes de la classe <str> - encodage / décodage des chaînes de caractères en UTF-8
Chapitre 5	`[exceptions]`	Gestion des exceptions
Chapitre 6	`[fonctions]`	Portée des variables – mode de passage des paramètres – utiliser des modules – le Python Path – paramètres nommés – fonctions récursives
Chapitre 7	`[fichiers]`	Lecture / écriture d'un fichier texte – gestion des fichiers encodés en UTF-8 – gestion des fichiers jSON
Chapitre 8	`[impots/v01]`	Version 1 de l'exercice d'application, un calcul d'impôt sur les revenus. L'application est déclinée en 18 versions – La version 1 implémente une solution procédurale
Chapitre 9	`[imports]`	Gestion des dépendances d'une application par importation de modules – une méthode de gestion des dépendances est présentée – elle est utilisée dans tout le document – gestion du Python Path
Chapitre 10	`[impots/v02]`	La version 2 de l'application reprend la version 1 en rassemblant toutes les constantes de la configuration dans un fichier de configuration qui gère également le Python Path
Chapitre 11	`[impots/v03]`	La version 3 de l'application reprend la version 2 en utilisant des fonctions encapsulées dans un module – la gestion des dépendances est faite par configuration – introduction de fichier jSON pour lire les données nécessaires au calcul de l'impôt et écrire les résultats des calculs
Chapitre 12	`[classes/01]`	Classes – héritage – méthodes et propriétés – getters / setters – constructeur – propriété `[__dict__]`
Chapitre 13	`[classes/02]`	Présentation des classes `[BaseEntity]` et `[MyException]` utilisées dans le reste du document – `[BaseEntity]` facilite les conversions objet / dictionnaire
Chapitre 14	`[troiscouches]`	Architecture en couches et programmation par interfaces. Ce chapitre présente les méthodes de programmation utilisées dans le reste du document
Chapitre 15	`[impots/v04]`	Version 4 de l'application – cette version implémente une solution avec une architecture en couches, la programmation par interfaces, l'utilisation de classes dérivées de `[BaseEntity]` et `[MyException]`
Chapitre 16	`[databases/mysql]`	Installation du SGBD MySQL – connexion à une base de données – création d'une table – exécution d'ordres SQL SELECT, UPDATE, DELETE, INSERT – transaction – requêtes SQL paramétrées
Chapitre 17	`[databases/postgresql]`	Installation du SGBD PostgreSQL – connexion à une base de données – création d'une table – exécution d'ordres SQL SELECT, UPDATE, DELETE, INSERT – transaction – requêtes SQL paramétrées

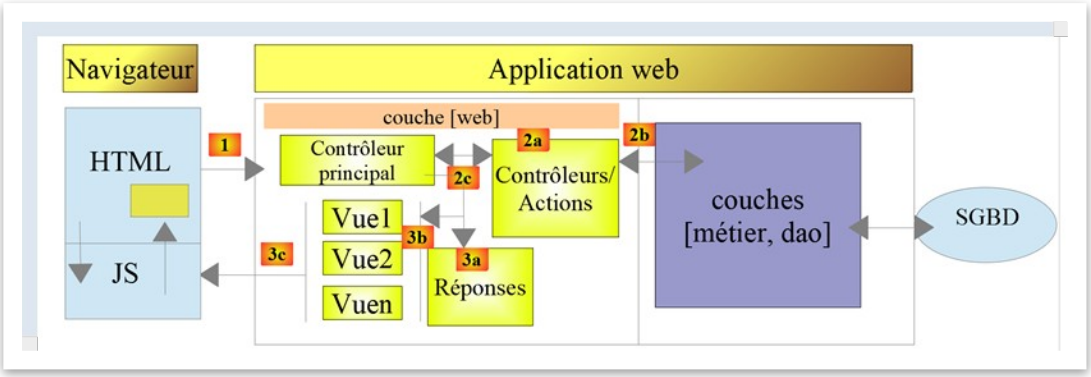
Chapitre 18	**[databases/anysgbd]**	Ecriture de code indépendant du SGBD
Chapitre 19	**[databases/sqlalchemy]**	L'ORM (Object Relational Mapper) SqlAlchemy – un ORM permet de travailler de façon unifiée avec différents SGBD - mapping classes / tables SQL – opérations sur les classes images des tables SQL
Chapitre 20	**[impots/v05]**	Version 5 de l'application de calcul de l'impôt – Utilisation de l'architecture en couches de la version 04 et de l'ORM SqlAlchemy pour travailler avec les SGBD MySQL et PostgreSQL
Chapitre 21	**[inet]**	Programmation internet – protocole TCP / IP (Transfer Control Protocol / Internet Protocol) - protocoles HTTP (HyperText Transfer Protocol) – SMTP (Simple Mail Transfer Protocol) – POP (Post Office Protocol) – IMAP (Internet Message Access Protocol)
Chapitre 22	**[flask]**	Services web avec le framework web Flask – affichage d'une page HTML – service web jSON – requêtes GET et POST – gstion d'une session web
Chapitre 23	**[impots/http-servers/01]** **[impots/http-clients/01]**	Version 6 de l'exercice d'application - Création d'un service web jSON de calcul de l'impôt avec une architecture multicouche - Ecriture d'un client web pour ce serveur avec une architecture multicouche – programmation client / serveur – utilisation du module **[requests]**
Chapitre 24	**[impots/http-servers/02]** **[impots/http-clients/02]**	Version 7 de l'exercice de l'application – la version 6 est améliorée : le client et le serveur sont multithtreadés – utilitaires **[Logger]** pour loguer les échanges client / serveur – **[SendMail]** pour envoyer un mail à l'administrateur de l'application
Chapitre 25	**[impots/http-servers/03]** **[impots/http-clients/03]**	Version 8 de l'exercice d'application – la version 7 est améliorée par l'usage d'une session web
Chapitre 26	**[xml]**	Gestion du XML (eXtended Markup Language) avec le module **[xmltodict]**
Chapitre 27	**[impots/http-servers/04]** **[impots/http-clients/04]**	Version 9 de l'exercice d'application – la version 8 est modifiée pour avoir des échanges client / serveur en XML ;
Chapitre 28	**[impots/http-servers/05]** **[impots/http-clients/05]**	Version 10 de l'exercice d'application – au lieu de traiter N contribuables par N requêtes GET, on utilise une unique requête POST avec les N contribuables dans le corps du POST
Chapitre 29	**[impots/http-servers/06]** **[impots/http-clients/06]**	Version 11 de l'exercice d'application – l'architecture client / serveur de l'application est modifiée : la couche **[métier]** passe du serveur au client
Chapitre 30	**[impots/http-servers/07]**	Version 12 de l'exercice d'application – cette version implémente un serveur MVC (Model – View – Controller) délivrant indifféremment, à la demande du client, du jSON, du XML et du HTML. Ce chapitre implémente les versions jSON et XML du serveur
Chapitre 31	**[impots/http-clients/07]**	Implémentation des clients jSON et XML du serveur MVC de la version 12
Chapitre 32	**[impots/http-servers/07]**	Implémentation du serveur HTML de la version 12 – utilisation du framework CSS Bootstrap –
Chapitre 33	**[impots/http-servers/08]**	Version 13 de l'exercice d'application - Refactorisation du code de la version 12 – gestion de la session avec le module **[flask_session]** et un serveur Redis – utilisation de mots de passe cryptés
Chapitre 34	**[impots/http-servers/09]**	Version 14 de l'exercice d'application – implémentation d'URL avec un

	[impots/http-clients/09]	jeton CSRF (Cross Site Request Forgery)
Chapitre 35	**[impots/http-servers/10]**	Version 15 de l'exercice d'application – refactorisation du code de la version 14 pour gérer deux types d'actions : ASV (Action Show View) qui ne servent qu'à afficher une vue sans modifier l'état du serveur, ADS (Action Do Something) qui font une action qui modifie l'état du serveur – ces actions se terminent toutes par une redirection vers une action ASV – cela permet de gérer correctement les rafraîchissements de page du navigateur client
Chapitre 36	**[impots/http-servers/11]**	Version 16 de l'application – gestion des URL avec préfixe
Chapitre 37	**[impots/http-servers/12]**	Version 17 de l'application – portage de la version 16 sur un serveur Apache / Windows
Chapitre 38	**[impots/http-servers/13]**	Version 18 de l'application – corrige une anomalie de la version 17

La version finale de l'exercice d'application est une application client / serveur de calcul de l'impôt avec l'architecture suivante :



La couche **[web]** ci-dessus est implémentée avec une architecture MVC :

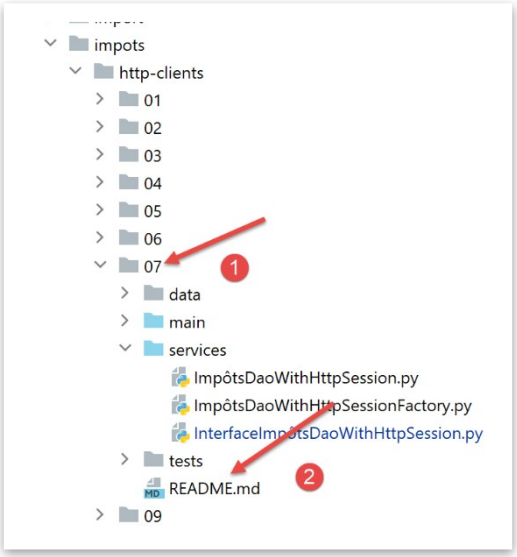


Le contenu du document est dense. Le lecteur qui ira jusqu'au bout aura une bonne vision de la programmation web MVC en Python / Flask et au-delà une bonne vision de la programmation web MVC dans d'autres langages.

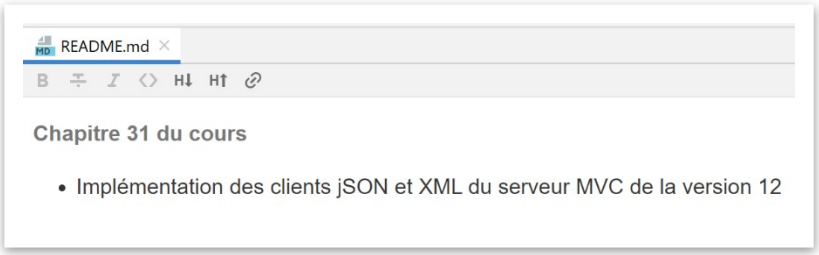
Le lecteur qui préfère voir du code, le tester et le modifier plutôt que de lire un cours pourra procéder ainsi :

- installer un environnement de travail :
  - un interpréteur Python ;
  - l'IDE PyCharm Community ;
  - Laragon (serveur Apache, SGBD MySQL, serveur Redis) ;
  - le SGBD PostgreSQL ;
  - le client HTTP Postman ;
  - le serveur de mails hMailServer ;

- explorer le code des exemples |https://tahe.developpez.com/tutoriels-cours/python-flask-2020/documents/python-flask-2020.rar| :
- dans chaque dossier, on trouve un fichier **[README.md]** qui lie le dossier à un chapitre du cours et résume son contenu :



Le fichier **[README.md]** ressemble à ceci :



Il est peu probable que le lecteur lise la totalité du cours :

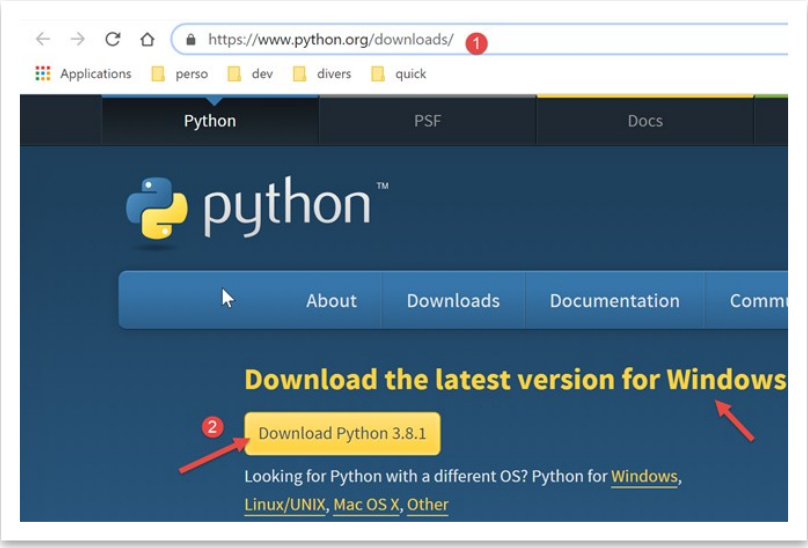
- le lecteur débutant pourra lire les chapitres 1 à 15 et s'arrêter là. Il pourra ensuite passer du temps à coder ses propres scripts Python avant de revenir à ce cours ;
- le lecteur ayant les bases de Python et voulant s'initier aux bases de données et à l'ORM (Object Relational Mapper) **[sqlalchemy]** pourra se contenter des chapitres 16 à 20 ;
- le lecteur voulant s'initier à la programmation internet (HTTP, SMTP, POP3, IMAP) pourra lire le chapitre 21. Ce chapitre est assez complexe et montre des scripts avancés. On peut le lire à deux niveaux :
  - pour découvrir les protocoles de l'internet ;
  - pour obtenir des scripts exploitant ces protocoles ;
- le lecteur ayant les bases de Python et voulant s'initier à la programmation web avec le framework Flask lira le chapitre 22 ;
- le lecteur désirant approfondir la programmation web avec le framework Flask pourra étudier les chapitres 23 à 38. On y construit des applications client / serveur de plus en plus complexes ainsi qu'une application HTML / Python suivant le modèle de développement MVC (Modèle – Vue – Contrôleur). Cette application est développée au chapitre 32. On pourra s'arrêter là. Les chapitres suivants amènent des modifications non fondamentales ;

Serge Tahé, septembre 2020

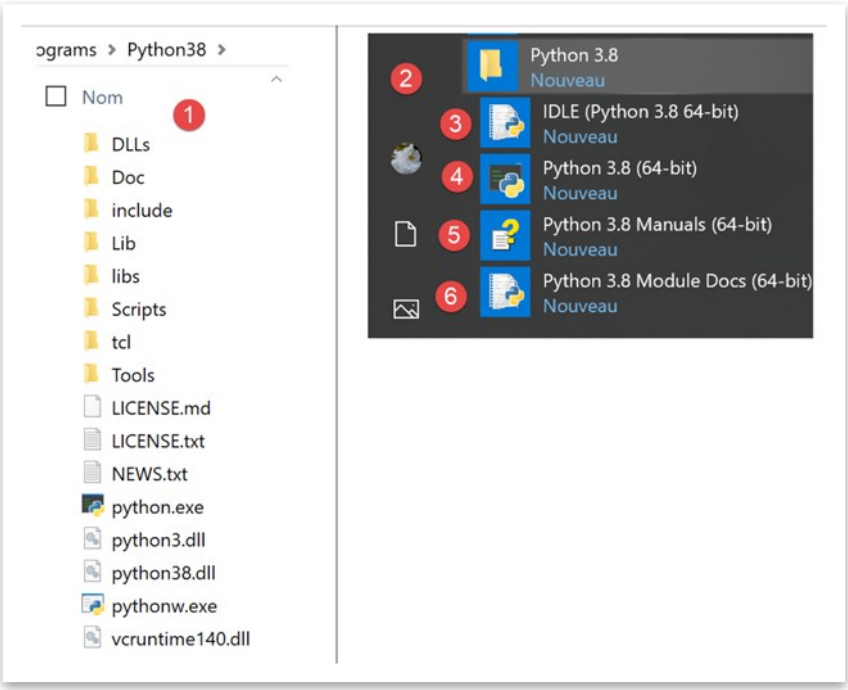
2 Installation d'un environnement de travail

2.1 Python 3.8.1

Les exemples de ce document ont été testés avec l'interpréteur Python 3.8.1 disponible à l'URL |https://www.python.org/downloads/| (fév 2020) sur une machine Windows 10 :

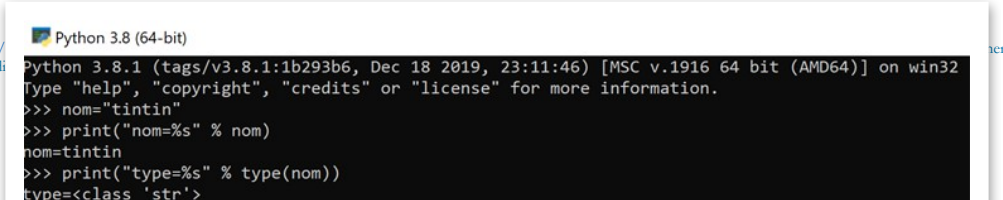


L'installation de Python donne naissance à l'arborescence de fichiers **[1]** et au menu **[2]** dans la liste des programmes :



- **[3-4]** : deux interpréteurs Python interactifs ;
- **[5]** : la documentation Python ;
- **[6]** : la documentation des modules Python ;

Nous n'utiliserons pas l'interpréteur Python interactif. Il faut simplement savoir que les scripts de ce document pourraient être exécutés avec cet interpréteur. Pratique pour tester le fonctionnement d'une fonctionnalité Python, il l'est peu pour des scripts devant être réutilisés. Voici un exemple avec l'option **[4]** ci-dessus :



Le prompt >>> permet d'émettre une instruction Python qui est immédiatement exécutée. Le code tapé ci-dessus a la signification suivante :

```
1.  >>> nom="tintin"
2.  >>> print("nom=%s" % nom)
3.  nom=tintin
4.  >>> print("type=%s" % type(nom))
5.  type=<class 'str'>
6.  >>>
```

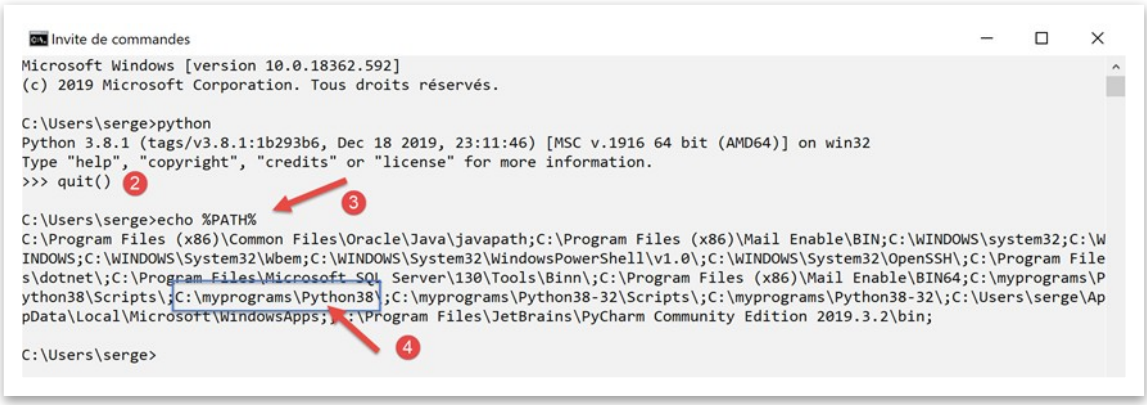
Lignes :

- 1 : initialisation d'une variable. En Python, on ne déclare pas le type des variables. Celles-ci ont automatiquement le type de la valeur qu'on leur affecte. Ce type peut changer au cours du temps ;
- 2 : affichage du nom. '*nom=%s*' est un format d'affichage où %s est un paramètre formel désignant une chaîne de caractères. *nom* est le paramètre effectif qui sera affiché à la place de %s ;
- 3 : le résultat de l'affichage ;
- 4 : l'affichage du type de la variable *nom ;*
- 5 : la variable nom est ici de type *class*. Avec Python 2.7 elle aurait la valeur *<type 'str'>* ;

Maintenant, ouvrons une console Windows :



Le fait qu'on ait pu taper **[python]** en **[1]** et que l'exécutable **[python.exe]** ait été trouvé montre que celui-ci est dans le PATH de la machine Windows. C'est important car cela signifie que les outils de développement Python sauront trouver l'interpréteur Python. On peut le vérifier ainsi :

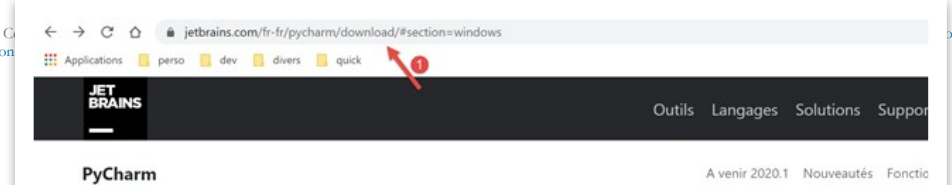


- en **[2]**, on quitte l'interpréteur Python ;
- en **[3]**, la commande qui affiche le PATH des exécutables de la machine Windows ;
- en **[4]**, on voit que le dossier de l'interpréteur Python 3.8 fait partie du PATH ;

2.2 L'IDE PyCharm Community

2.2.1 Introduction

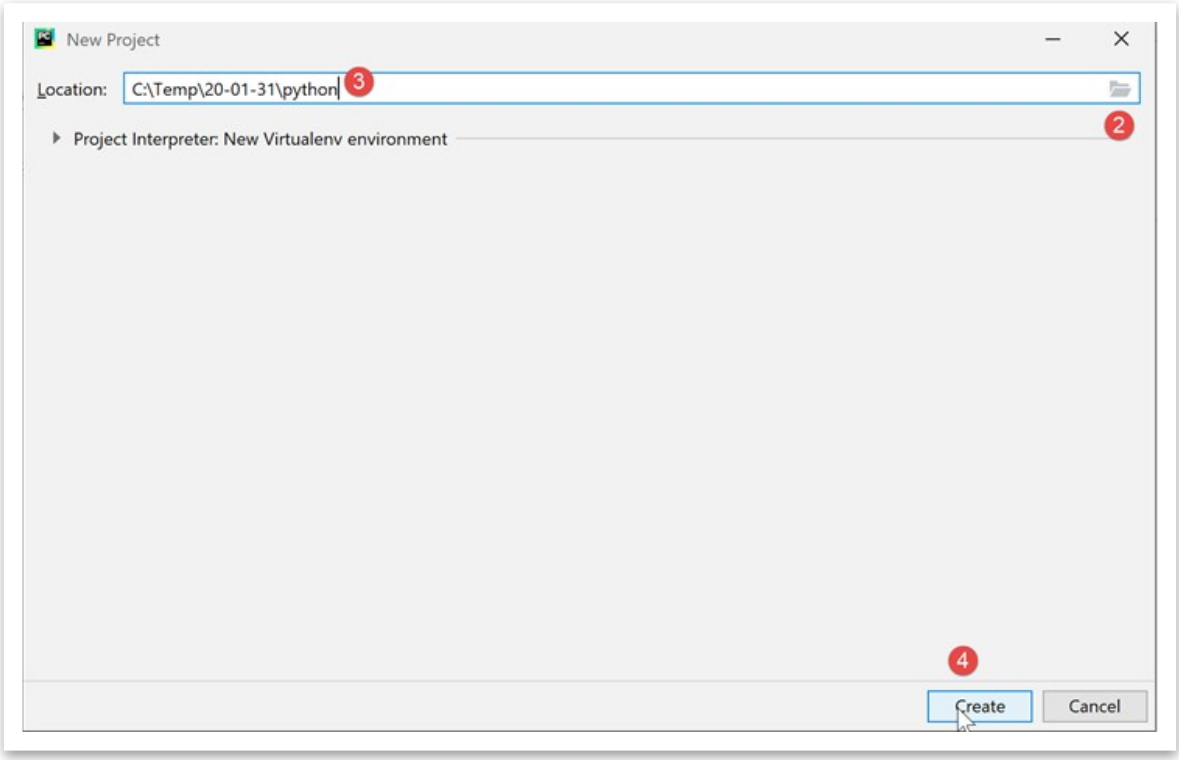
Pour construire et exécuter les scripts de ce document, nous avons utilisé l'éditeur **[PyCharm]** édition Community disponible (fév 2020) à l'URL | https://www.jetbrains.com/fr-fr/pycharm/download/#section=windows | :



Téléchargez l'IDE PyCharm Community **[1-3]** et installez-le.

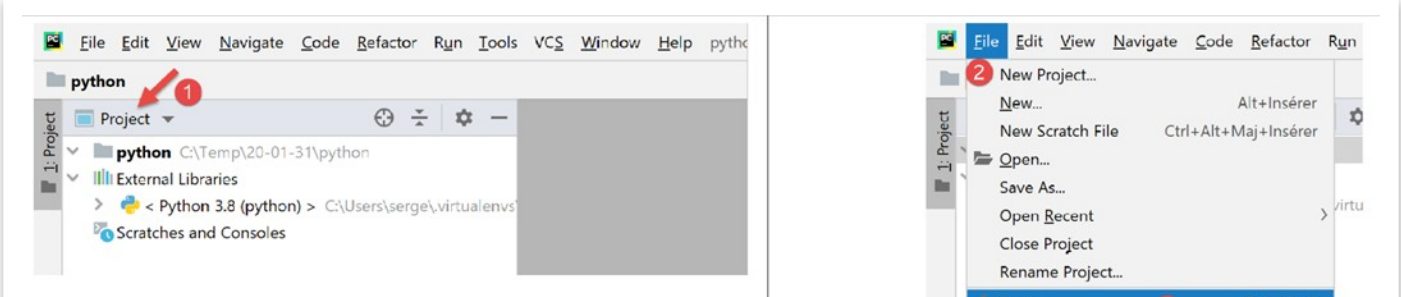
Lançons l'IDE PyCharm puis créons un premier projet Python :



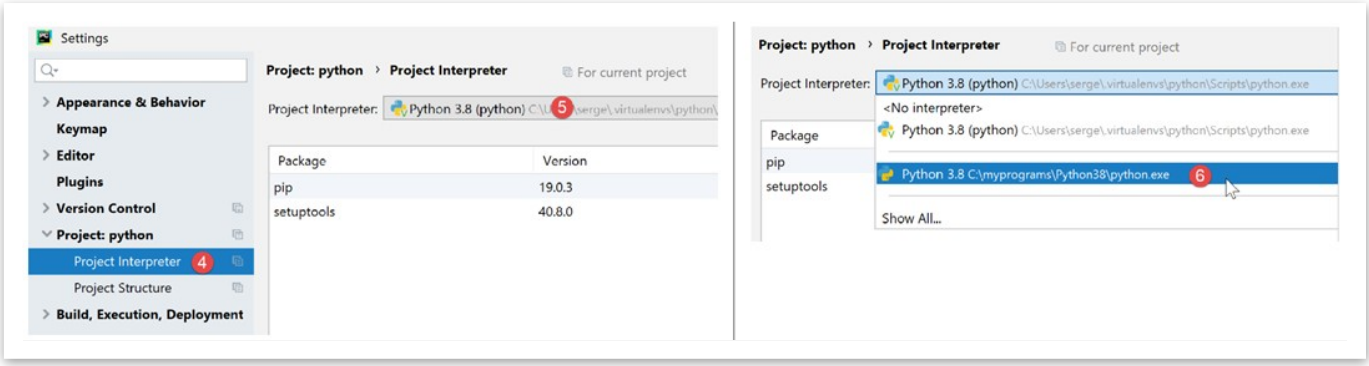


- en **[2-4]**, créez un nouveau projet ;

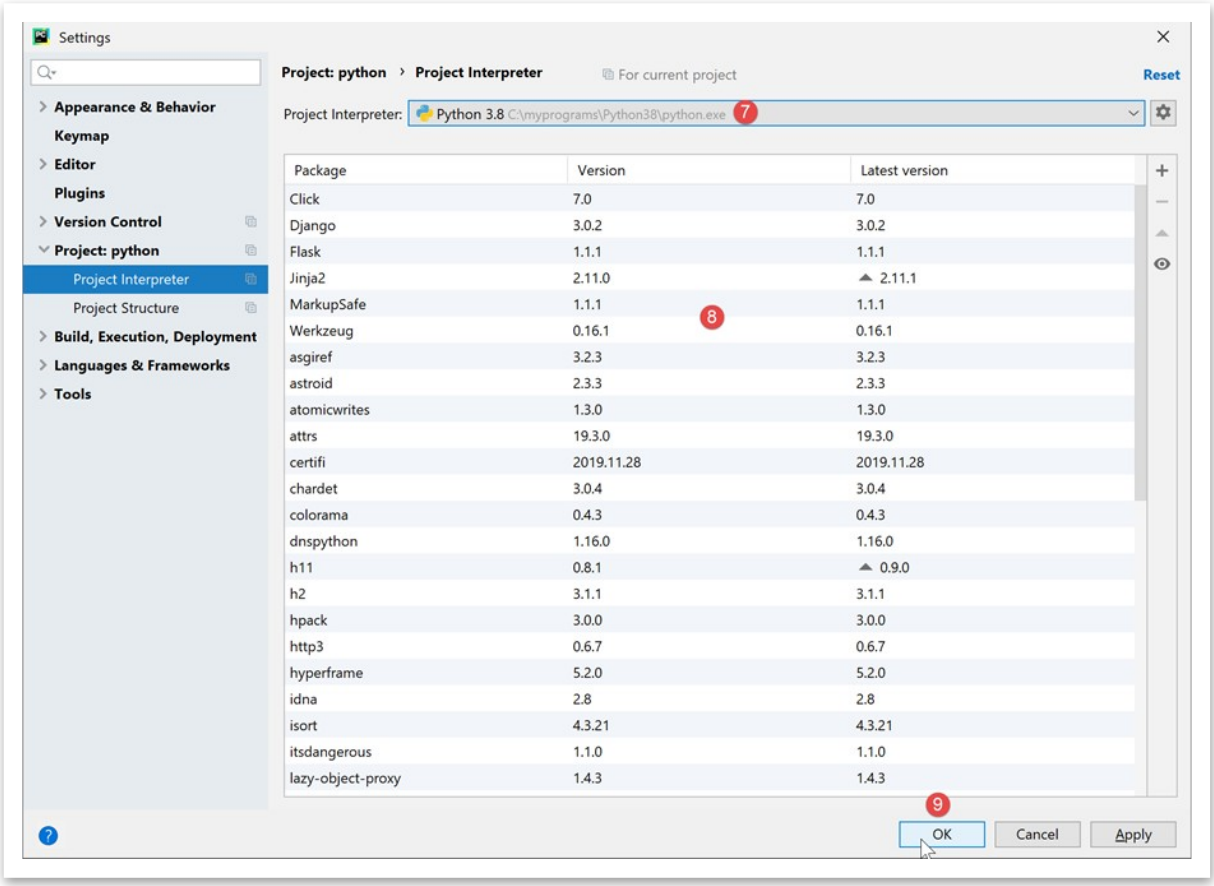
L'IDE PyCharm présente le projet créé sous la forme suivante :



- en **[2-3]**, examinons les propriétés de l'IDE ;

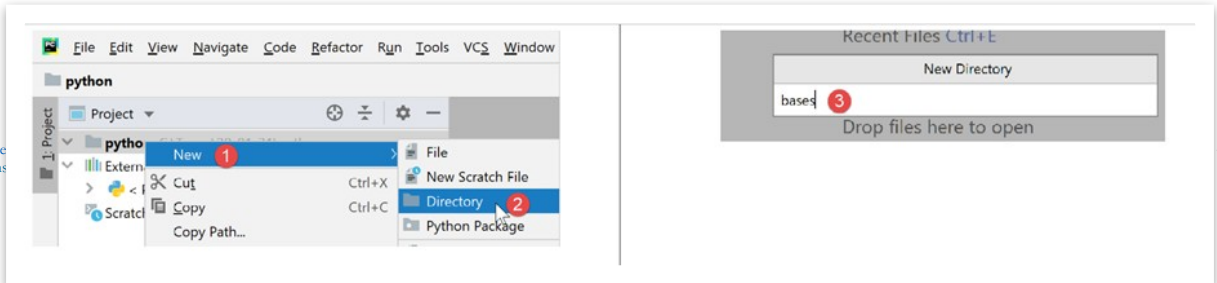


- en **[4]**, l'interpréteur Python qui sera utilisé pour le projet ;
- en **[5]**, une liste déroulante des interpréteurs disponibles ;
- en **[6]**, on choisit l'interpréteur téléchargé au paragraphe |Python 3.8.1| ;



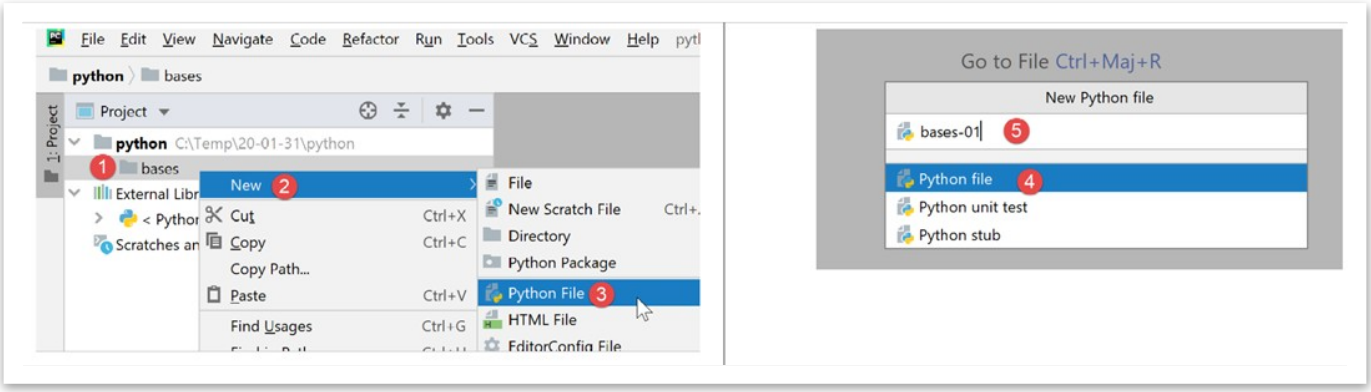
- en **[7]**, l'interpréteur choisi ;
- en **[8]**, la liste des packages disponibles avec cet interpréteur. Les packages contiennent des modules que les scripts Python peuvent utiliser. Il existe des centaines de modules disponibles ;

Commençons par créer un dossier dans lequel nous mettrons notre premier script Python :



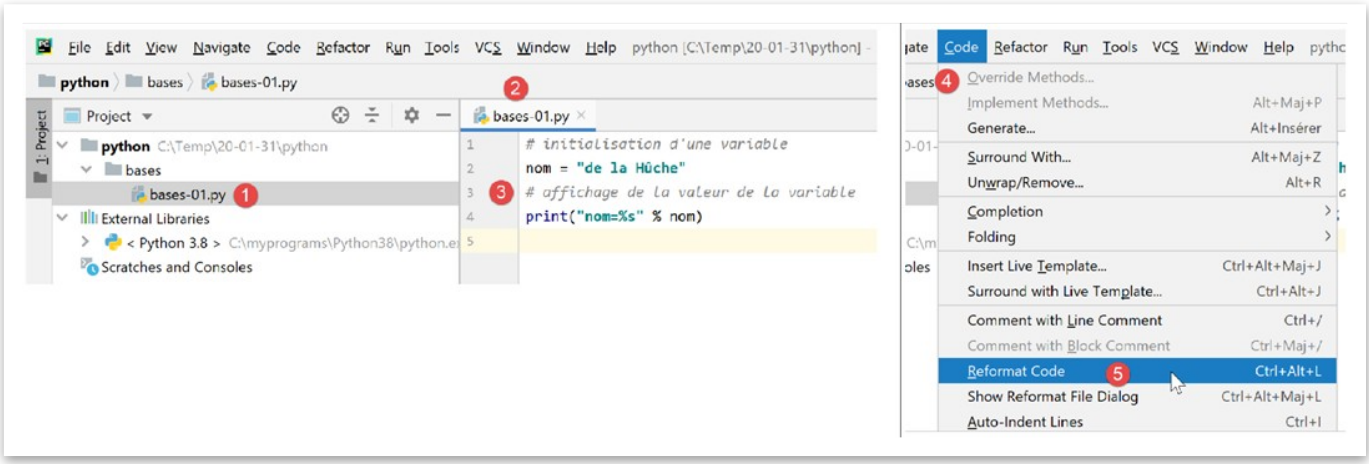
- cliquer droit sur le projet, puis **[1-2]** pour créer un dossier ;
- en **[3]**, tapez le nom du dossier : il sera créé dans le dossier du projet ;

Puis créons un script Python :



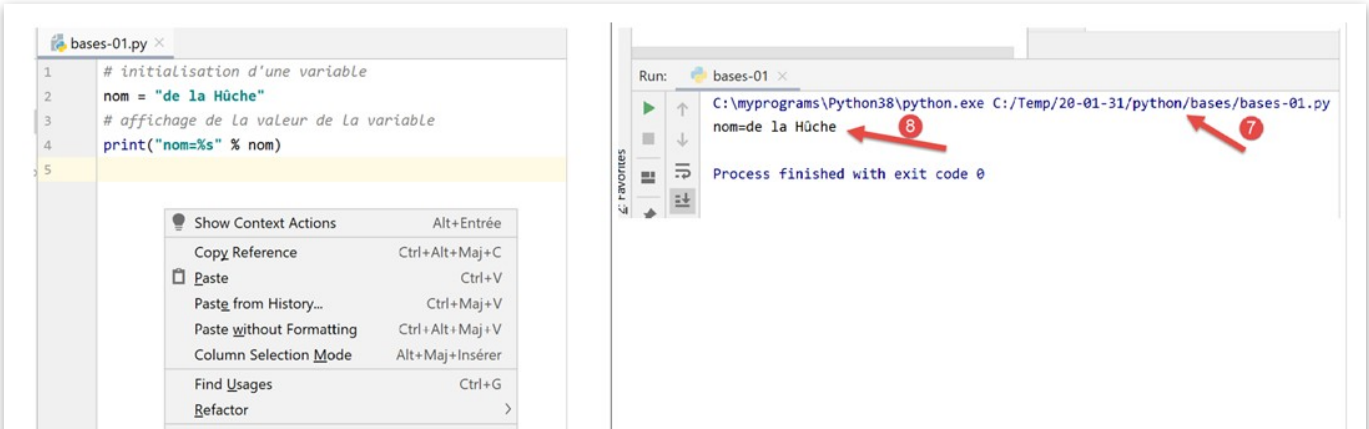
- cliquer droit sur le dossier **[bases]**, puis **[1-3]** ;
- en **[4-5]**, indiquez le nom du script ;

Écrivons notre premier script :



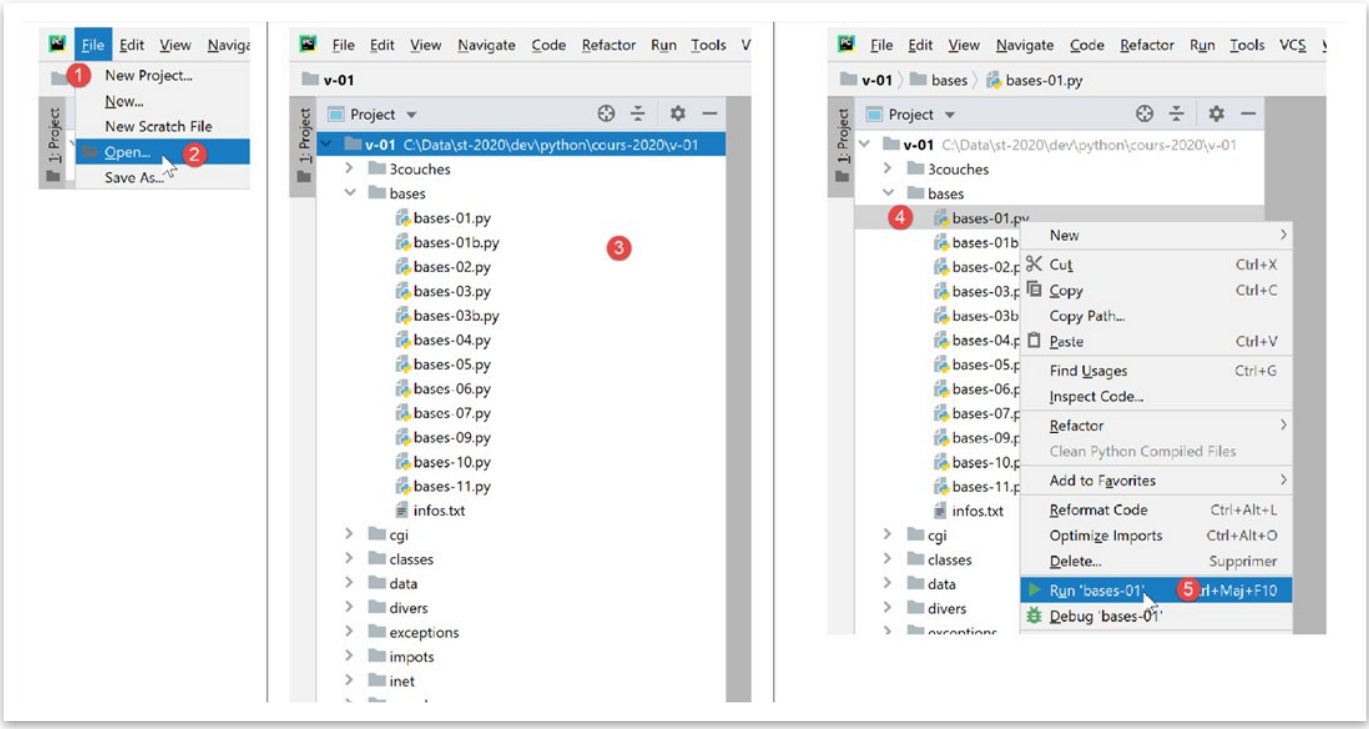
- en **[3]**, nous écrivons le script suivant :
  - lignes 1, 3 : les commentaires commencent avec le signe # ;
  - ligne 2 : initialisation d'une variable. Python ne déclare pas le type de ses variables ;
  - ligne 4 : affichage écran. La syntaxe utilisée ici est **[format % données]** avec :
    - **format** : nom=%s où %s désigne l'emplacement d'une chaîne de caractères. Celle-ci sera trouvée dans la partie **[données]** de l'expression ;
    - **données** : la valeur de la variable **[nom]** viendra remplacer le format %s dans la chaîne de format ;
- avec **[4-5]**, on reformate le code selon les recommandations de l'organisme gérant Python ;

Le script est exécuté avec un clic droit sur le code **[6]** :

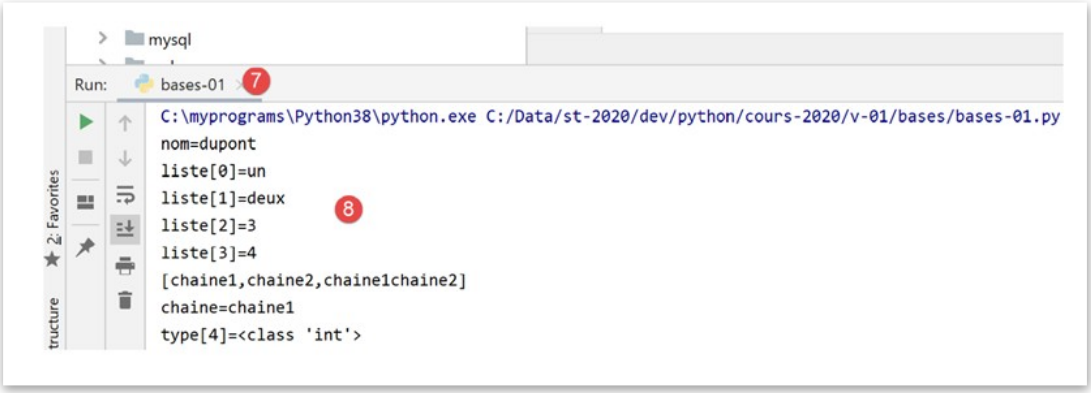


- en **[7]**, la commande exécutée ;
- en **[8]**, le résultat de l'exécution ;

Pour exécuter le script du document, téléchargez le code à l'URL |https://tahe.developpez.com/tutoriels-cours/python-flask-2020/documents/python-flask-2020.rar| puis dans PyCharm procédez comme suit :



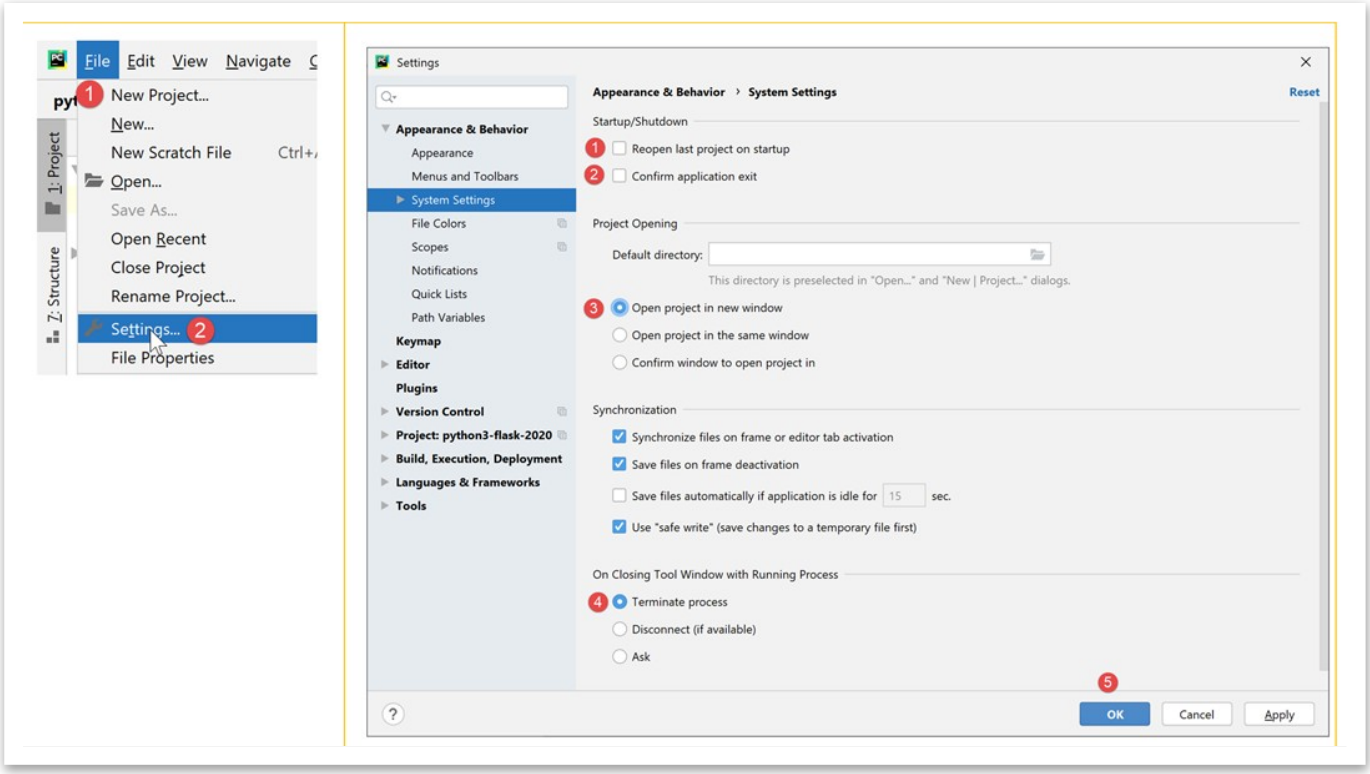
- en **[1-2]**, ouverture d'un projet existant : désigner le dossier du code téléchargé ;
- en **[3]**, le projet ouvert ;
- en **[4-5]**, exécution de l'un des scripts du projet ;



- en **[7-8]**, les résultats de l'exécution ;

2.2.2 Environnement virtuel d'exécution

Notre environnement de travail est désormais opérationnel. Nous allons cependant le modifier pour écrire les scripts de ce cours.

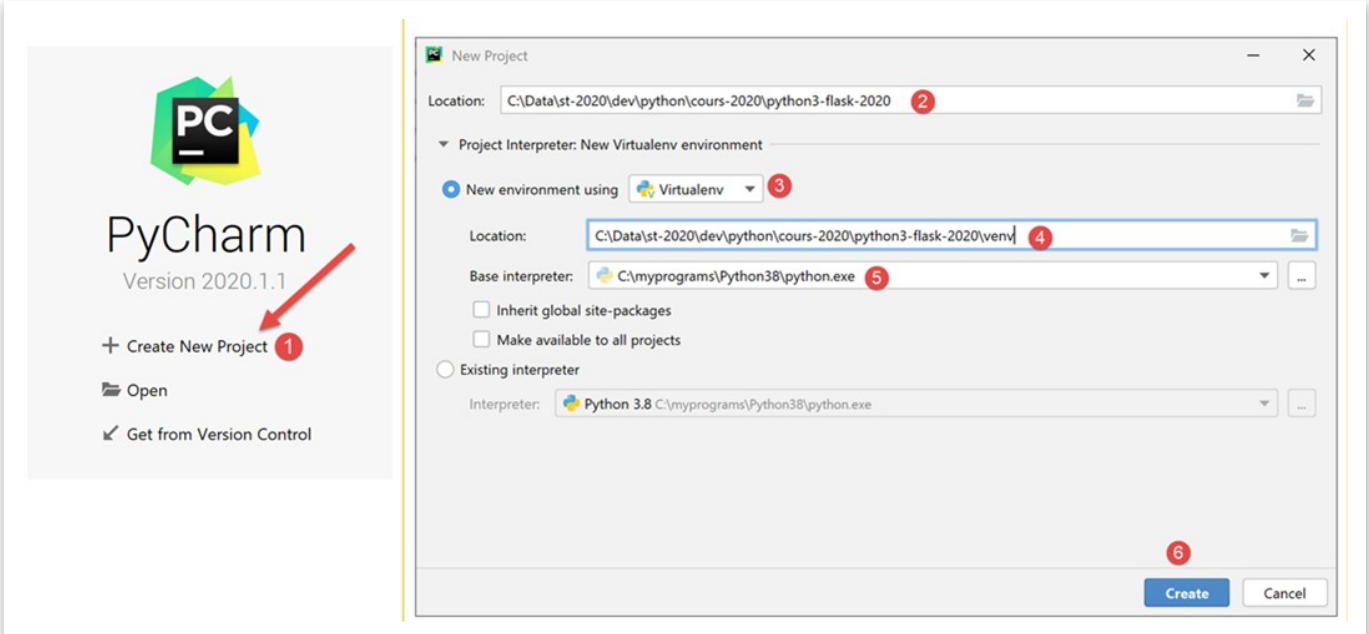


Tout d'abord, modifions la configuration de Pycharm :

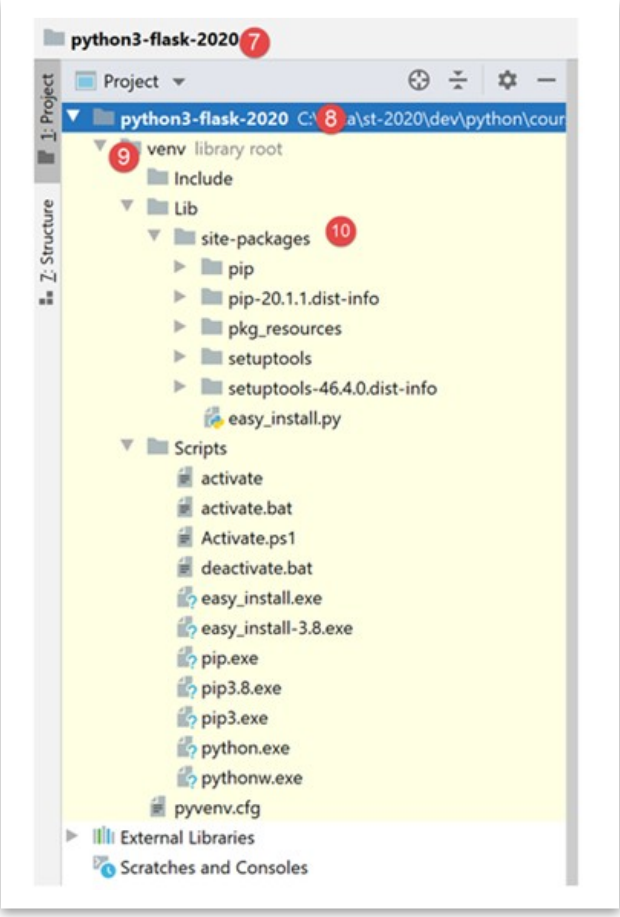
Dans la fenêtre de droite :

- par défaut, la case **[1]** est cochée. On la décoche pour que Pycharm n'ouvre pas par défaut le dernier projet ouvert mais nous laisse choisir celui qu'il faut ouvrir ;
- en **[2]**, on ne confirme pas la sortie de Pycharm lorsqu'on ferme la fenêtre de l'application ;
- en **[3]**, on ouvre les nouveaux projets dans une autre fenêtre ;
- en **[4]**, si on ferme Pycharm et qu'un programme est en cours d'exécution, celui-ci est arrêté ;

Fermons maintenant Pycharm puis rouvrons-le :



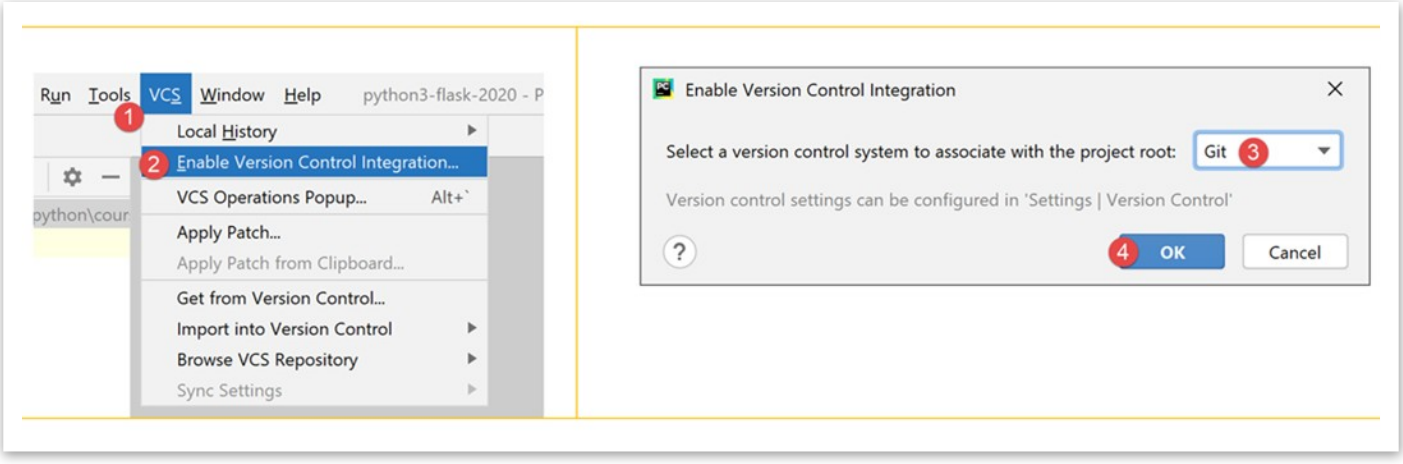
- en **[1]**, on crée un nouveau projet ;
- en **[2]**, indiquer le dossier du projet ;
- en **[3]**, prendre un environnement virtuel. Un environnement virtuel est propre au projet que l'on crée. Il ne se mélange pas avec les environnements virtuels d'autres projets. Un projet Python / Flask utilise de nombreuses bibliothèques externes qu'il faut installer. Un projet P1 peut utiliser une bibliothèque B dans sa version v1 et un projet P2 utiliser la même bibliothèque B mais dans sa version v2. Ces deux versions peuvent être plus ou moins comptatibles. Or lorsqu'on installe une bibliothèque dans sa version v2 et qu'une version v1 est déjà installée, celle-ci est écrasée par la version v2. Cela peut être problématique pour le projet qui utilisait la version v1 si la nouvelle version v2 n'est pas totalement compatible avec la version v1. Pour éviter ces problèmes, on isole chaque projet dans un environnement virtuel ;
- en **[4]**, désigner le dossier où seront rangées les bibliothèques python qui seront téléchargées au cours du projet. Nous avons choisi ici un dossier **[venv]** (virtual environment) à l'intérieur du dossier du projet. Rien n'oblige à faire cela ;
- en **[5]**, l'interpréteur Python du projet. C'est celui que nous avons installé dans l'étape précédente ;
- en **[6]**, créer le projet ;



- en **[7-8]**, le projet créé ;
- en **[9]**, l'environnement d'exécution du projet, appelé environnement virtuel ;
- en **[10]**, le dossier **[site-packages]** est le dossier dans lequel seront rangées les bibliothèques téléchargées ultérieurement ;

2.2.3 Git

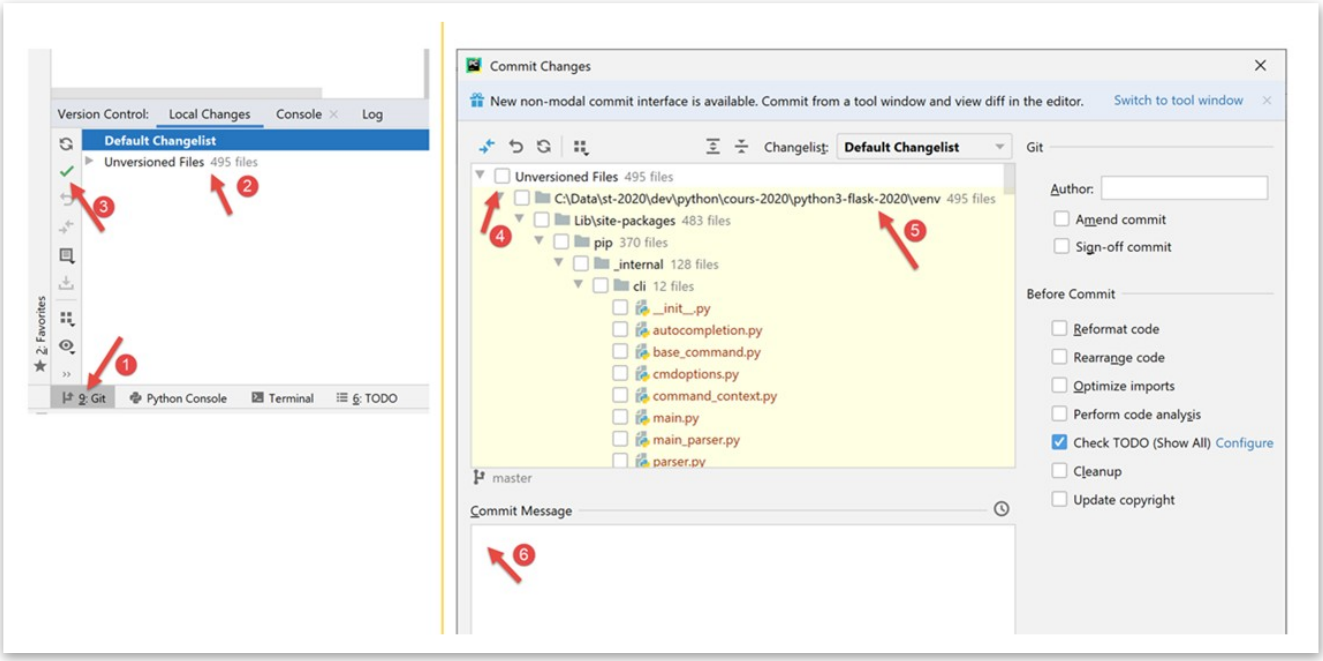
Ensuite, on active un logiciel de contrôle du code source. Ce sera ici Git **[1-4]** :



Un logiciel de contrôle du code source (VCS : Version Control System) permet de suivre les évolutions d'un projet. On peut prendre des photos, par une opération appelée **commit**, du projet à différents moments de sa vie. Si on fait deux commit aux temps T et T+1, le VCS permet de connaître ce qui a changé entre les deux versions commitées. Normalement le VCS est utilisé par une équipe de développeurs. Ceux-ci font des commit de leur code lorsque celui-ci a été dûment testé. A partir du VCS, les autres développeurs peuvent récupérer ce code validé et l'utiliser.

Ici, il n'y a qu'un développeur. L'expérience montre qu'on peut avoir une application qui marche au temps T et ne marche plus au temps T+1. On voudrait bien alors revenir en arrière au temps T pour repartir de zéro. Le VCS permet cela et c'est pour cette raison que l'on va l'utiliser ici.

Nous montrons comment procéder avec Git.



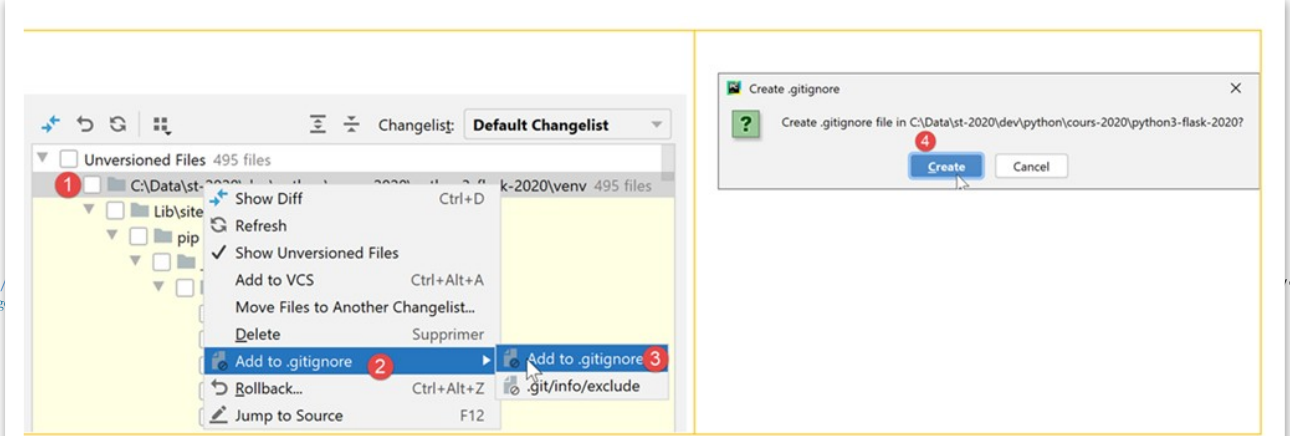
- en **[1]**, sélectionner l'onglet **[Git]** (en bas à gauche) ;
- en **[2]**, on voit qu'il y a 495 fichiers du projet qui ne sont pas versionnés par Git. Cela veut dire qu'ils ne feront pas partie des photos du projet, lors des commit ;
- en **[3]**, le bouton du **[commit]** ou bouton de validation du projet. Comme il a été dit, Git prend alors une photo du projet et la stocke dans un dossier **[.git]** à la racine du projet (non affiché par Pycharm mais visible dans l'explorateur windows) ;

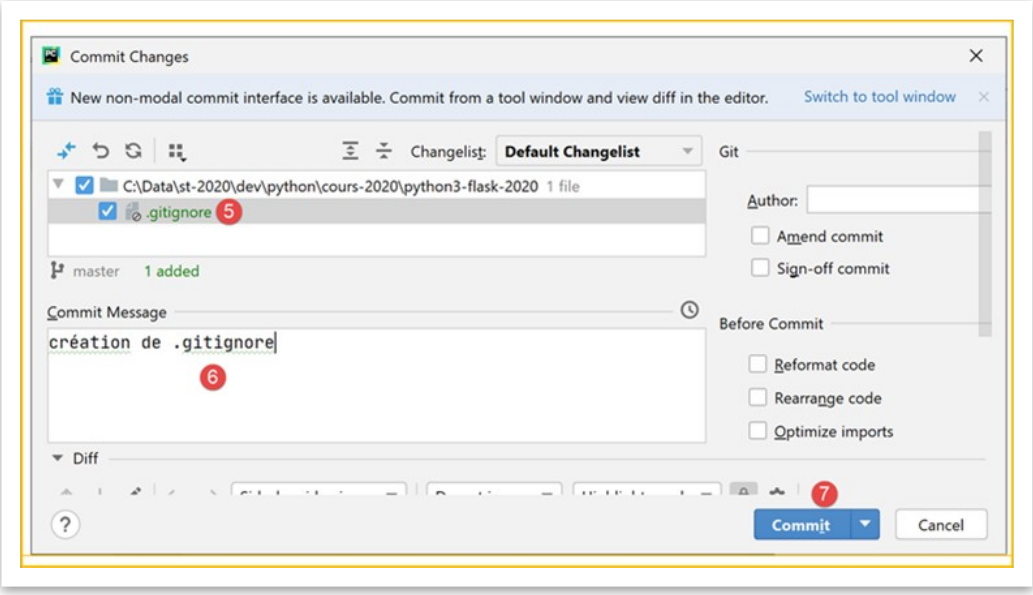
Validons **[3]** le projet dans son état actuel.

- en **[4]**, la liste des fichiers non versionnés ;
- en **[5]**, la totalité de ces fichiers non versionnés sont ceux de l'environnement virtuel **[venv]** ;
- en **[6]**, tout commit doit s'accompagner d'un message. Le développeur met ici les modifications amenées par la version qui va être commitée vis-à-vis de la dernière version commitée ;

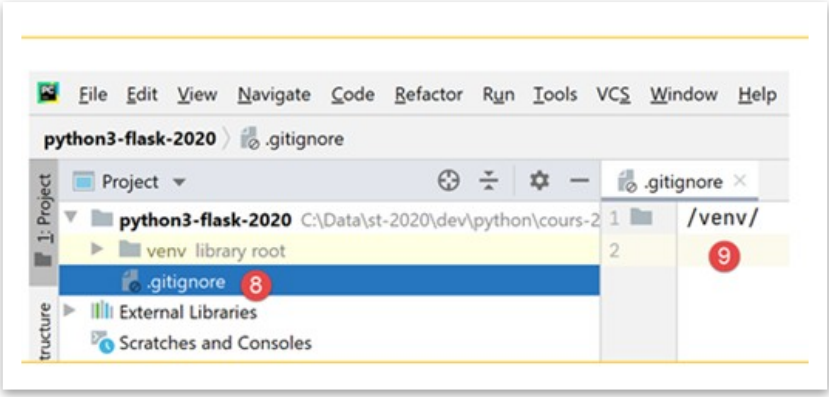
Certains dossiers ou fichiers peuvent être ignorés par Git. Ils ne font alors jamais partie de la photo. En **[5]** ci-dessus, cliquons droit sur le dossier **[venv]**.

- en **[1-3]**, on indique que le dossier **[venv]** ne doit pas faire partie des photos de Git. La liste des dossiers et fichiers ignorés par Git sont mis dans un fichier appelé **[.gitignore]** **[4]** ;



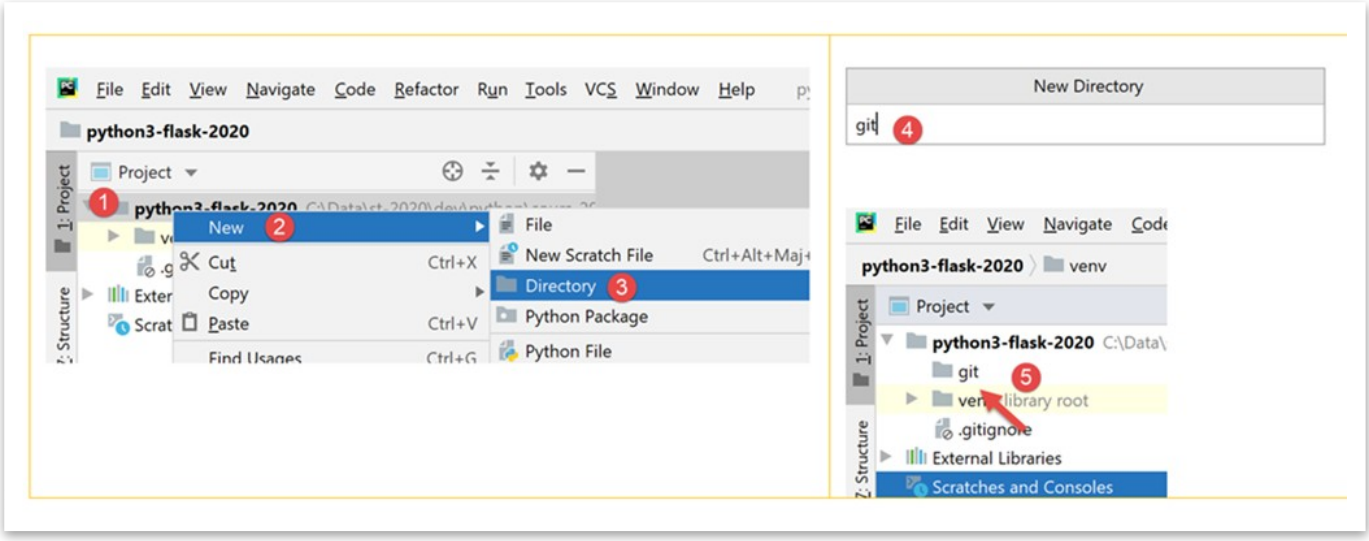


- après l'opération précédente, tous les fichiers du dossier **[venv]** disparaissent de la liste des fichiers non versionnés. Ne reste plus que le fichier **[.gitignore]** qui vient d'être créé ;
- en **[5]**, nous le sélectionnons pour qu'il soit sauvegardé ;
- en **[6]**, on crée un message de commit :
- en **[7]**, on valide. La photo du projet est alors prise ;

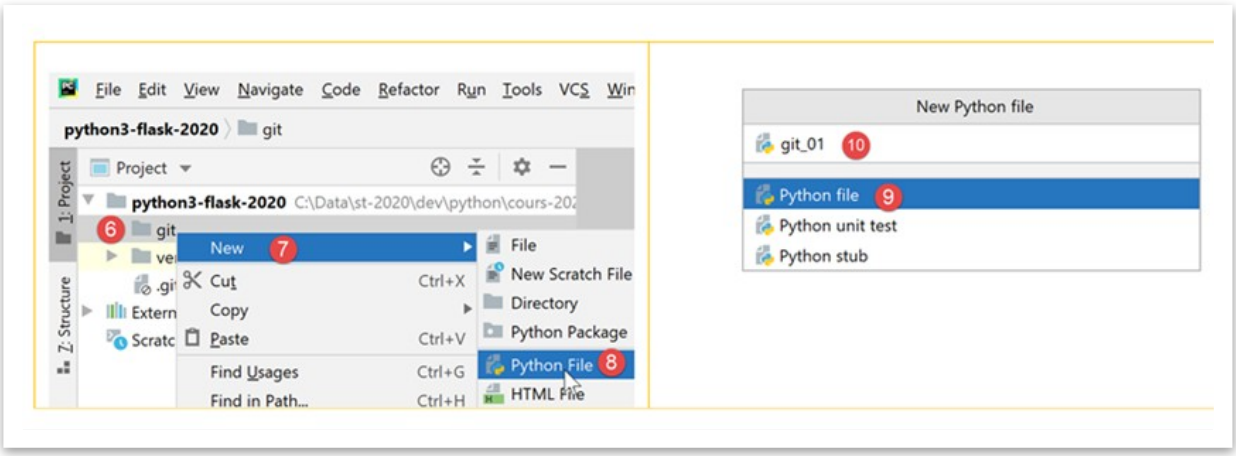


- en **[8-9]**, le contenu du fichier **[.gitignore]** : une seule ligne avec le nom du dossier **[/venv]** qui indique que son contenu doit être ignoré dans les photos ;

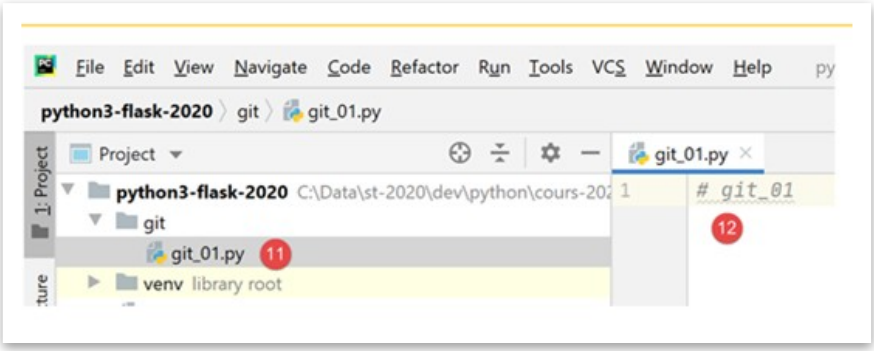
Montrons maintenant à quoi peut nous servir Git. Tout d'abord, nous créons un dossier **[git]** (vous pouvez utiliser tout autre nom – il pourra être détruit à la fin de la démonstration) :



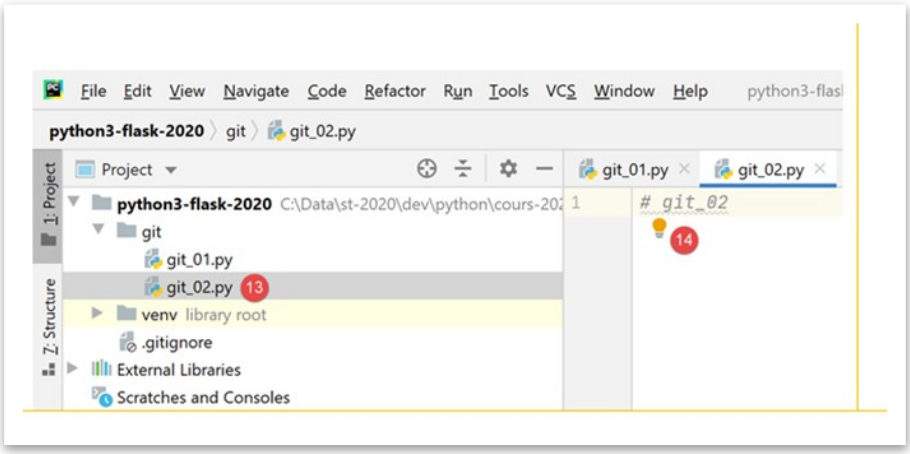
- en **[1-5]**, on crée un dossier **[git]** ;



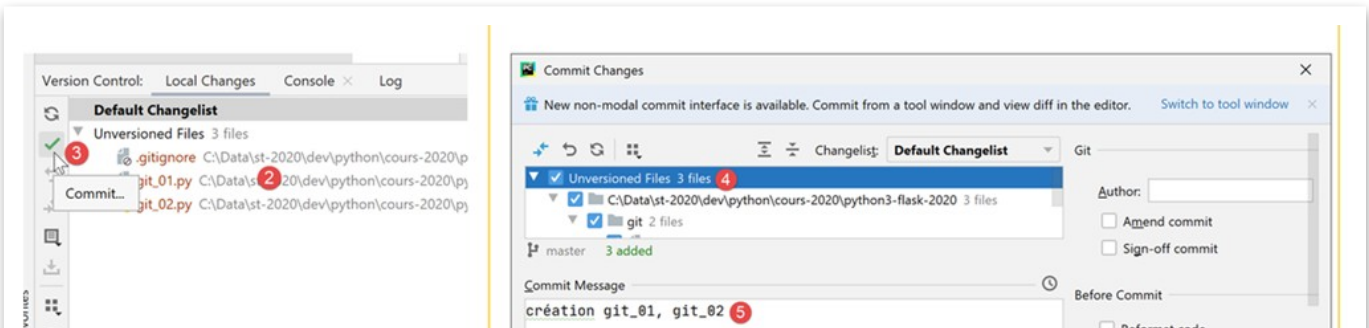
- en **[6-10]**, on crée un script Python **[git_01]** ;



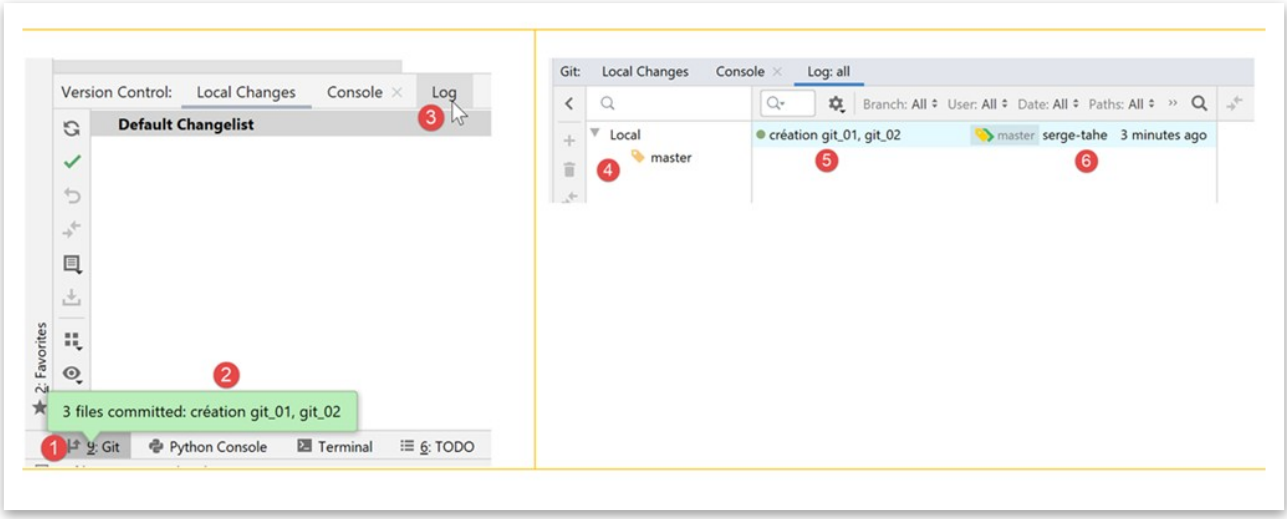
- en **[11-12]**, le script ne contient qu'un commentaire ;
- en **[13-14]**, on crée une copie de **[git_01]**. Pour cela, sélectionner **[git_01]** et faire Ctrl-C (Copier) puis (Ctrl-V) (Coller) et indiquer **[git_02]** comme nom de fichier ;



Maintenant, committons notre projet. Une photo va être prise avec les deux fichiers **[git_01, git_02]** ;

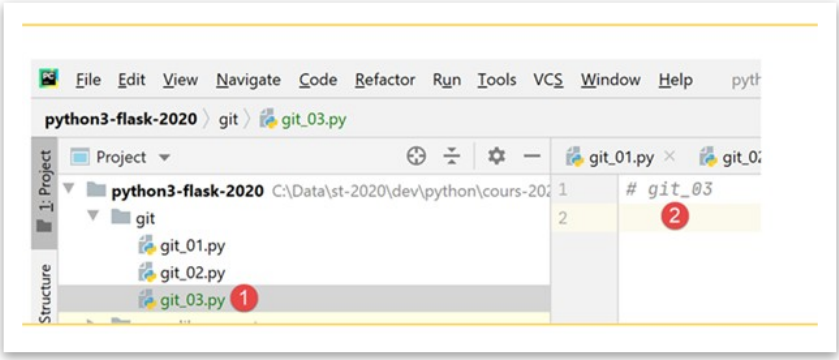


- en **[1-3]**, on committe le projet ;
- en **[4]**, on sélectionne les fichiers non versionnés qu'on veut committer, ici tous ;
- en **[5]**, on met le message identifiant le commit ;
- en **[6]**, on valide ;

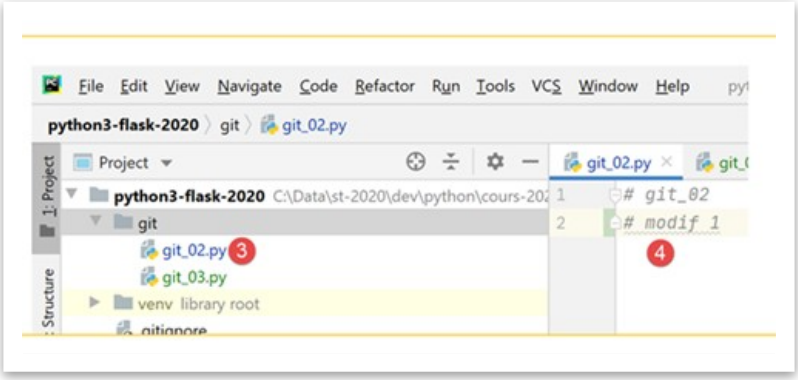


- en **[1-2]**, confirmation du commit ;
- en **[3]**, on sélectionne l'onglet **[Log]** ;
- en **[4]**, une vue des branches du projet. Ici, une seule branche appelée **[master]** ;
- en **[5-6]**, le commit qui vient d'être effectué ;

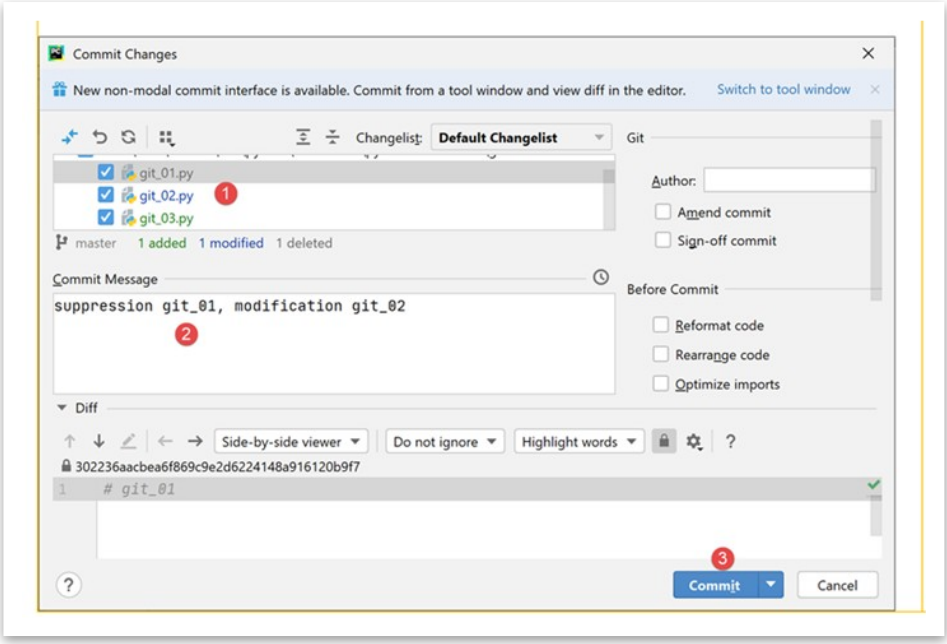
Maintenant créons de la même façon un script **[git_03]** :



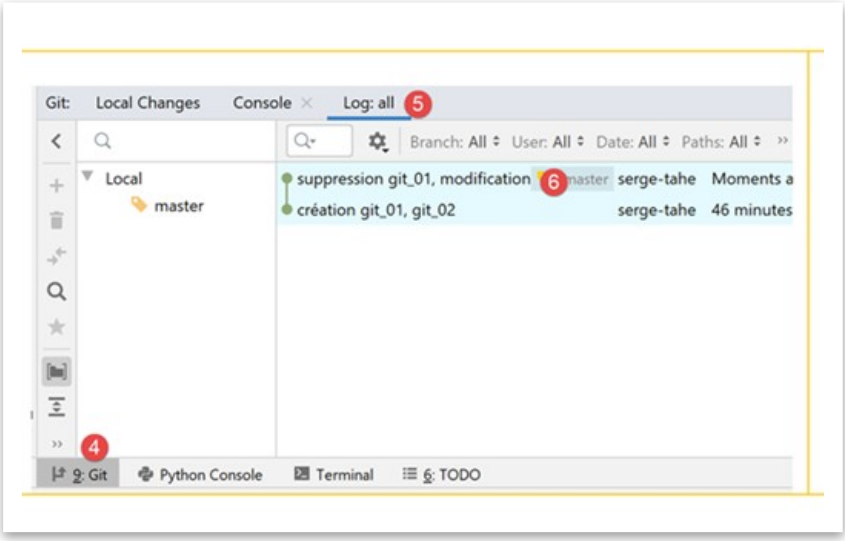
Modifions le script **[git_02]** et supprimons le script **[git_01]** :



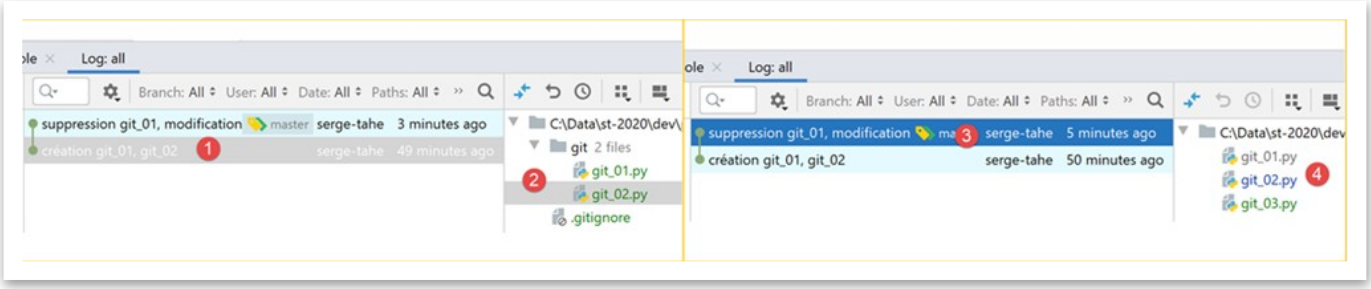
Puis committons la nouvelle version :



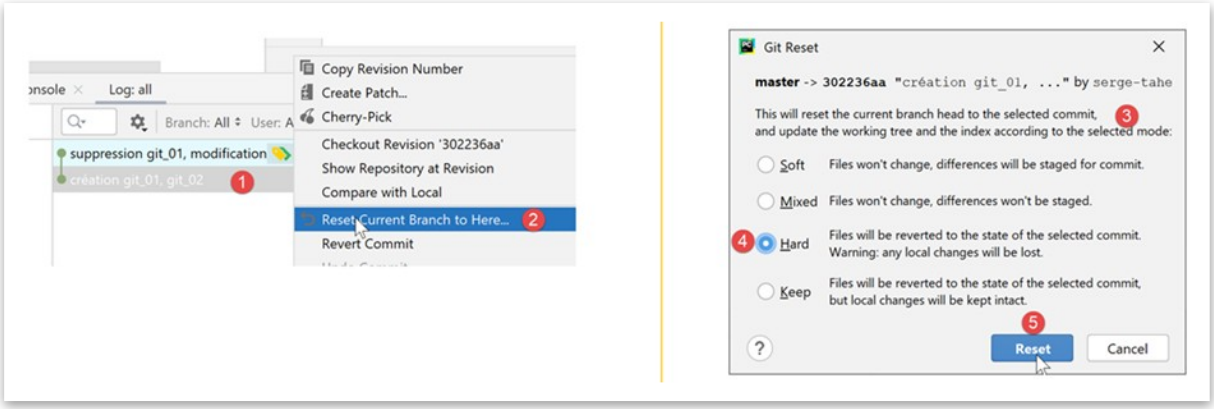
Maintenant nous avons deux commits dans les logs :



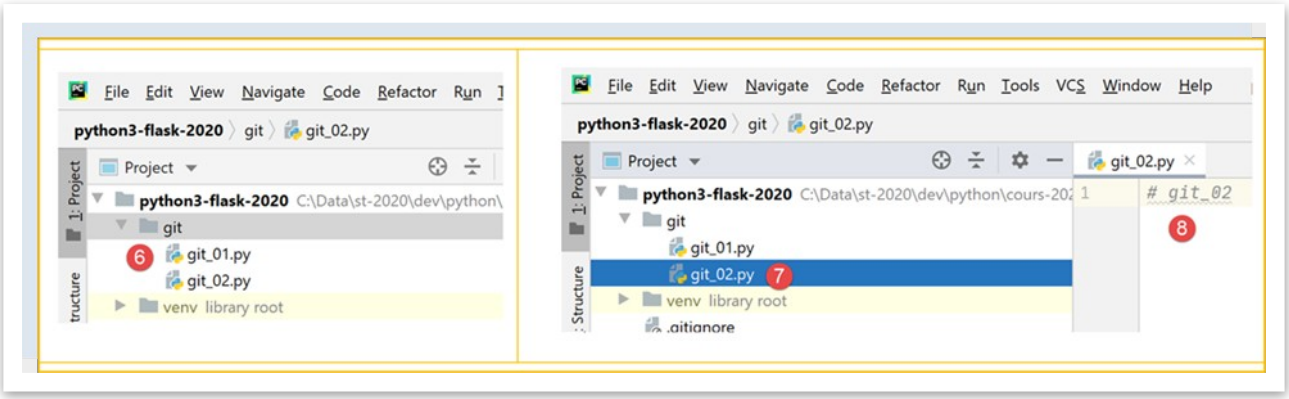
Lorsqu'on sélectionne un commit particulier, apparaît sur sa droite l'arbre des fichiers du projet :



Maintenant supposons que le dernier commit nous ait amenés à une impasse et que l'on voudrait revenir à une situation correspondant à l'un des commit précédents :

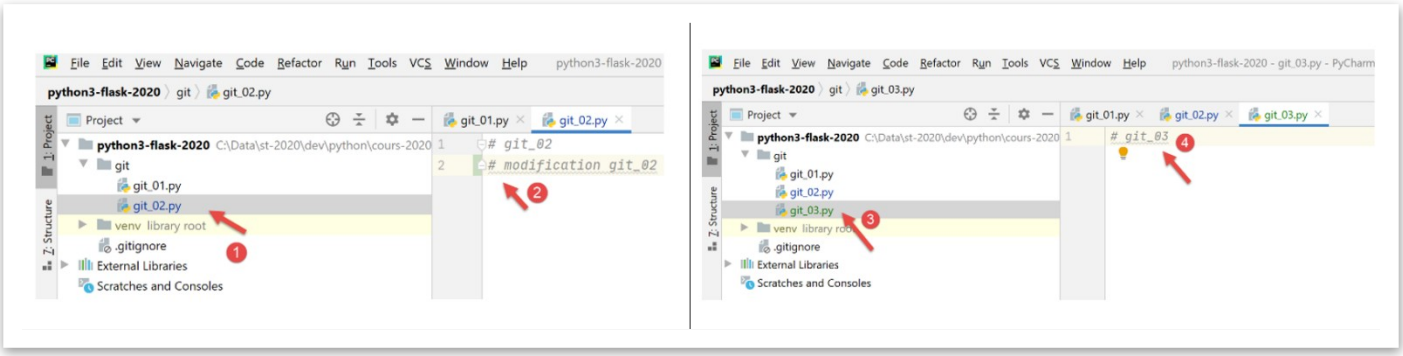


- en **[1-2]**, on sélectionne le commit dans l'état duquel on veut revenir ;
- en **[3-5]**, existent plusieurs modes de reset. Nous choisissons le mode **[hard]** qui revient à l'état sélectionné en perdant les changements qui ont été faits depuis ;

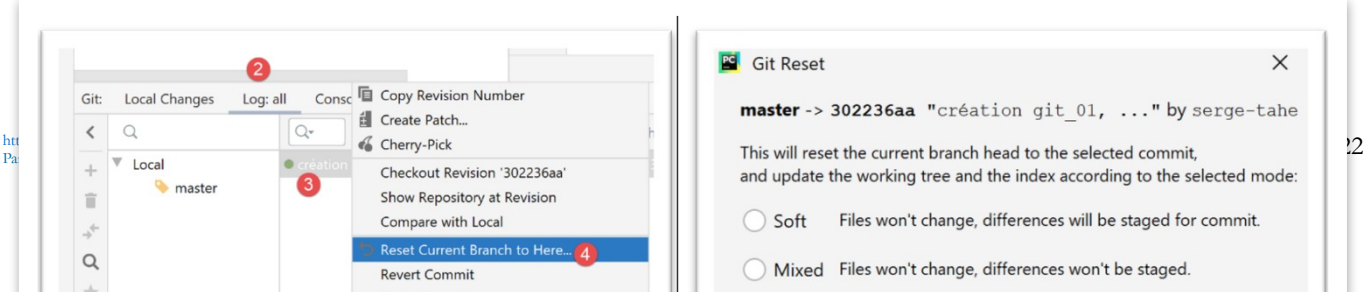


- en **[6]**, on a récupéré **[git_01]** qui avait été supprimé ;
- en **[7-8]**, on retrouve **[git_02]** dans son état original sans la modification qui avait été faite ;

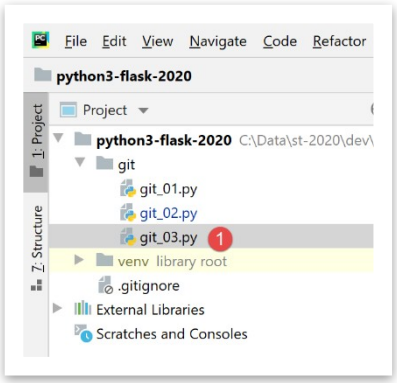
Maintenant, modifions **[git_02]** et ajoutons **[git_03] [1-4]** :

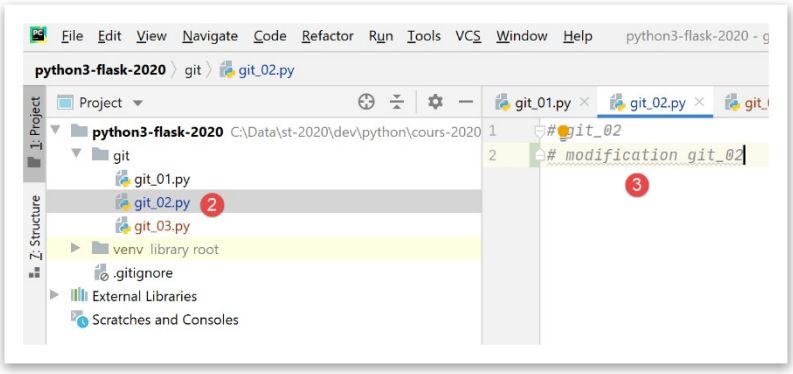


Maintenant, refaisons l'opération de revenir au commit initial :



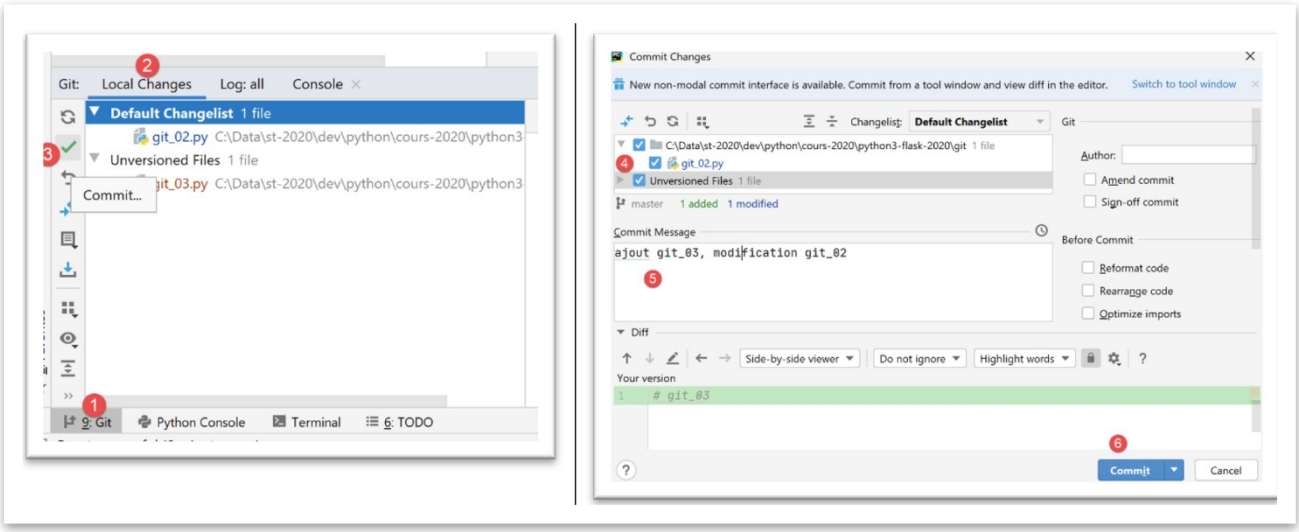
- en **[1-4]**, on revient à la photo du commit n° 1 ;
- en **[5-6]**, on prend l'option **[Keep]** au lieu de **[Hard]**. Ces options ne sont pas simples à comprendre. Aussi faut-il les essayer :



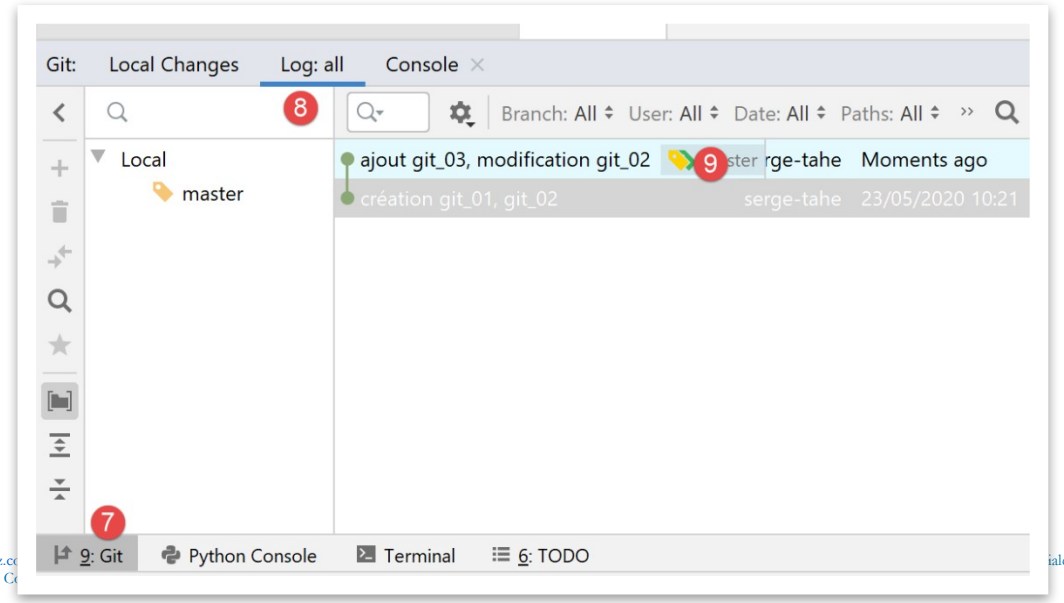


- en **[1]**, le fichier **[git_03]** est toujours là ;
- en **[2-3]**, le fichier **[git_02]** a gardé ses modifications ;

Difficile de dire ce qu'a fait ce **[Revert Commit]**. Maintenant committons la situation actuelle :

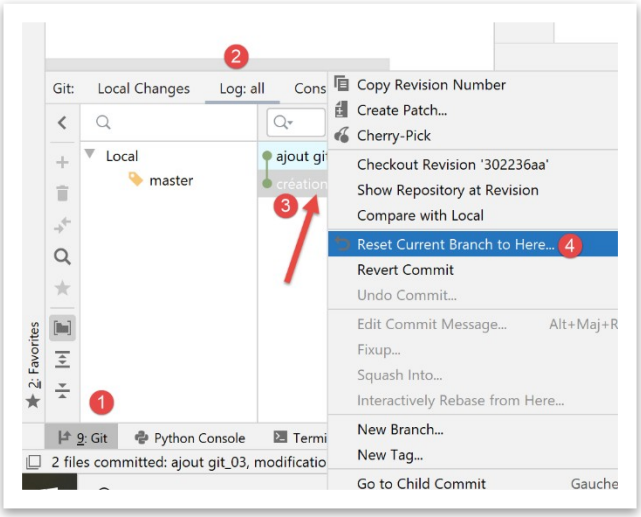


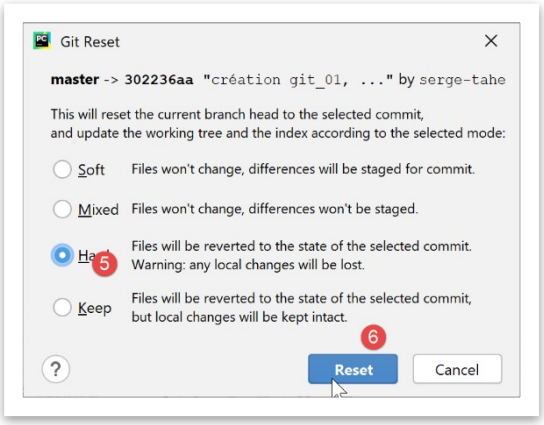
- en **[1-6]**, le commit ;



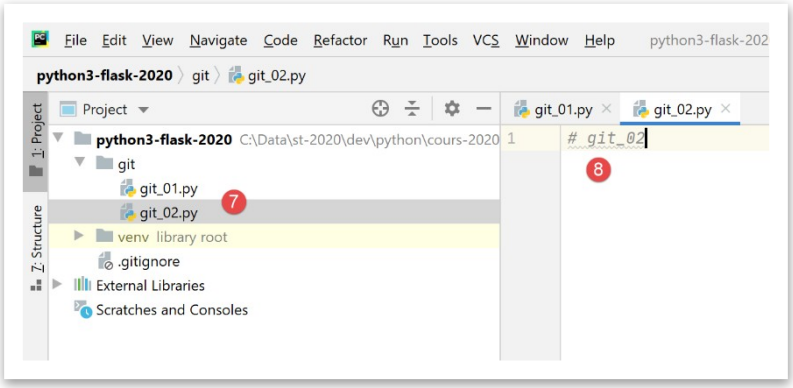
- en **[9]**, on voit le nouveau commit ;

Maintenant, essayons de revenir au commit 1 comme il a déjà été fait :





- en **[1-6]**, on revient au commit n° 1 en mode **[Hard]** ;



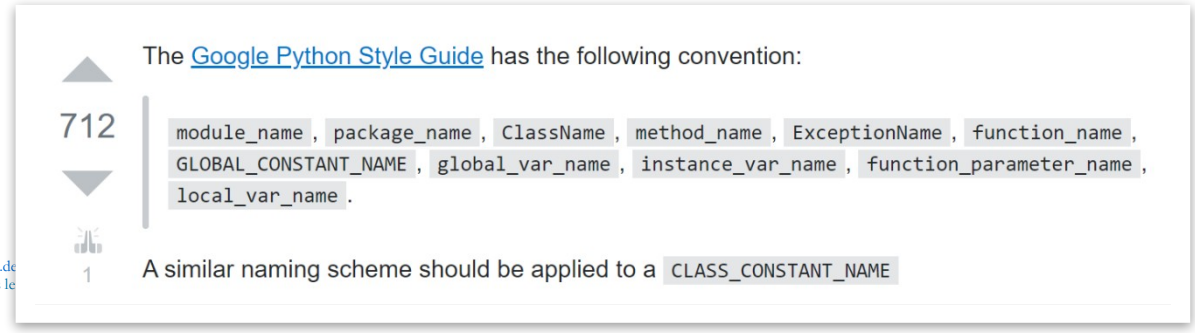
- en **[7-8]**, on a cette fois bien perdu toutes les modifications faites depuis le 1er commit ;

Dans la suite, nous ne reviendrons plus sur **[Git]**. Le lecteur pourra travailler de la façon suivante :

- il pourra suivre les exemples qui sont donnés soit en les tapant lui-même, soit en les récupérant sur le site du cours ;
- à chaque fois qu'un exemple fonctionne, il pourra committer son projet ;
- lorsqu'il développe ses propres codes et qu'il est dans une impasse, il sait qu'il peut revenir à une situation stable, en revenant à un commit précédent ;

2.3 Conventions d'écriture du code Python

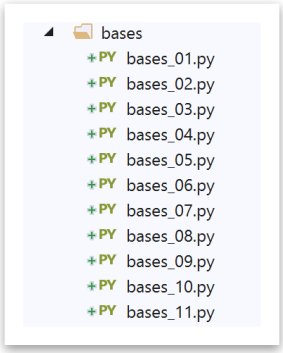
On peut écrire du code Python sans conventions d'écriture et ça ne l'empêchera pas de fonctionner. Mais il pourrait ne pas être apprécié de la communauté Python qui a édicté des conventions d'écriture. Celles-ci sont résumées dans un Post à l'adresse | https://stackoverflow.com/questions/159720/what-is-the-naming-convention-in-python-for-variable-and-function-names | :



- le nom d'un module (module_name) suit une convention appelée parfois **[snake_case]** : tout en minuscules comportant éventuellement des mots séparés par le caractère souligné. Cette convention **[snake_case]** s'applique aux noms de méthodes, de packages, de variables, de fonctions ;
- le nom d'une classe (ClassName) suit une convention appelée parfois **[PascalCase]** : une suite de mots collés avec chaque lettre de début de mot en majuscule ;
- les noms de constantes suivent la convention **[SNAKE_CASE]** : suite de mots en majuscules séparés par le caractère souligné ;

Ces conventions ont été globalement suivies dans ce document. Néanmoins pour les modules définissant une classe, j'ai donné au module le même nom que la classe qu'il contient. Il suit donc la convention **[PascalCase]** au lieu de **[snake_case]**. Je voulais pouvoir repérer rapidement les modules de classes.

3 Les bases de Python



3.1 Script [bases_01] : opérations élémentaires

Le script **[bases_01]** présente les premières caractéristiques de Python.

```
1.  # ---------------------------------
2.  def affiche(chaine):
3.      # affiche chaine
4.      print("chaine=%s" % chaine)
5.
6.
7.  # ---------------------------------
8.  def affiche_type(variable):
9.      # affiche le type de variable
10.     print("type[%s]=%s" % (variable, type(variable)))
11.
12.
13. # ---------------------------------
14. def f1(param):
15.     # ajoute 10 à param
16.     return param + 10
17.
18.
19. # ---------------------------------
20. def f2():
21.     # rend un tuple de 3 valeurs
22.     return "un", 0, 100
23.
24.
25. # ------------------------------- programme principal -----------------------------------
26. # ceci est un commentaire
27. # variable utilisée sans avoir été déclarée
28. nom = "dupont"
29.
30. # un affichage écran
31. print("nom=%s" % nom)
32.
33. # une liste avec des éléments de type différent
34. liste = ["un", "deux", 3, 4]
35.
36. # son nombre d'éléments
37. n = len(liste)
38.
39. # une boucle
40. for i in range(n):
41.     print("liste[%d]=%s" % (i, liste[i]))
42.
43. # initialisation de 2 variables avec un tuple
44. (chaine1, chaine2) = ("chaine1", "chaine2")
45.
46. # concaténation des 2 chaînes
47. chaine3 = chaine1 + chaine2
48.
49. # affichage résultat
```

```
50. print("[%s,%s,%s]" % (chaine1, chaine2, chaine3))
51. 
52. # utilisation fonction
53. affiche(chaine1)
54. 
55. # le type d'une variable peut être connu
56. affiche_type(n)
57. affiche_type(chaine1)
58. affiche_type(liste)
59. 
60. # le type d'une variable peut changer en cours d'exécution
61. n = "a changé"
62. affiche_type(n)
63. 
64. # une fonction peut rendre un résultat
65. res1 = f1(4)
66. print("res1=%s" % res1)
67. 
68. # une fonction peut rendre une liste de valeurs
69. (res1, res2, res3) = f2()
70. print("(res1,res2,res3)=[%s,%s,%s]" % (res1, res2, res3))
71. 
72. # on aurait pu récupérer ces valeurs dans une variable
73. liste = f2()
74. for i in range(len(liste)):
75.     print("liste[%s]=%s" % (i, liste[i]))
76. 
77. # des tests
78. for i in range(len(liste)):
79.     # n'affiche que les chaînes
80.     if type(liste[i]) == "str":
81.         print("liste[%s]=%s" % (i, liste[i]))
82. 
83. # d'autres tests
84. for i in range(len(liste)):
85.     # n'affiche que les entiers >10
86.     if type(liste[i]) == "int" and liste[i] > 10:
87.         print("liste[%s]=%s" % (i, liste[i]))
88. 
89. # une boucle while
90. liste = (8, 5, 0, -2, 3, 4)
91. i = 0
92. somme = 0
93. while i < len(liste) and liste[i] > 0:
94.     print("liste[%s]=%s" % (i, liste[i]))
95.     somme += liste[i]  # somme=somme+liste[i]
96.     i += 1  # i=i+1
97. print("somme=%s" % somme)
98. # fin programme
```

Commentaires

- ligne 2 : le mot clé **def** définit une fonction ;
- ligne 2 : la fonction reçoit le paramètre **[chaine]**. On n'indique pas le type du paramètre. Python utilise exclusivement le passage par **valeur.** Celle-ci diffère selon la donnée :
  - pour un type simple (nombre, booléen…), cette valeur est la **valeur** encapsulée par la donnée (4, True…) ;
  - pour un type complexe (liste, classe…), cette valeur est **l'adresse** de la donnée ;
- lignes 3-4 : le contenu de la fonction. Il est décalé à droite d'une tabulation. C'est cette indentation associée au caractère **:** de l'instruction *def* qui définit le contenu de la fonction. Cela est vrai pour toutes les instructions ayant du contenu : *if, else, while, for, try, except ;*
- ligne 4 : la syntaxe utilisée ici est **[print('text1%F1text2%F2…' % data1, data2)]** :
  - les **[%Fi]** (ici %s) sont des formats d'affichage :
  - %s (string) : pour une chaîne de caractères ;
  - %d (decimal) : pour les nombres entiers décimaux signés ;
  - %f (float) : format décimal pour les nombres réels ;
  - %e (exponentiel) : format exponentiel pour un nombre réel ;
  - …
- **[data1, data2…]** sont les expressions dont on veut afficher la valeur :
  - **[data1]** sera affichée avec le format F1 ;
  - **[data2]** sera affichée avec le format F2 ;
  - …

- ligne 10 : Python gère en interne le type des variables. On peut connaître le type d'une variable avec la fonction *type(variable)* qui rend une variable de type '*type*'. L'expression *'%s' % (type(variable))* est une chaîne de caractères représentant le type de la variable ;
- ligne 25 : le programme principal. Celui-ci vient habituellement (mais pas nécessairement) après la définition de toutes les fonctions du script. Son contenu est non indenté ;
- ligne 28 : en Python, on ne déclare pas les variables. Python est sensible à la casse. La variable *Nom* est différente de la variable *nom*. Une chaîne de caractères peut être entourée de guillemets " ou d'apostrophes '. On peut donc écrire '*dupont*' ou "*dupont*" ;
- ligne 34 : il y a une différence entre un **tuple** (1,2,3) (notez les parenthèses) et une **liste `[1,2,3]`** (notez les crochets). Le tuple est **non modifiable** alors que la liste l'est. Dans les deux cas, l'élement n° i est noté **`[i]`** ;
- ligne 40 : *range(n)* est le tuple (0,1,2,…,n-1) ;
- ligne 41 : le format %d est utilisé pour les nombres entiers signés ;
- ligne 74 : *len(var)* est le nombre d'éléments de la collection *var* (tuple, liste, dictionnaire…) ;
- ligne 84 : la structure **`[for in …]`** permet d'itérer une structure itérable. Les listes et les tuples sont des éléments itérables ;
- ligne 86 : les autres opérateurs booléens sont *or* et *not ;*
- ligne 93 : fait la somme des nombres >0 de la liste ;

Les résultats écran sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/bases/bases_01.py
2.  nom=dupont
3.  liste[0]=un
4.  liste[1]=deux
5.  liste[2]=3
6.  liste[3]=4
7.  [chaine1,chaine2,chaine1chaine2]
8.  chaine=chaine1
9.  type[4]=<class 'int'>
10. type[chaine1]=<class 'str'>
11. type[['un', 'deux', 3, 4]]=<class 'list'>
12. type[a changé]=<class 'str'>
13. res1=14
14. (res1,res2,res3)=[un,0,100]
15. liste[0]=un
16. liste[1]=0
17. liste[2]=100
18. liste[0]=8
19. liste[1]=5
20. somme=13
21.
22. Process finished with exit code 0
```

3.2 Script `[bases_02]` : chaînes de formatage

Python 3 a amené une nouvelle façon de formater les chaînes de caractères :

```
1.  # chaînes de formatage
2.  # les formats sont ceux du langage C
3.  # entier
4.  int1 = 10
5.  print(f"[int1={int1}]")
6.  print(f"[int1={int1:4d}]")
7.  print(f"[int1={int1:04d}]")
8.  # float
9.  float1=8.2
10. print(f"[float1={float1}]")
11. print(f"[float1={float1:8.2f}]")
12. print(f"[float1={float1:.3e}]")
13. # string
14. str1="abcd"
15. print(f"[str1={str1}]")
16. print(f"[str1={str1:8s}]")
17. str2="jean de florette"
18. print(f"[{str2:20.10s}]")
19. # les chaînes formatées peuvent être affectées à des variables
20. str3=f"[{str2:20.10s}]"
21. print(str3)
```

La syntaxe de la chaîne formatée est la suivante :

```
f'…{expr1:format1} …. {expr2:format2} ….'
```

avec :

- **[expri]** : une expression ;
- **[formati]** : le format de l'expression **[expri]**. Ces formats sont ceux du langage C :
- %d : pour les nombres entiers ;
- %f : notation décimale pour les nombres réels ;
- %e : notation exponentielle pour les nombres réels ;
- %s : pour les chaînes de caractères. C'est le format utilisé lorsqu'aucun format n'est utilisé pour **[expri]** ;
- %nd, %nf, %ns : affiche **[expri]** sur n caractères : la chaîne est soit tronquée, soit complétée avec des espaces;
- ligne 7 : **[04d]**, entier sur 4 caractères complétés à gauche avec des zéros;
- ligne 11 : **[8.2f]**, réel décimal sur 8 caractères dont 2 après la virgule ;
- ligne 12 : **[.3e]**, réel sous forme exponentielle avec 3 décimales pour la mantisse ;
- ligne 18 : **[20.10s]**, les 10 premiers caractères d'une chaîne complétée avec des espaces pour faire 20 caractères ;

Les résultats de l'exécution sont les suivants :

```
C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/bases/bases_02.py
[int1=10]
[int1=  10]
[int1=0010]
[float1=8.2]
[float1=    8.20]
[float1=8.200e+00]
[str1=abcd]
[str1=abcd    ]
[jean de fl          ]
[jean de fl          ]

Process finished with exit code 0
```

3.3 Script [bases_03] : changements de types

On s'intéresse ici aux changements de types avec des données de type **str** (chaîne de caractères), **int** (entier), **float** (réel), **bool** (booléen).

```
# changements de type
# int --> str, float, bool
x = 4
print(x, type(x))
x = str(4)
print(x, type(x))
x = float(4)
print(x, type(x))
x = bool(4)
print(x, type(x))

# bool --> int, float, str
x = True
print(x, type(x))
x = int(True)
print(x, type(x))
x = float(True)
print(x, type(x))
x = str(True)
print(x, type(x))

# str --> int, float, bool
x = "4"
print(x, type(x))
x = int("4")
print(x, type(x))
x = float("4")
print(x, type(x))
x = bool("4")
print(x, type(x))

# float --> str, int, bool
x = 4.32
```

```
34. print(x, type(x))
35. x = str(4.32)
36. print(x, type(x))
37. x = int(4.32)
38. print(x, type(x))
39. x = bool(4.32)
40. print(x, type(x))
41.
42. # gestion des erreurs de changement de type
43. try:
44.     x = int("abc")
45.     print(x, type(x))
46. except ValueError as erreur:
47.     print(erreur)
48.
49. # cas divers
50. x = bool("abc")
51. print(x, type(x))
52. x = bool("")
53. print(x, type(x))
54. x = bool(0)
55. print(x, type(x))
56. x = None
57. print(x, type(x))
58. x = bool(None)
59. print(x, type(x))
60. x = bool(0.0)
61. print(x, type(x))
62.
63. # toutes les données sont des instances de classe et à ce titre ont des méthodes
64. # chaîne de caractères
65. str1 = "abc"
66. print(str1.capitalize())
67. # nombre entier
68. int1 = 4
69. print(int1.bit_length())
70. # booléen
71. bool1=True
72. print(bool1.conjugate())
73. # nombre réel
74. float1=8.2
75. print (float1.is_integer())
```

De nombreux changements de type sont possibles. Certains peuvent échouer, comme celui des lignes 46-47 qui essaient de transformer la chaîne 'abc' en nombre entier. On a géré l'erreur avec une structure *try* / *except*. Une forme générale de cette structure est la suivante :

```
1.  try:
2.          actions
3.  except Exception as ex:
4.          actions
5.  finally:
6.          actions
```

Si l'une des actions du **try** lance une exception (signale une erreur), il y a branchement immédiat sur la clause **except**. Si les actions du **try** ne lancent pas d'exception, la clause **except** est ignorée. Les attributs *Exception* et *ex* de l'instruction *except* sont facultatifs. Lorsqu'ils sont présents, *Exception* précise le type d'exception interceptée par l'instruction *except* et *ex* contient l'exception qui s'est produite. Il peut y avoir plusieurs instructions *except*, si on veut gérer différents types d'exceptions dans le même *try*.
L'instruction *finally* est facultative. Si elle est présente, les actions du *finally* sont **toujours exécutées qu'il y ait eu exception ou non**.

Nous reviendrons sur les exceptions un peu plus loin.

Les lignes 49-61 montrent diverses tentatives pour transformer une donnée de type *str*, *int*, *float*, *NoneType* en booléen. C'est toujours possible. Les règles sont les suivantes :

- **bool(int i)** vaut *False* si i vaut 0, *True* dans tous les autres cas ;
- **bool(float f)** vaut *False* si f vaut 0.0, *True* dans tous les autres cas ;
- **bool(str chaine)** vaut *False* si chaine a 0 caractère, *True* dans tous les autres cas ;
- **bool(None)** vaut False. **None** est une valeur spéciale qui signifie que la variable existe mais n'a pas de valeur.

Les résultats écran sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/bases/bases_03.py
2.  4 <class 'int'>
3.  4 <class 'str'>
4.  4.0 <class 'float'>
5.  True <class 'bool'>
6.  True <class 'bool'>
7.  1 <class 'int'>
8.  1.0 <class 'float'>
9.  True <class 'str'>
10. 4 <class 'str'>
11. 4 <class 'int'>
12. 4.0 <class 'float'>
13. True <class 'bool'>
14. 4.32 <class 'float'>
15. 4.32 <class 'str'>
16. 4 <class 'int'>
17. True <class 'bool'>
18. invalid literal for int() with base 10: 'abc'
19. True <class 'bool'>
20. False <class 'bool'>
21. False <class 'bool'>
22. None <class 'NoneType'>
23. False <class 'bool'>
24. False <class 'bool'>
25. Abc
26. 3
27. 1
28. False
29.
30. Process finished with exit code 0
```

On notera que toutes les données sont des objets, c'est-à-dire des instances de classe. Cela signifie qu'elles peuvent avoir des méthodes. C'est ce que montrent les lignes 63-75 du code. Nous ne cherchons pas ici à expliquer ce que font les méthodes utilisées mais simplement à montrer qu'elles existent.

3.4 Script `[bases_04]`: portée des variables

Le script **`[bases_04]`** montre que Python n'a pas la notion de variable de portée bloc :

```
1.  # portée des variables
2.  i = 4
3.  if True:
4.      i += 1
5.      j = 7
6.  print(f"i={i}, j={j}")
```

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/bases/bases_04.py
2.  i=5, j=7
3.
4.  Process finished with exit code 0
```

Commentaires

Les résultats montrent deux choses :

- ligne 4 : la variable **`[i]`** du bloc **`[if]`** est la même que la variable i utilisée ligne 2 ;
- ligne 6 : la variable **`[j]`** est celle initialisée dans le bloc **`[if]`** ;

Dans certains langages, où on déclare les variables, une variable définie dans un bloc (comme celui des lignes 3-5) n'est pas connue à l'extérieur de celui-ci. En Python, rien de tel.

3.5 Script `[bases_05]` : listes - 1

Le script **`[bases_05]`** est le suivant :

```
1.  # listes à 1 dimension
2.  # initialisation
```

```
3.  list1 = [0, 1, 2, 3, 4, 5]
4.
5.  # parcours - 1
6.  print(f"list1 a {len(list1)} éléments")
7.  for i in range(len(list1)):
8.      print(f"list1[{i}]={list1[i]}")
9.
10. list1[1] = 10
11. # parcours - 2
12. print(f"list1 a {len(list1)} éléments")
13. for element in list1:
14.     print(element)
15.
16. # ajout de deux éléments
17. list1[len(list1):] = [10, 11]
18. # le format %s permet d'afficher la liste sur une ligne
19. print("%s" % list1)
20.
21. # suppression des deux derniers éléments
22. list1[len(list1) - 2:] = []
23. # le format par défaut permet d'afficher la liste sur une ligne
24. print(f"{list1}")
25.
26. # ajout en début de liste d'une liste
27. list1[:0] = [-10, -11, -12]
28. print(f"{list1}")
29.
30. # insertion en milieu de liste de deux éléments
31. list1[3:3] = [100, 101]
32. print(f"{list1}")
33.
34. # suppression de deux éléments en milieu de liste
35. list1[3:4] = []
36. print(f"{list1}")
```

Notes :

- la notation tableau**[i:j]** désigne les éléments **i** à **j-1** du tableau ;
- la notation **[i:]** désigne les éléments **i** et **suivants** du tableau ;
- la notation **[:i]** désigne les éléments **0** à **i-1** du tableau ;
- ligne 19 : print (%s) % (list1) affiche la chaîne de caractères : "**[list1[0]**, list1**[2]**…, list1**[n-1]**]" ;
- ligne 24 : la notation *print (f'{list1}')* fait la même chose ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/bases/bases_05.py
2.  list1 a 6 éléments
3.  list1[0]=0
4.  list1[1]=1
5.  list1[2]=2
6.  list1[3]=3
7.  list1[4]=4
8.  list1[5]=5
9.  list1 a 6 éléments
10. 0
11. 10
12. 2
13. 3
14. 4
15. 5
16. [0, 10, 2, 3, 4, 5, 10, 11]
17. [0, 10, 2, 3, 4, 5]
18. [-10, -11, -12, 0, 10, 2, 3, 4, 5]
19. [-10, -11, -12, 100, 101, 0, 10, 2, 3, 4, 5]
20. [-10, -11, -12, 101, 0, 10, 2, 3, 4, 5]
21.
22. Process finished with exit code 0
```

3.6 Script [bases_06] : listes - 2

Le code précédent peut être écrit différemment (**bases_06**) en utilisant certaines méthodes des listes :

```
1.  # listes à 1 dimension
```

```
2.
3.  # initialisation
4.  list1 = [0, 1, 2, 3, 4, 5]
5.
6.  # parcours - 1
7.  print(f"list1 a {len(list1)} éléments")
8.  for i in range(len(list1)):
9.      print(f"list1[{i}]={list1[i]}")
10.
11. # modification d'un élément
12. list1[1] = 10
13.
14. # parcours - 2
15. print(f"list1 a {len(list1)} éléments")
16. for element in list1:
17.     print(element)
18.
19. # ajout de deux éléments
20. list1.extend([10, 11])
21. print(f"{list1}")
22.
23. # suppression des deux derniers éléments
24. del list1[len(list1) - 2:]
25. print(f"{list1}")
26.
27. # ajout en début de liste d'un tuple
28. for i in (-12, -11, -10):
29.     list1.insert(0, i)
30. print(f"{list1}")
31.
32. # insertion en milieu de liste
33. for i in (101, 100):
34.     list1.insert(3, i)
35. print(f"{list1}")
36.
37. # suppression en milieu de liste
38. del list1[3:4]
39. print(f"{list1}")
```

Les résultats obtenus sont les mêmes qu'avec la version précédente.

3.7 script [bases_07] : le dictionnaire

Le script **[bases_07]** montre comment définir et exploiter un dictionnaire, parfois appelé tableau associatif.

```
1.  # une fonction qui vérifie si la clé mari existe dans le dictionnaire conjoints
2.  def existe(conjoints, mari):
3.      if mari in conjoints:
4.          print(f"La clé [{mari}] existe associée à la valeur [{conjoints[mari]}]")
5.      else:
6.          print(f"La clé [{mari}] n'existe pas")
7.
8.
9.  # ---------------------------- Main
10. # un dictionnaire
11. conjoints = {"Pierre": "Gisèle", "Paul": "Virginie", "Jacques": "Lucette", "Jean": ""}
12.
13. # parcours - 1
14. print(f"Nombre d'éléments du dictionnaire : {len(conjoints)}")
15. for (clé, valeur) in conjoints.items():
16.     print(f"conjoints[{clé}]={valeur}")
17.
18. # liste des clés du dictionnaire
19. print("liste des clés-------------")
20. clés = conjoints.keys()
21. print(f"{clés}")
22.
23. # liste des valeurs du dictionnaire
24. print("liste des valeurs------------")
25. valeurs = conjoints.values()
26. print(f"{valeurs}")
27.
28. # recherche d'une clé
29. existe(conjoints, "Jacques")
30. existe(conjoints, "Lucette")
31. existe(conjoints, "Jean")
```

```
32.
33. # suppression d'une clé-valeur
34. del (conjoints["Jean"])
35. print(f"Nombre d'éléments du dictionnaire : {len(conjoints)}")
36. print(f"{conjoints}")
37.
38. # les clés et valeurs d'un dictionnaire ne sont pas des listes
39. print(f"type des clés : {type(clés)}")
40. print(f"type des valeurs : {type(valeurs)}")
41.
42. # on peut les transformer en listes
43. lclés = list(clés)
44. print(f"clés : {type(lclés)}, {lclés}")
45. lvaleurs = list(valeurs)
46. print(f"valeurs : {type(lvaleurs)}, {lvaleurs}")
```

Notes :

- ligne 11 : la définition en dur d'un dictionnaire ;
- ligne 15 : *conjoints.items()* rend la liste des couples (clé,valeur) du dictionnaire *conjoints ;*
- ligne 20 : *conjoints.keys()* rend les clés du dictionnaire *conjoints ;*
- ligne 25 : *conjoints.values()* rend les valeurs du dictionnaire *conjoints ;*
- ligne 3 : *mari in conjoints* rend *True* si la clé *mari* existe dans le dictionnaire *conjoints*, *False* sinon ;
- ligne 36 : un dictionnaire peut être affiché en une seule ligne.

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/bases/bases_07.py
2.  Nombre d'éléments du dictionnaire : 4
3.  conjoints[Pierre]=Gisèle
4.  conjoints[Paul]=Virginie
5.  conjoints[Jacques]=Lucette
6.  conjoints[Jean]=
7.  liste des clés-------------
8.  dict_keys(['Pierre', 'Paul', 'Jacques', 'Jean'])
9.  liste des valeurs------------
10. dict_values(['Gisèle', 'Virginie', 'Lucette', ''])
11. La clé [Jacques] existe associée à la valeur [Lucette]
12. La clé [Lucette] n'existe pas
13. La clé [Jean] existe associée à la valeur []
14. Nombre d'éléments du dictionnaire : 3
15. {'Pierre': 'Gisèle', 'Paul': 'Virginie', 'Jacques': 'Lucette'}
16. type des clés : <class 'dict_keys'>
17. type des valeurs : <class 'dict_values'>
18. clés : <class 'list'>, ['Pierre', 'Paul', 'Jacques']
19. valeurs : <class 'list'>, ['Gisèle', 'Virginie', 'Lucette']
20.
21. Process finished with exit code 0
```

Notes :

- on notera aux lignes 16-17 des résultats que les clés et valeurs d'un dictionnaire ne forment pas une liste mais un type 'dict_keys' ;
- lignes 18-19 : un simple changement de type permet de les convertir en un type **[list]** ;

3.8 script [bases_08] : les tuples

Le **tuple** a des similitudes avec la liste mais est non modifiable :

```
1.  # tuples
2.  # initialisation
3.  tuple1 = (0, 1, 2, 3, 4, 5)
4.
5.  # parcours - 1
6.  print(f"tuple1 a {len(tuple1)} elements")
7.  for i in range(len(tuple1)):
8.      print(f"tuple1[{i}]={tuple1[i]}")
9.
10. # parcours - 2
11. print(f"tuple1 a {len(tuple1)} elements")
12. for element in tuple1:
13.     print(element)
14.
```

```
15. # un tuble peut être affiché en une ligne
16. print(f"tuple1={tuple1}")
17.
18. # un tuple ne peut être modifié
19. tuple1[0] = 10
```

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/bases/bases_08.py
2.  tuple1 a 6 elements
3.  tuple1[0]=0
4.  tuple1[1]=1
5.  tuple1[2]=2
6.  tuple1[3]=3
7.  tuple1[4]=4
8.  tuple1[5]=5
9.  tuple1 a 6 elements
10. 0
11. 1
12. 2
13. 3
14. 4
15. 5
16. tuple1=(0, 1, 2, 3, 4, 5)
17. Traceback (most recent call last):
18.   File "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/bases/bases_08.py", line 19, in <module>
19.     tuple1[0] = 10
20. TypeError: 'tuple' object does not support item assignment
21.
22. Process finished with exit code 1
```

Notes :

- lignes 17-20 des résultats : montrent qu'un tuple ne peut pas être modifié.

3.9 Script `[bases_09]` : les listes et dictionnaires à plusieurs dimensions

Le script **`[bases_09]`** montre comment définir et exploiter une liste ou un dictionnaire multidimensionnel :

```
1.  # listes multidimensionnelles
2.  # initialisation
3.  multi = [[0, 1, 2], [10, 11, 12, 13], [20, 21, 22, 23, 24]]
4.
5.  # parcours
6.  for i1 in range(len(multi)):
7.      for i2 in range(len(multi[i1])):
8.          print(f"multi[{i1}][{i2}]={multi[i1][i2]}")
9.
10. # affichage en une ligne
11. print(f"multi={multi}")
12.
13. # dictionnaires multidimensionnels
14. # initialisation
15. multi = {"zéro": [0, 1], "un": [10, 11, 12, 13], "deux": [20, 21, 22, 23, 24]}
16.
17. # parcours
18. for (clé, valeur) in multi.items():
19.     for i2 in range(len(valeur)):
20.         print(f"multi[{clé}][{i2}]={multi[clé][i2]}")
21.
22. # affichage en une ligne
23. print(f"multi={multi}")
```

Commentaires

- ligne 7 : multi**`[i1]`** est une liste ;
- ligne 18 : valeur est une liste ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/bases/bases_09.py
```

```
multi[0][0]=0
multi[0][1]=1
multi[0][2]=2
multi[1][0]=10
multi[1][1]=11
multi[1][2]=12
multi[1][3]=13
multi[2][0]=20
multi[2][1]=21
multi[2][2]=22
multi[2][3]=23
multi[2][4]=24
multi=[[0, 1, 2], [10, 11, 12, 13], [20, 21, 22, 23, 24]]
multi[zéro][0]=0
multi[zéro][1]=1
multi[un][0]=10
multi[un][1]=11
multi[un][2]=12
multi[un][3]=13
multi[deux][0]=20
multi[deux][1]=21
multi[deux][2]=22
multi[deux][3]=23
multi[deux][4]=24
multi={'zéro': [0, 1], 'un': [10, 11, 12, 13], 'deux': [20, 21, 22, 23, 24]}

Process finished with exit code 0
```

3.10 Script `[bases_10]` : liens entre chaînes et listes

Le script **`[bases_10]`** montrent comment récupérer dans une liste les éléments d'une chaîne séparés par un même séparateur.

```
1.  # chaîne vers liste
2.  chaine = '1:2:3:4'
3.  liste = chaine.split(':')
4.  print(type(liste))
5.
6.  # affichage liste
7.  print(f"liste a {len(liste)} éléments")
8.  print(f"liste={liste}")
9.
10. # liste vers chaîne
11. chaine2 = ":".join(liste)
12. print(f"chaine2={chaine2}")
13.
14. # ajoutons un champ vide
15. chaine += ":"
16. print(f"chaine={chaine}")
17. liste = chaine.split(":")
18.
19. # affichage liste
20. print(f"liste a {len(liste)} éléments")
21. print(f"liste={liste}")
22.
23. # ajoutons de nouveau un champ vide
24. chaine += ":"
25. print(f"chaine={chaine}")
26. liste = chaine.split(":")
27.
28. # affichage liste
29. print(f"liste a {len(liste)} éléments")
30. print(f"liste={liste}")
```

Notes :

- ligne 3 : la méthode *chaine.split(séparateur)* découpe la chaîne de caractères *chaine* en éléments séparés par *séparateur* et les rend sous forme de liste. Ainsi l'expression *'1:2:3:4'.split(":")* a pour valeur la liste *('1','2','3','4') ;*
- ligne 11 : 'separateur'.join(liste) a pour valeur la chaîne de caractères 'liste**`[0]`**+separateur+liste**`[1]`**+separateur+…'.

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/bases/bases_10.py
2.  <class 'list'>
3.  liste a 4 éléments
```

```
liste=['1', '2', '3', '4']
chaine2=1:2:3:4
chaine=1:2:3:4:
liste a 5 éléments
liste=['1', '2', '3', '4', '']
chaine=1:2:3:4::
liste a 6 éléments
liste=['1', '2', '3', '4', '', '']

Process finished with exit code 0
```

3.11 Script [bases_11] : les expressions régulières

Le script **[bases_11]** montre comment utiliser des expressions régulières :

```
1.  # on importe le module des expressions régulières
2.  import re
3.
4.
5.  # ------------------------------------------------------------------------
6.  def compare(modèle, chaine):
7.      # compare la chaîne [chaîne] au modèle [modèle]
8.      # affichage résultats
9.      print(f"\nRésultats({chaine},{modèle})")
10.     match = re.match(modèle, chaine)
11.     if match:
12.         print(match.groups())
13.     else:
14.         print(f"La chaîne [{chaine}] ne correspond pas au modèle [{modèle}]")
15.
16.
17. # expression régulières en python
18. # récupérer les différents champs d'une chaîne
19. # le modèle : une suite de chiffres entourée de caractères quelconques
20. # on ne veut récupérer que la suite de chiffres
21. modèle = r"^.*?(\d+).*?$"
22.
23. # on confronte la chaîne au modèle
24. compare(modèle, "xyz1234abcd")
25. compare(modèle, "12 34")
26. compare(modèle, "abcd")
27.
28. # le modèle : une suite de chiffres entourée de caractères quelconques
29. # on veut la suite de chiffres ainsi que les champs qui suivent et précèdent
30. modèle = r"^(.*?)(\d+)(.*?)$"
31.
32. # on confronte la chaîne au modèle
33. compare(modèle, "xyz1234abcd")
34. compare(modèle, "12 34")
35. compare(modèle, "abcd")
36.
37. # le modèle - une date au format jj/mm/aa
38. modèle = r"^\s*(\d\d)\/(\d\d)\/(\d\d)\s*$"
39. compare(modèle, "10/05/97")
40. compare(modèle, " 04/04/01 ")
41. compare(modèle, "5/1/01")
42.
43. # le modèle - un nombre décimal
44. modèle = r"^\s*([+-]?)\s*(\d+\.\d*|\.\d+|\d+)\s*$"
45. compare(modèle, "187.8")
46. compare(modèle, "-0.6")
47. compare(modèle, "4")
48. compare(modèle, ".6")
49. compare(modèle, "4.")
50. compare(modèle, " + 4")
51. # fin
```

Notes :

- noter le module **[re]** importé en ligne 2. C'est lui qui contient les fonctions de gestion des expressions régulières ;
- ligne 10 : la comparaison d'une chaîne à une expression régulière (modèle) rend le booléen *True* si la chaîne correspond au modèle, *False* sinon ;
- ligne 12 : *match.groups()* est un tuple dont les éléments sont les parties de la chaîne qui correspondent aux éléments de l'expression régulière entourés de parenthèses. Dans le modèle :

- `^.*?(\d+).*?`, *match.groups*() sera un tuple d'un élément parce qu'il y a une parenthèse ;
- `^(.*?)(\d+)(.*?)$`, *match.groups*() sera un tuple de 3 éléments parce qu'il y a trois parenthèses ;
- ligne 21 : une expression régulière littérale est notée r"xxx". C'est le symbole r qui fait de la chaîne une expression régulière ;

Les expressions régulières nous permettent de tester le format d'une chaîne de caractères. Ainsi on peut vérifier qu'une chaîne représentant une date est au format jj/mm/aa. On utilise pour cela un modèle et on compare la chaîne à ce modèle. Ainsi dans cet exemple, j m et a doivent être des chiffres. Le modèle d'un format de date valide est alors **"\d\d/\d\d/\d\d"** où le symbole **\d** désigne un chiffre. Les symboles utilisables dans un modèle sont les suivants :

Caractère	Description
\	Marque le caractère suivant comme caractère spécial ou littéral. Par exemple, "n" correspond au caractère "n" alors que "\n" correspond à un caractère de nouvelle ligne. La séquence "\\" correspond à "\", tandis que "\(" correspond à "(".
^	Correspond au début de la chaîne.
$	Correspond à la fin de la chaîne.
*	Correspond au caractère précédent, zéro fois ou plusieurs fois. Ainsi, "zo*" correspond à "z" ou à "zoo".
+	Correspond au caractère précédent, une ou plusieurs fois. Ainsi, "zo+" correspond à "zoo", mais pas à "z".
?	Correspond au caractère précédent, zéro ou une fois. Par exemple, "a?ve?" correspond à "ve" dans "lever".
.	Correspond à tout caractère unique, sauf le caractère de nouvelle ligne.
(modèle**)**	Recherche le *modèle* et mémorise la correspondance. La sous-chaîne correspondante peut être extraite de la collection **match.groups().** Pour trouver des correspondances avec des caractères entre parenthèses ( ), utilisez "\(" ou "\)".
x|y	Correspond soit à *x* soit à *y*. Par exemple, "z|foot" correspond à "z" ou à "foot". "(z|f)oo" correspond à "zoo" ou à "foo".
{*n*}	*n* est un nombre entier non négatif. Correspond exactement à *n* fois le caractère. Par exemple, "o{2}" ne correspond pas à "o" dans "Bob," mais aux deux premiers "o" dans "fooooot".
{*n*,}	*n* est un entier non négatif. Correspond à au moins *n* fois le caractère. Par exemple, "o{2,}" ne correspond pas à "o" dans "Bob", mais à tous les "o" dans "fooooot". "o{1,}" équivaut à "o+" et "o{0,}" équivaut à "o*".
{*n*,*m*}	*m* et *n* sont des entiers non négatifs. Correspond à au moins *n* et à au plus *m* fois le caractère. Par exemple, "o{1,3}" correspond aux trois premiers "o" dans "fooooooot" et "o{0,1}" équivaut à "o?".
[*xyz*]	Jeu de caractères. Correspond à l'un des caractères indiqués. Par exemple, "**[abc]**" correspond à "a" dans "plat".
[^*xyz*]	Jeu de caractères négatif. Correspond à tout caractère non indiqué. Par exemple, "**[^abc]**" correspond à "p" dans "plat".
[*a-z*]	Plage de caractères. Correspond à tout caractère dans la série spécifiée. Par exemple, "**[a-z]**" correspond à tout caractère alphabétique minuscule compris entre "a" et "z".
[^*m-z*]	Plage de caractères négative. Correspond à tout caractère ne se trouvant pas dans la série spécifiée. Par exemple, "**[^m-z]**" correspond à tout caractère ne se trouvant pas entre "m" et "z".
\b	Correspond à une limite représentant un mot, autrement dit, à la position entre un mot et un espace. Par exemple, "er\b" correspond à "er" dans "lever", mais pas à "er" dans "verbe".
\B	Correspond à une limite ne représentant pas un mot. "en*t\B" correspond à "ent" dans "bien entendu".
\d	Correspond à un caractère représentant un chiffre. Équivaut à **[0-9]**.
\D	Correspond à un caractère ne représentant pas un chiffre. Équivaut à **[^0-9]**.
\f	Correspond à un caractère de saut de page.
\n	Correspond à un caractère de nouvelle ligne.

\r	Correspond à un caractère de retour chariot.
\s	Correspond à tout espace blanc, y compris l'espace, la tabulation, le saut de page, etc. Équivaut à **"[\f\n\r\t\v]"**.
\S	Correspond à tout caractère d'espace non blanc. Équivaut à **"[^ \f\n\r\t\v]"**.
\t	Correspond à un caractère de tabulation.
\v	Correspond à un caractère de tabulation verticale.
\w	Correspond à tout caractère représentant un mot et incluant un trait de soulignement. Équivaut à **"[A-Za-z0-9_]"**.
\W	Correspond à tout caractère ne représentant pas un mot. Équivaut à **"[^A-Za-z0-9_]"**.
****num	Correspond à *num*, où *num* est un entier positif. Fait référence aux correspondances mémorisées. Par exemple, "(.)\1" correspond à deux caractères identiques consécutifs.
*****n*	Correspond à *n*, où *n* est une valeur d'échappement octale. Les valeurs d'échappement octales doivent comprendre 1, 2 ou 3 chiffres. Par exemple, "\11" et "\011" correspondent tous les deux à un caractère de tabulation. "\0011" équivaut à "\001" & "1". Les valeurs d'échappement octales ne doivent pas excéder 256. Si c'était le cas, seuls les deux premiers chiffres seraient pris en compte dans l'expression. Permet d'utiliser les codes ASCII dans des expressions régulières.
\x*n*	Correspond à *n*, où *n* est une valeur d'échappement hexadécimale. Les valeurs d'échappement hexadécimales doivent comprendre deux chiffres obligatoirement. Par exemple, "\x41" correspond à "A". "\x041" équivaut à "\x04" & "1". Permet d'utiliser les codes ASCII dans des expressions régulières.

Un élément dans un modèle peut être présent en 1 ou plusieurs exemplaires. Considérons quelques exemples autour du symbole **\d** qui représente 1 chiffre :

modèle	**signification**
\d	un chiffre
\d?	0 ou 1 chiffre
\d*	0 ou davantage de chiffres
\d+	1 ou davantage de chiffres
\d{2}	2 chiffres
\d{3,}	au moins 3 chiffres
\d{5,7}	entre 5 et 7 chiffres

Imaginons maintenant le modèle capable de décrire le format attendu pour une chaîne de caractères :

chaîne recherchée	**modèle**
une date au format jj/mm/aa	\d{2}/\d{2}/\d{2}
une heure au format hh:mm:ss	\d{2}:\d{2}:\d{2}
un nombre entier non signé	\d+
un suite d'espaces éventuellement vide	\s*
un nombre entier non signé qui peut être précédé ou suivi d'espaces	\s*\d+\s*
un nombre entier qui peut être signé et précédé ou suivi d'espaces	\s***[+|-]**?\s*\d+\s*
un nombre réel non signé qui peut être précédé ou suivi d'espaces	\s*\d+(.\d*)?\s*
un nombre réel qui peut être signé et précédé ou suivi d'espaces	\s***[+-]**?\s*\d+(.\d*)?\s*
une chaîne contenant le mot juste	\bjuste\b

On peut préciser où on recherche le modèle dans la chaîne :

modèle	**signification**

^modèle	le modèle commence la chaîne
modèle$	le modèle finit la chaîne
^modèle$	le modèle commence et finit la chaîne
modèle	le modèle est cherché partout dans la chaîne en commençant par le début de celle-ci.

chaîne recherchée	**modèle**
une chaîne se terminant par un point d'exclamation	!$
une chaîne se terminant par un point	\.$
une chaîne commençant par la séquence //	^//
une chaîne ne comportant qu'un mot éventuellement suivi ou précédé d'espaces	^\s*\w+\s*$
une chaîne ne comportant deux mot éventuellement suivis ou précédés d'espaces	^\s*\w+\s*\w+\s*$
une chaîne contenant le mot secret	\bsecret\b

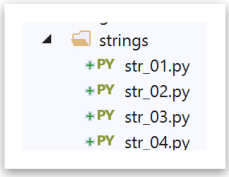
Les sous-ensembles d'un modèle peuvent être "récupérés". Ainsi non seulement, on peut vérifier qu'une chaîne correspond à un modèle particulier mais on peut récupérer dans cette chaîne les éléments correspondant aux sous-ensembles du modèle qui ont été entourés de parenthèses. Ainsi si on analyse une chaîne contenant une date jj/mm/aa et si on veut de plus récupérer les éléments jj, mm, aa de cette date on utilisera le modèle (\d\d)/(\d\d)/(\d\d).

Résultats du script

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/bases/bases_11.py
2.
3.  Résultats(xyz1234abcd,^.*?(\d+).*?$)
4.  ('1234',)
5.
6.  Résultats(12 34,^.*?(\d+).*?$)
7.  ('12',)
8.
9.  Résultats(abcd,^.*?(\d+).*?$)
10. La chaîne [abcd] ne correspond pas au modèle [^.*?(\d+).*?$]
11.
12. Résultats(xyz1234abcd,^(.*?)(\d+)(.*?)$)
13. ('xyz', '1234', 'abcd')
14.
15. Résultats(12 34,^(.*?)(\d+)(.*?)$)
16. ('', '12', ' 34')
17.
18. Résultats(abcd,^(.*?)(\d+)(.*?)$)
19. La chaîne [abcd] ne correspond pas au modèle [^(.*?)(\d+)(.*?)$]
20.
21. Résultats(10/05/97,^\s*(\d\d)\/(\d\d)\/(\d\d)\s*$)
22. ('10', '05', '97')
23.
24. Résultats( 04/04/01 ,^\s*(\d\d)\/(\d\d)\/(\d\d)\s*$)
25. ('04', '04', '01')
26.
27. Résultats(5/1/01,^\s*(\d\d)\/(\d\d)\/(\d\d)\s*$)
28. La chaîne [5/1/01] ne correspond pas au modèle [^\s*(\d\d)\/(\d\d)\/(\d\d)\s*$]
29.
30. Résultats(187.8,^\s*([+-]?)\s*(\d+\.\d*|\.\d+|\d+)\s*$)
31. ('', '187.8')
32.
33. Résultats(-0.6,^\s*([+-]?)\s*(\d+\.\d*|\.\d+|\d+)\s*$)
34. ('-', '0.6')
35.
36. Résultats(4,^\s*([+-]?)\s*(\d+\.\d*|\.\d+|\d+)\s*$)
37. ('', '4')
38.
39. Résultats(.6,^\s*([+-]?)\s*(\d+\.\d*|\.\d+|\d+)\s*$)
40. ('', '.6')
41.
42. Résultats(4.,^\s*([+-]?)\s*(\d+\.\d*|\.\d+|\d+)\s*$)
43. ('', '4.')
44.
45. Résultats( + 4,^\s*([+-]?)\s*(\d+\.\d*|\.\d+|\d+)\s*$)
46. ('+', '4')
```

```
47.
48. Process finished with exit code 0
```

4 Les chaînes de caractères



4.1 Script [str_01] : notation des chaînes de caractères

Le script **[str_01]** est le suivant :

```
1.  # chaînes de caractères
2.  # trois notations possibles
3.  chaine1 = "un"
4.  chaine2 = 'deux'
5.  chaine3 = """hélène va au
6.   marché acheter des légumes"""
7.  # affichage
8.  print(f"chaine1=[{chaine1}], chaine2=[{chaine2}], chaine3=[{chaine3}]")
```

Commentaires

- ligne 3 : une chaîne délimitée par des guillemets " ;
- ligne 4 : une chaîne délimitée par des apostrophes ' ;
- ligne 5 : une chaîne délimitée par des triples guillemets """. Dans ce cas, la chaîne peut s'étaler sur plusieurs lignes ;

Les résultats sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/strings/str_01.py
2.  chaine1=[un], chaine2=[deux], chaine3=[hélène va au
3.   marché acheter des légumes]
4.
5.  Process finished with exit code 0
```

4.2 Script [str_02] : les méthodes de la classe <str>

Le script **[str_02]** présente quelques-unes des méthodes de la classe <str> qui est la classe des chaînes de caractères :

```
1.  # fonctions sur chaînes de caractères
2.  # chaîne en minuscules
3.  print(f"'ABCD'.lower()={'ABCD'.lower()}")
4.  # chaîne en majuscules
5.  print(f"'abcd'.upper()={'abcd'.upper()}")
6.  # caractère n° 2
7.  print(f"'cheval[2]={'cheval'[2]}")
8.  # sous-chaîne avec les caractères 5 et 6
9.  print(f"'caractères accentués'[5:7]={'caractères accentués'[5:7]}")
10. # sous-chaîne à partir du caractère 4 inclus
11. print(f"'caractères accentués'[4:]={'caractères accentués'[4:]}")
12. # sous-chaîne jusqu'au caractère 6 exclus
13. print(f"'caractères accentués'[:5]={'caractères accentués'[:5]}")
14. # longueur de la chaîne
15. print(f"len('123')={len('123')}")
16. # élimination des blancs qui précèdent et suivent la chaîne
17. print(f"'  abcd '.strip()=[{'  abcd '.strip()}]")
18. # élimination des blancs qui suivent la chaîne
19. print(f"'  abcd '.rstrip()=[{'  abcd '.rstrip()}]")
20. # élimination des blancs qui précèdent la chaîne
21. print(f"'  abcd '.lstrip()=[{'  abcd '.lstrip()}]")
22. # le terme blanc recouvre en fait différents caractères
23. str = '  \r\nabcd \t\f'
24. print(f"str.strip()=[{str.strip()}]")
25. # remplacement d'une sous-chaîne par une autre
```

```
print(f"'abcd'.replace('a','x')={'abcd'.replace('a', 'x')}")
print(f"'abcd'.replace('ab','xy')={'abcd'.replace('ab', 'xy')}")
# recherche d'une sous-chaîne : rend la position ou -1 si sous-chaîne pas trouvée
print(f"'abcd'.find('bc')={'abcd'.find('bc')}")
print(f"'abcd'.find('bc')={'abcd'.find('Bc')}")
# début d'une chaîne
print(f"'abcd'.startswith('ab')={'abcd'.startswith('ab')}")
print(f"'abcd'.startswith('x')={'abcd'.startswith('x')}")
# fin d'une chaîne
print(f"'abcd'.endswith('cd')={'abcd'.endswith('cd')}")
print(f"'abcd'.endswith('x')={'abcd'.endswith('x')}")
# passage d'une liste de chaînes à une chaîne
print(f"'[X]'.join(['abcd', '123', 'èéà'])={'[X]'.join(['abcd', '123', 'èéà'])}")
print(f"''.join(['abcd', '123', 'èéà'])={''.join(['abcd', '123', 'èéà'])}")
# passage d'une chaîne à une liste de chaînes
print(f"'abcd 123 cdXY'.split('cd')={'abcd 123 cdXY'.split('cd')}")
# récupérer les mots d'une chaîne
print(f"'abcd 123 cdXY'.split(None)={'abcd 123 cdXY'.split(None)}")
```

Les commentaires alliés aux résultats obtenus suffisent pour la compréhension du script. Les résultats sont les suivants :

```
C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/strings/str_02.py
'ABCD'.lower()=abcd
'abcd'.upper()=ABCD
'cheval[2]=e
'caractères accentués'[5:7]=tè
'caractères accentués'[4:]=ctères accentués
'caractères accentués'[:5]=carac
len('123')=3
'  abcd '.strip()=[abcd]
'  abcd '.rstrip()=[  abcd]
'  abcd '.lstrip()=[abcd ]
str.strip()=[abcd]
'abcd'.replace('a','x')=xbcd
'abcd'.replace('ab','xy')=xycd
'abcd'.find('bc')=1
'abcd'.find('bc')=-1
'abcd'.startswith('ab')=True
'abcd'.startswith('x')=False
'abcd'.endswith('cd')=True
'abcd'.endswith('x')=False
'[X]'.join(['abcd', '123', 'èéà'])=abcd[X]123[X]èéà
''.join(['abcd', '123', 'èéà'])=abcd123èéà
'abcd 123 cdXY'.split('cd')=['ab', ' 123 ', 'XY']
'abcd 123 cdXY'.split(None)=['abcd', '123', 'cdXY']

Process finished with exit code 0
```

4.3 Script [str_03] : codage des chaînes de caractères (1)

Le script **[str_03]** présente des notions sur le codage des chaînes de caractères :

```
# codage des caractères

# chaîne de type str
str = "hélène va au marché acheter des légumes"
print(f"str=[{str}, type={type(str)}]")
# encodage utf-8
print("--- utf-8")
bytes1 = str.encode('utf-8')
print(f"bytes1={bytes1}, type={type(bytes1)}")
bytes2 = bytes(str, 'utf-8')
print(f"bytes2={bytes2}, type={type(bytes2)}")
# encodage iso-8859-1
print("--- iso-8859-1")
bytes1 = str.encode('iso-8859-1')
print(f"bytes1={bytes1}, type={type(bytes1)}")
bytes2 = bytes(str, 'iso-8859-1')
print(f"bytes2={bytes2}, type={type(bytes2)}")
# encodage latin1=iso-8859-1
print("--- latin1")
bytes1 = str.encode('latin1')
print(f"bytes1={bytes1}, type={type(bytes1)}")
bytes2 = bytes(str, 'latin1')
print(f"bytes2={bytes2}, type={type(bytes2)}")
```

L'encodage d'une chaîne de caractère de type <str> produit une chaîne binaire où chaque caractère de la chaîne a été représenté par un ou plusieurs octets. Il existe différents types d'encodage. Le script ci-dessus présente les deux plus courants en occident "utf-8" et "iso-8859-1" dit aussi "latin1.

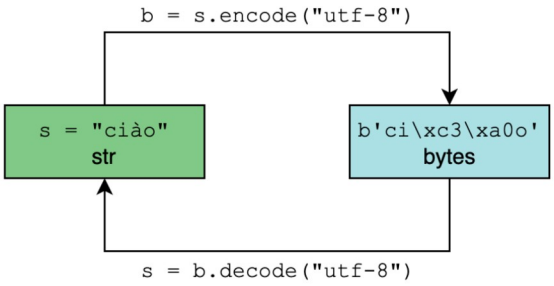
Le principe de l'encodage / décodage est illustré ci-dessous (ref. |https://realpython.com/python-encodings-guide/ |) :

Encoding and Decoding in Python 3

Python 3's `str` type is meant to represent human-readable text and can contain any Unicode character.

The `bytes` type, conversely, represents binary data, or sequences of raw bytes, that do not intrinsically have an encoding attached to it.

Encoding and decoding is the process of going from one to the other:



Encoding vs decoding (Image: Real Python)

In `.encode()` and `.decode()`, the `encoding` parameter is `"utf-8"` by default, though it's generally safer and more unambiguous to specify it:

Commentaires

- lignes 4-5 : la chaîne de caractères initiale qui va être encodée. Les instances du type <str> sont des chaînes unicode |https://docs.python.org/3/howto/unicode.html|, |https://realpython.com/python-encodings-guide/ | ;
- lignes 6-11 : deux façons pour encoder en UTF68 une chaîne de caractères :
- ligne 8 : str.encode('utf-8) ;
- ligne 10 : bytes(str, 'utf-8');
- lignes 12-17 : on refait la même chose avec l'encodage 'iso-8859-1' ;
- lignes 18-23 : 'latin1' est un autre nom de l'encodage 'iso-8859-1' ;

Les résultats sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/strings/str_03.py
2.  str=[hélène va au marché acheter des légumes, type=<class 'str'>
3.  --- utf-8
4.  bytes1=b'h\xc3\xa9l\xc3\xa8ne va au march\xc3\xa9 acheter des l\xc3\xa9gumes', type=<class 'bytes'>
5.  bytes2=b'h\xc3\xa9l\xc3\xa8ne va au march\xc3\xa9 acheter des l\xc3\xa9gumes', type=<class 'bytes'>
6.  --- iso-8859-1
7.  bytes1=b'h\xe9l\xe8ne va au march\xe9 acheter des l\xe9gumes', type=<class 'bytes'>
8.  bytes2=b'h\xe9l\xe8ne va au march\xe9 acheter des l\xe9gumes', type=<class 'bytes'>
9.  --- latin1
10. bytes1=b'h\xe9l\xe8ne va au march\xe9 acheter des l\xe9gumes', type=<class 'bytes'>
11. bytes2=b'h\xe9l\xe8ne va au march\xe9 acheter des l\xe9gumes', type=<class 'bytes'>
12.
13. Process finished with exit code 0
```

Commentaires

- **ligne 4 : on voit que les caractères accentués ont été codés sur deux octets :**
- **é : [\xc3\xa9]** qui est la suite binaire 11000011 10101001 ;
- è : **[\xc3\xa8]** qui est la suite binaire 11000011 10101000 ;
- lignes 7 : avec le codage iso-8859-1, ces deux caractères accentués sont codés différemment :
- é : **[\xe9]** qui est la suite binaire 11101001 ;
- è : **[\xe8]** qui est la suite binaire 11101000 ;

4.4 Script [str_04] : encodage des chaînes de caractères (2)

Le script **[str_04]** présente deux autres types d'encodage : 'base64' et 'quoted-printable'. Ces deux encodages n'encodent pas des chaînes de caractères Unicode mais des objets binaires. Par exemple, lorsqu'on attache un document Word à un mail, celui-ci va subir l'un des deux encodages selon le gestionnaire de courrier utilisé. Ce sera le cas pour la plupart des fichiers attachés.

Le script est le suivant :

```
1.  # encodage / décodage
2.  import codecs
3.
4.  # chaîne
5.  print("---- chaîne unicode")
6.  str1 = "hélène va au marché acheter des légumes"
7.  print(f"str1=[{str1}], type(str1)={type(str1)}")
8.
9.  # encodage utf-8
10. print("---- chaîne unicode -> binaire utf-8")
11. bytes1 = bytes(str1, "utf-8")
12. print(f"bytes1=[{bytes1}], type(bytes1)={type(bytes1)}")
13.
14. # décodage utf-8
15. print("---- binaire utf-8 -> chaîne unicode")
16. str2 = bytes1.decode("utf-8")
17. print(f"str2=[{str2}], type(str2)={type(str2)}")
18. print(f"str2==str1={str2 == str1}")
19.
20. # encodage iso-8859-1
21. print("---- chaîne unicode -> binaire iso-8859-1")
22. bytes2 = bytes(str1, "iso-8859-1")
23. print(f"bytes2=[{bytes2}], type(bytes2)={type(bytes2)}")
24.
25. # décodage iso-8859-1
26. print("---- binaire iso-8859-1 -> chaîne unicode")
27. str3 = bytes2.decode("iso-8859-1")
28. print(f"str3=[{str3}], type(str3)={type(str3)}")
29. print(f"str3==str1={str3 == str1}")
30.
31. # erreur de décodage - bytes1 est en utf-8 - on le décode en iso-8859-1
32. print("--- binaire utf-8 (décodage iso-8859-1) --> chaîne unicode")
33. str4 = bytes1.decode("iso-8859-1")
34. print(f"str4=[{str4}], type(str4)={type(str4)}")
35.
36. # encodage utf-8 d'une chaîne Unicode
37. print("---- chaîne unicode -> binaire utf-8")
38. bytes3 = codecs.encode(str1, "utf-8")
39. print(f"bytes3=[{bytes3}], type(bytes3)={type(bytes3)}")
40.
41. # encodage d'une chaîne binaire UTF-8 en base64
42. print("---- binaire utf-8 -> binaire base64")
43. bytes4 = codecs.encode(bytes1, "base64")
44. print(f"bytes4=[{bytes4}], type(bytes4)={type(bytes4)}")
45.
46. # retour à la chaîne unicode d'origine
47. print("---- binaire base64 -> binaire utf-8 -> chaîne unicode")
48. str6 = codecs.decode(bytes4, "base64").decode("utf-8")
49. print(f"str6=[{str6}], type(str6)={type(str6)}")
50.
51. # encodage d'une chaîne binaire en quoted-printable
52. print("---- binaire utf-8 -> binaire quoted-printable")
53. str7 = codecs.encode(bytes1, "quoted-printable")
54. print(f"str7=[{str7}], type(str7)={type(str7)}")
55.
56. # retour à la chaîne unicode d'origine
57. print("---- binaire quoted-printable -> binaire utf-8 -> chaîne unicode")
58. str8 = codecs.decode(str7, "quoted-printable").decode("utf-8")
59. print(f"str8=[{str8}], type(str8)={type(str8)}")
```

Commentaires

- ligne 2 : le module **[codecs]** permet de faire les codages 'base64' et 'quoted-printable'. Il peut en faire beuacoup d'autres ;
- lignes 4-7 : la chaîne Unicode qui va subir divers encodages ;
- lignes 9-12 : encodage utf-8. on obtient un binaire ;
- lignes 14-18 : décodage utf-8 pour revenir à la chaîne Unicode d'origine ;

- lignes 20-29 : on répète le même processus avec l'encodage 'iso-8859-1' ;
- lignes 31-34 : on montre une erreur de décodage :
- ligne 33 : bytes1 est une chaîne binaire encodée en 'utf-8'. On la décode en 'iso-8859-1' ;
- lignes 36-39 : autre façon d'encoder une chaîne de caractères en utf-8 avec le module **[codecs]** ;
- lignes 41-44 : une chaîne binaire 'utf-8' est encodée en 'base64' ;
- lignes 46-49 : montrent comment passer de la chaîne binaire 'base64' à la chaîne unicode d'origine ;
- lignes 51-59 : on répète ce processus avec un encodage 'quoted-printable' au lieu de 'base64' ;

Les résultats sont les suivants :

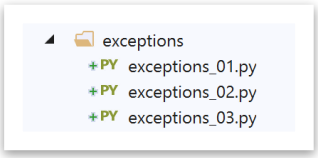
```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/strings/str_04.py
2.  ---- chaîne unicode
3.  str1=[hélène va au marché acheter des légumes], type(str1)=<class 'str'>
4.  ---- chaîne unicode -> binaire utf-8
5.  bytes1=[b'h\xc3\xa9l\xc3\xa8ne va au march\xc3\xa9 acheter des l\xc3\xa9gumes'], type(bytes1)=<class
    'bytes'>
6.  ---- binaire utf-8 -> chaîne unicode
7.  str2=[hélène va au marché acheter des légumes], type(str2)=<class 'str'>
8.  str2==str1=True
9.  ---- chaîne unicode -> binaire iso-8859-1
10. bytes2=[b'h\xe9l\xe8ne va au march\xe9 acheter des l\xe9gumes'], type(bytes2)=<class 'bytes'>
11. ---- binaire iso-8859-1 -> chaîne unicode
12. str3=[hélène va au marché acheter des légumes], type(str3)=<class 'str'>
13. str3==str1=True
14. --- binaire utf-8 (décodage iso-8859-1) --> chaîne unicode
15. str4=[hÃ©lÃ¨ne va au marchÃ© acheter des lÃ©gumes], type(str4)=<class 'str'>
16. ---- chaîne unicode -> binaire utf-8
17. bytes3=[b'h\xc3\xa9l\xc3\xa8ne va au march\xc3\xa9 acheter des l\xc3\xa9gumes'], type(bytes3)=<class
    'bytes'>
18. ---- binaire utf-8 -> binaire base64
19. bytes4=[b'aMOpbMOobmUgdmEgYXUgbWFyY2jDqSBhY2hldGVyIGRlcyBsw6lndW1lcw==\n'], type(bytes4)=<class 'bytes'>
20. ---- binaire base64 -> binaire utf-8 -> chaîne unicode
21. str6=[hélène va au marché acheter des légumes], type(str6)=<class 'str'>
22. ---- binaire utf-8 -> binaire quoted-printable
23. str7=[b'h=C3=A9l=C3=A8ne=20va=20au=20march=C3=A9=20acheter=20des=20l=C3=A9gumes'], type(str7)=<class
    'bytes'>
24. ---- binaire quoted-printable -> binaire utf-8 -> chaîne unicode
25. str8=[hélène va au marché acheter des légumes], type(str8)=<class 'str'>
26.
27. Process finished with exit code 0
```

- lignes 14-15 : un binaire utf-8 est décodé en chaîne Unicode avec le mauvais décodeur 'iso-8859-1'. Aussi certains caractères Unicode générés sont incorrects, ici les caractères accentués ;
- lignes 18-19 : le codage 'base64' consiste à utiliser 64 caractères ASCII (codés sur 7 bits) pour encoder un binaire quleconque. Cela augmente, comme on le voit, la taille du binaire de la chaîne ;
- lignes 22-23 : le codage 'quoted-printable' consiste lui aussi à utiliser des caractères ASCII (codés sur 7 bits) pour encoder un binaire quelconque ;

On se souviendra que lorsqu'on reçoit un binaire, du réseau internet par exemple, qui représente un texte, pour retrouver celui-ci il faut connaître les encodages qu'il a subis.

5 Les exceptions

Nous nous intéressons maintenant aux exceptions.



5.1 script [exceptions_01]

Le premier script illustre la nécessité de gérer les exceptions.

```
1.  # on provoque volontairement une erreur
2.  x = 4 / 0
```

On provoque volontairement une erreur pour voir ce qui se passe (*exceptions-01.py*) :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/exceptions/exceptions_01.py
1.  Traceback (most recent call last):
2.    File "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/exceptions/exceptions_01.py", line 2, in
    <module>
3.      x = 4 / 0
4.  ZeroDivisionError: division by zero
5.
6.  Process finished with exit code 1
```

La ligne 4 nous donne :

- le type de l'exception : ZeroDivisionError ;
- le message d'erreur associé : division by zero. Il est en anglais. C'est quelque chose qu'on peut vouloir changer.

Une règle essentielle est qu'on doit tout faire pour éviter les exceptions produites par l'interpréteur Python. Il nous faut gérer les erreurs nous-mêmes.

La syntaxe d'une gestion d'exceptions est la suivante :

```
1.  try:
2.          actions susceptibles de lancer une exception
3.  except (Ex1, Ex2…) as ex:
4.          actions de gestion de l'exception [ex]
5.  except (Ex11, Ex12…) as ex:
6.          actions de gestion de l'exception [ex]
7.  finally:
8.     actions toujours exécutées qu'il y ait exception ou non
```

Dans le **try**, l'exécution des actions s'arrête dès qu'une exception (erreur) survient. Dans ce cas, l'exécution se poursuit avec les actions de l'une des clauses **except** :

- ligne 3 : si l'exception **[ex]** qui s'est produite est d'un type appartenant au tuple (Ex1, Ex2…) ou dérivé de l'un d'eux, alors les actions de la ligne 4 sont exécutées ;
- ligne 5 : si l'exception n'a pas été interceptée par la ligne 3, et qu'une autre clause **[except]** existe, alors le même processus se déroule. Etc… ;
- il peut y avoir autant de clauses **[except]** que nécessaires pour gérer les différents types d'exception qui peuvent de produire dans le **[try]** ;
- si l'exception n'a été traitée par aucune des clauses **[except]** alors elle remontera au code appelant. Si celui-ci est lui-même dans une structure **try** / **except**, l'exception est de nouveau gérée sinon elle continue à remonter la chaîne des méthodes appelées. En dernier ressort, elle arrive à l'interpréteur Python. Celui-ci arrête alors le programme excécuté et affiche un message d'erreur du type montré dans l'exemple précédent. La règle est donc que le programme principal doit arrêter toutes les exceptions qui peuvent remonter des méthodes appelées ;
- ligne 7 : la clause **[finally]** est toujours exécutée qu'il y ait eu exception (suite du **except**) ou pas (suite du **try**). Ceci est vrai même si une exception a eu lieu et qu'elle n'a pas été interceptée. Dans ce cas, la clause **[finally]** sera exécutée avant que l'exception ne remonte au code appelant ;

Une exception transporte avec elle des informations sur l'erreur qui s'est produite. On peut les obtenir avec la syntaxe suivante :

```
except MyException as exception:
```

[exception] est l'exception qui s'est produite. **[exception.args]** représente le tuple des paramètres de l'exception.

Pour lancer une exception, on utilise la syntaxe

```
raise MyException(param1, param2…)
```

où le plus souvent *MyException* est une classe dérivée de la classe *BaseException*. Les paramètres passés au constructeur de la classe seront disponibles à la clause **except** des structures d'interception des exceptions avec la syntaxe **[ex.args]** si **[ex]** est l'exception interceptée par la clause **[except]**.

Ces concepts sont illustrés par le script suivant.

5.2 script [exceptions_02]

Le script suivant gère explicitement les erreurs :

```
# gestion des exceptions
essai = 0

# on provoque une erreur et on la gère
x = 2
try:
    x = 4 / 0
except ZeroDivisionError as erreur:
    # erreur est l'exception interceptée
    print(f"essai n° {essai} : {erreur}")
# la valeur de x n'a pas changé
print(f"x={x}")

# on recommence
essai += 1
try:
    x = 4 / 0
except BaseException as erreur:
    # on intercepte l'exception la plus générale
    # erreur est l'exception interceptée
    print(f"essai n° {essai} : {erreur}")

# on peut intercepter différents types d'exceptions
# l'exécution s'arrête sur le 1er [except] capable de traiter l'exception
essai += 1
try:
    x = 4 / 0
except ValueError as erreur:
    # cette exception ne se produit pas ici
    print(f"essai n° {essai} : {erreur}")
except BaseException as erreur:
    # on intercepte l'exception la plus générale
    print(f"essai n° {essai} : (Exception) {erreur}")
except ZeroDivisionError as erreur:
    # on intercepte un type précis
    print(f"essai n° {essai} : (ZeroDivisionError) {erreur}")

# on recommence en changeant l'ordre
essai += 1
try:
    x = 4 / 0
except ValueError as erreur:
    # cette exception ne se produit pas ici
    print(f"essai n° {essai} : {erreur}")
except ZeroDivisionError as erreur:
    # on intercepte un type précis
    print(f"essai n° {essai} : (ZeroDivisionError) {erreur}")
except BaseException as erreur:
    # on intercepte l'exception la plus générale
    print(f"essai n° {essai} : (Exception) {erreur}")

# une clause except sans arguments
```

```
53. essai += 1
54. try:
55.     x = 4 / 0
56. except:
57.     # on ne s'intéresse pas à la nature de l'exception
58.     print(f"essai n° {essai} : il y a eu un problème")
59.
60. # un autre type d'exception
61. essai += 1
62. try:
63.     # x ne peut être converti en nombre entier
64.     x = int("x")
65. except ValueError as erreur:
66.     # erreur est l'exception interceptée
67.     print(f"essai n° {essai} : {erreur}")
68.
69. # une exception transporte des informations dans un tuple accessible au programme
70. essai += 1
71. try:
72.     x = int("x")
73. except ValueError as erreur:
74.     # erreur est l'exception interceptée
75.     print(f"essai n° {essai} : {erreur}, paramètres={erreur.args}")
76.
77. # on peut lancer des exceptions
78. essai += 1
79. try:
80.     raise ValueError("param1", "param2", "param3")
81. except ValueError as erreur:
82.     # erreur est l'exception interceptée
83.     print(f"essai n° {essai} : {erreur}, paramètres={erreur.args}")
84.
85.
86. # on peut créer ses propres exceptions
87. # elles doivent dériver de la classe [BaseException]
88. class MyError(BaseException):
89.     pass
90.
91.
92. # on lance l'exception MyError
93. essai += 1
94. try:
95.     raise MyError("info1", "info2", "info3")
96. except MyError as erreur:
97.     # erreur est l'exception interceptée
98.     print(f"essai n° {essai} : {erreur}, paramètres={erreur.args}")
99.
100.# on lance l'exception MyError avec un msg d'erreur
101.essai += 1
102.try:
103.    raise MyError("mon msg d'erreur")
104.except MyError as erreur:
105.    # erreur est l'exception interceptée
106.    print(f"essai n° {essai} : {erreur.args[0]}")
107.
108.# la clause finally est toujours exécutée
109.# qu'il y ait exception ou non
110.essai += 1
111.x = None
112.try:
113.    x = 1
114.except:
115.    # exception
116.    print(f"essai n° {essai} : exception")
117.finally:
118.    # exécuté dans tous les cas
119.    print(f"essai n° {essai} : finally x={x}")
120.
121.essai += 1
122.x = None
123.try:
124.    x = 2 / 0
125.except:
126.    # exception
127.    print(f"essai n° {essai} : exception")
128.finally:
129.    # exécuté dans tous les cas
```

```
130.    print(f"essai n° {essai} : finally x={x}")
131.
132.# on n'est pas obligés de mettre une clause [except]
133.essai += 1
134.try:
135.    # on provoque une erreur
136.    x = 4 / 0
137.finally:
138.    # exécuté dans tous les cas
139.    print(f"essai n° {essai} : finally x={x}")
```

Notes :

- lignes 4-12 : on gère une division par zéro ;
- ligne 8 : on intercepte l'exception exacte qui se produit et on l'affiche ;
- ligne 12 : à cause de l'exception qui s'est produite, *x* n'a pas reçu de valeur ligne 7 et n'a donc pas changé de valeur ;
- lignes 14-21 : on refait la même chose mais en interceptant une exception de plus haut niveau de type *BaseException*. Comme l'exception *ZeroDivisionError* dérive de la classe *BaseException*, la clause *except* l'arrêtera ;
- lignes 23-36 : on met plusieurs clauses *except* pour gérer plusieurs types d'exception. Une seule clause *except* sera exécutée ou aucune si l'exception ne vérifie aucune clause *except ;*
- lignes 38-50 : on refait la même chose en changeant l'ordre des clauses **[except]** pour montrer le rôle de celui-ci ;
- lignes 52-58 : la clause *except* peut n'avoir aucun argument. Dans ce cas, elle arrête toutes les exceptions ;
- lignes 60-67 : introduisent l'exception *ValueError ;*
- lignes 69-75 : on récupère les informations transportées par l'exception ;
- lignes 77-83 : introduisent la façon de lancer (raise) une exception ;
- lignes 86-106 : illustrent l'utilisation d'une classe d'exception propriétaire *MyError*. La classe *MyError* se contente de dériver la classe de base *BaseException*. Elle n'ajoute rien à sa classe de base. Mais maintenant, elle peut être nommée explicitement dans les clauses *except ;*
- lignes 108-130 : illustrent l'utilisation de la clause *finally* ;
- lignes 132-139 : ces lignes montrent que la clause **[except]** n'est pas obligatoire ;

Les résultats écran sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/exceptions/exceptions_02.py
2.  Traceback (most recent call last):
3.    File "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/exceptions/exceptions_02.py", line 136, in
    <module>
4.      x = 4 / 0
5.  ZeroDivisionError: division by zero
6.  essai n° 0 : division by zero
7.  x=2
8.  essai n° 1 : division by zero
9.  essai n° 2 : (Exception) division by zero
10. essai n° 3 : (ZeroDivisionError) division by zero
11. essai n° 4 : il y a eu un problème
12. essai n° 5 : invalid literal for int() with base 10: 'x'
13. essai n° 6 : invalid literal for int() with base 10: 'x', paramètres=("invalid literal for int() with base
    10: 'x'",)
14. essai n° 7 : ('param1', 'param2', 'param3'), paramètres=('param1', 'param2', 'param3')
15. essai n° 8 : ('info1', 'info2', 'info3'), paramètres=('info1', 'info2', 'info3')
16. essai n° 9 : mon msg d'erreur
17. essai n° 10 : finally x=1
18. essai n° 11 : exception
19. essai n° 11 : finally x=None
20. essai n° 12 : finally x=None
21.
22. Process finished with exit code 1
```

5.3 script [exceptions_03]

Ce nouveau script illustre la remontée des exceptions dans la chaîne des fonctions appelantes :

```
1.  # une exception propriétaire
2.  class MyError(BaseException):
3.      pass
4.
5.
6.  # trois fonctions
7.  def f1(x: int) -> int:
8.      # on ne gère pas les exceptions - elles remontent automatiquement
9.      return f2(x)
```

```
10.
11.
12. def f2(y: int) -> int:
13.     # on ne gère pas les exceptions - elles remontent automatiquement
14.     return f3(y)
15.
16.
17. def f3(z: int) -> int:
18.     if (z % 2) == 0:
19.         # si z est pair, on lance une exception
20.         raise MyError("exception dans f3")
21.     else:
22.         return 2 * z
23.
24.
25. # ---------- main
26.
27. # les exceptions remontent la chaîne des méthodes appelées
28. # jusqu'à ce qu'une méthode l'intercepte. Ici ce sera main
29. try:
30.     print(f1(4))
31. except MyError as erreur:
32.     print(f"type : {type(erreur)}, arguments : {erreur.args}")
33.
34.
35. # trois autres fonctions qui enrichissent les exceptions qu'elle remonte
36. def f4(x: int) -> int:
37.     try:
38.         return f5(x)
39.     except MyError as erreur:
40.         # on enrichit l'exception puis on la relance
41.         raise MyError("exception dans f4", erreur)
42.
43.
44. def f5(y: int) -> int:
45.     try:
46.         return f6(y)
47.     except MyError as erreur:
48.         # on enrichit l'exception puis on la relance
49.         raise MyError("exception dans f5", erreur)
50.
51.
52. def f6(z: int) -> int:
53.     if (z % 2) == 0:
54.         # on lance une exception si z est pair
55.         raise MyError("exception dans f6")
56.     else:
57.         return 2 * z
58.
59.
60. # ---------- main
61. try:
62.     print(f4(4))
63. except MyError as erreur:
64.     # affichage de l'exception
65.     print(f"type : {type(erreur)}, arguments : {erreur.args}")
66.     # on peut remonter la pile des exceptions
67.     err = erreur
68.     # on affiche le msg d'erreur
69.     print(err.args[0])
70.     # une exception est-elle encapsulée ?
71.     while len(err.args) == 2 and isinstance(err.args[1], BaseException):
72.         # changement d'exception
73.         err = err.args[1]
74.         # le 1er argument est le msg d'erreur
75.         print(err.args[0])
```

Notes :

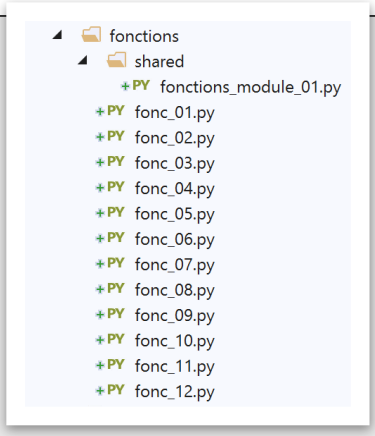
- lignes 25-32, dans l'appel *main --> f1 --> f2 --> f3* (ligne 30), l'exception *MyError* lancée par *f3* va remonter jusqu'à *main*. Elle sera alors traitée par la clause *except* de la ligne 31 ;
- lignes 61-75 : dans l'appel *main --> f4 --> f5 --> f6* (ligne 62), l'exception lancée *MyError* par *f6* va remonter jusqu'à *main*. Elle sera alors traitée par la clause *except* de la ligne 63. Cette fois-ci, dans sa remontée de la chaîne des fonctions appelantes, l'exception *MyError* qui remonte, est elle-même encapsulée dans une autre exception ;
- lignes 66-75 : montrent comment remonter la pile des exceptions ;

- ligne 71 : la fonction **[isinstance(instance, Classe)]** rend True si l'objet **[instance]** est de type **[Classe]** ou dérivé. Ici, nous avons utilisé l'exception de niveau le plus haut **[BaseException]**, ce qui nous assure de récupérer toutes les exceptions ;

Les résultats écran sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/exceptions/exceptions_03.py
2.  type : <class '__main__.MyError'>, arguments : ('exception dans f3',)
3.  type : <class '__main__.MyError'>, arguments : ('exception dans f4', MyError('exception dans f5',
    MyError('exception dans f6')))
4.  exception dans f4
5.  exception dans f5
6.  exception dans f6
7.
8.  Process finished with exit code 0
```

6 Les fonctions



6.1 Script `[fonc_01]` : portée des variables

Le script **`[fonc_01]`** montre des exemples de portée de variables entre fonctions :

```
1.  # portée des variables
2.  def f1():
3.      # l'utilisation de variables globales est à éviter
4.      # variable globale i
5.      global i
6.      i += 1
7.      # variable locale j
8.      j = 10
9.      print(f"f1[i,j]=[{i},{j}]")
10.
11.
12. def f2():
13.     # l'utilisation de variables globales est à éviter
14.     # variable globale i
15.     global i
16.     i += 1
17.     # variable locale j
18.     j = 20
19.     print(f"f2[i,j]=[{i},{j}]")
20.
21.
22. def f3():
23.     # variable locale i
24.     i = 1
25.     # variable locale j
26.     j = 30
27.     print(f"f3[i,j]=[{i},{j}]")
28.
29.
30. # programme principal
31. i = 0
32. j = 0
33. # ces deux variables ne seront connues d'une fonction f
34. # que si celle-ci déclare explicitement par l'instruction global qu'elle veut les utiliser
35. # ou que la fonction n'utilise la variable globale qu'en lecture
36. f1()
37. f2()
38. f3()
39. # j n'a pas changé mais i a changé
40. print(f"[i,j]=[{i},{j}]")
```

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_01.py
2.  f1[i,j]=[1,10]
3.  f2[i,j]=[2,20]
4.  f3[i,j]=[1,30]
5.  [i,j]=[2,0]
```

```
6.
7.  Process finished with exit code 0
```

Notes :

- le script montre l'utilisation de la variable i, déclarée globale dans les fonctions f1 et f2. Dans ce cas, le programme principal et les fonctions f1 et f2 partagent la même variable i.

6.2 Script `[fonc_02]` : portée des variables

Le script **`[fonc_03]`** reprend le script **`[fonc_02]`** et montre comment éviter l'usage de variables globales :

```
1.  # portée des variables
2.  def f1(i):
3.      # variable locale i
4.      i += 1
5.      # variable locale j
6.      j = 10
7.      print(f"f1[i,j]=[{i},{j}]")
8.      # on rend la valeur modifiée
9.      return i
10.
11.
12. def f2(i):
13.     # variable locale i
14.     i += 1
15.     # variable locale j
16.     j = 20
17.     print(f"f2[i,j]=[{i},{j}]")
18.     # on rend la valeur modifiée
19.     return i
20.
21.
22. def f3():
23.     # variable locale i
24.     i = 1
25.     # variable locale j
26.     j = 30
27.     print(f"f3[i,j]=[{i},{j}]")
28.
29.
30. # programme principal
31. i = 0
32. j = 0
33. # ces deux variables ne seront connues d'une fonction f
34. # que si celle-ci déclare explicitement par l'instruction global qu'elle veut les utiliser
35. # ou si la fonction n'utilise la variable globale qu'en lecture
36. i = f1(i)
37. i = f2(i)
38. f3()
39. # j n'a pas changé mais i a changé
40. print(f"[i,j]=[{i},{j}]")
```

Commentaires :

- lignes 2, 12 : au lieu d'être déclarée globale, la variable **`[i]`** est passée en paramètre aux fonctions f1 et f2 ;
- lignes 9, 19 : les fonctions f1 et f2 rendent la variable **`[i]`** modifiée au programme principal. Celui-ci la récupère aux lignes 36 et 37 ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_02.py
2.  f1[i,j]=[1,10]
3.  f2[i,j]=[2,20]
4.  f3[i,j]=[1,30]
5.  [i,j]=[2,0]
6.
7.  Process finished with exit code 0
```

6.3 Script [fonc_03] : portée des variables

Le script **[fonc_03]** montre une particularité des variables utilisées à la fois dans une fonction et dans le code appelant celle-ci, selon que dans la fonction cette variable est utilisée uniquement en **lecture** ou non.

```
1.  def f1():
2.      # ici la variable globale i est connue
3.      print(f"[f1] i={i}")
4.      # ici la variable globale j est connue
5.      print(f"[f1] j={j}")
6.
7.
8.  def f2():
9.      # ici la variable globale i n'est pas connue
10.     # car la fonction f2 définit une variable locale de même nom
11.     # c'est alors elle qui a priorité
12.     try:
13.         # essai d'affichage de la variable locale i définie plus loin
14.         print(f"[f2] i={i}")
15.         # l'instruction qui suit fait de i une variable locale de la fonction f2
16.         i = 7
17.     except BaseException as erreur:
18.         print(f"[f2] erreur={erreur}")
19.
20.
21. def f3():
22.     # ici la variable globale i n'est pas connue
23.     # car la fonction f3 définit une variable locale de même nom
24.     # c'est alors elle qui a priorité
25.
26.     # l'instruction qui suit fait de i une variable locale
27.     i = 7
28.     # affichage - ici i est connue
29.     print(f"[f3] i={i}")
30.
31.
32. # main -----------
33. # variables globales aux fonctions
34. i = 10
35. j = 20
36. # appel de f1
37. f1()
38. print(f"[main] i={i}, j={j}")
39. # appel de f2
40. f2()
41. print(f"[main] i={i}")
42. # appel de f3
43. f3()
44. print(f"[main] i={i}")
```

Notes

- ligne 34 : le code principal définit une variable **[i]** ;
- lignes 1-5 : la fonction f1 utilise également une variable **[i]** sans lui donner de valeur. C'est une **lecture** de la variable **[i]**. Dans ce cas, la variable **[i]** utilisée est celle du code appelant, ligne 34 ;
- lignes 8-18 : la fonction f2 utilise également une variable **[i]** mais lui donne une valeur ligne 16. Le fait de donner une valeur à la variable **[i]** dans f2 fait automatiquement de **[i]** une variable locale à la fonction **[f2]**. Celle-ci 'cache' donc la variable **[i]** du code appelant ;
- ligne 14 : l'opération d'écriture de la variable locale **[i]** va échouer car celle-ci n'a pas de valeur lorsqu'on arrive à la ligne 14. Elle obtient sa valeur ligne 16. Il va se produire une exception. Pour cette raison, on a inséré la ligne 14 dans un try / catch ;
- lignes 21-29 : la fonction f3 fait la même chose que la fonction f2 mais définit plus tôt sa variable locale **[i]** ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_03.py
2.  [f1] i=10
3.  [f1] j=20
4.  [main] i=10, j=20
5.  [f2] erreur=local variable 'i' referenced before assignment
6.  [main] i=10
7.  [f3] i=7
8.  [main] i=10
```

```
9.
10. Process finished with exit code 0
```

6.4 Script [fonc_04] : mode de passage des paramètres

Le script est le suivant :

```
1.  # fonction f1
2.  def f1(a):
3.      a = 2
4.
5.
6.  # fonction f2
7.  def f2(a, b):
8.      a = 2
9.      b = 3
10.     return a, b
11.
12.
13. # ------------------------ main
14. x = 1
15. f1(x)
16. print(f"x={x}")
17. (x, y) = (-1, -1)
18. (x, y) = f2(x, y)
19. print(f"x={x}, y={y}")
```

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_04.py
2.  x=1
3.  x=2, y=3
4.
5.  Process finished with exit code 0
```

Notes :

- tout est objet en Python. Certains objets sont dits "immutable" (en anglais) : on ne peut pas les modifier. C'est le cas des nombres, des chaînes de caractères, des tuples. Lorsque des objets Python sont passés en paramètres à des fonctions, ce sont leur références qui sont passées sauf si ces objets sont "immutable" auquel cas, c'est la valeur de l'objet qui est passée ;
- les fonctions *f1* (ligne 2), et *f2* (ligne 7) veulent illustrer le passage d'un paramètre de sortie. On veut que le paramètre effectif d'une fonction soit modifié par la fonction ;
- lignes 2-3 : la fonction *f1* modifie son paramètre formel a. On veut savoir si le paramètre effectif va lui aussi être modifié ;
- lignes 14-15 : le paramètre effectif est *x=1*. La ligne 2 des résultats, montre que le paramètre effectif n'est pas modifié. Ainsi le paramètre effectif **x** et le paramètre formel **a** sont **deux objets différents ;**
- lignes 8-10 : la fonction *f2* modifie ses paramètres formels **a** et **b**, et les rend comme résultats ;
- lignes 17-18 : on passe à *f2* les paramètres effectifs (x,y) et le résultat de *f2* est remis dans (x,y). La ligne 3 des résultats montre que les paramètres effectifs (x,y) ont été modifiés.

On en conclut que lorsqu'on des objets "immutable" sont des paramètres de sortie, il faut qu'ils fassent partie des résultats renvoyés par la fonction.

6.5 Script [fonc_05] : ordre d'écriture des fonctions dans un script

Le script **[fonc_05]** montre qu'on ne peut appeler une fonction si elle n'a pas été précédemment rencontrée dans le code :

```
1.  # ------------------------ main
2.  print(f2(100, 200))
3.
4.  # fonction f1
5.  def f1(a):
6.      return a + 10
7.
8.
9.  # fonction f2
10. def f2(a, b):
11.     return f1(a + b)
```

Notes

- la ligne 2 va provoquer une erreur parce qu'elle utilise la fonction f2 qui n'a pas encore été définie dans le script ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_05.py
2.  Traceback (most recent call last):
3.    File "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_05.py", line 2, in <module>
4.      print(f2(100, 200))
5.  NameError: name 'f2' is not defined
6.
7.  Process finished with exit code 1
```

6.6 Script `[fonc_06]` : ordre d'écriture des fonctions dans un script

Le script **`[fonc_06]`** montre que ce qui vaut pour le code appelant ne vaut pas pour les fonctions :

```
1.  # fonction f2
2.  def f2(a, b):
3.      return f1(a + b)
4.
5.
6.  # fonction f1
7.  def f1(a):
8.      return a + 10
9.
10.
11. # ------------------------ main
12. print(f2(100, 200))
```

Notes

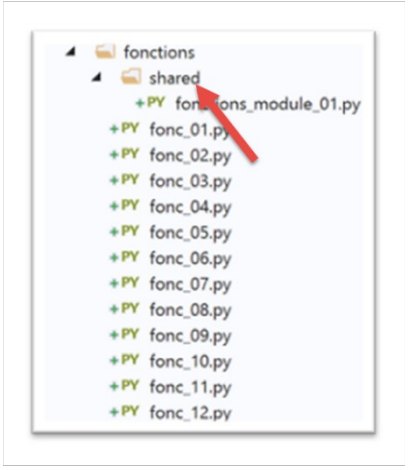
- ligne 3 : la fonction **`[f2]`** utilise la fonction **`[f1]`** définie plus loin dans le script. Cela ne provoque cependant pas d'erreur. On peut donc en conclure que l'ordre de définition des fonctions dans un script Python n'a pas d'importance ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_06.py
2.  310
3.
4.  Process finished with exit code 0
```

6.7 Script `[fonc_07]` : utilisation de modules

Le script **`[fonc_07]`** montre comment isoler les fonctions dans un module.



Nous isolons dans un module des fonctions réutilisables. Plutôt que de les transporter d'un script à l'autre :

- on les place dans un fichier à part que l'on déclare d'une façon particulière ;

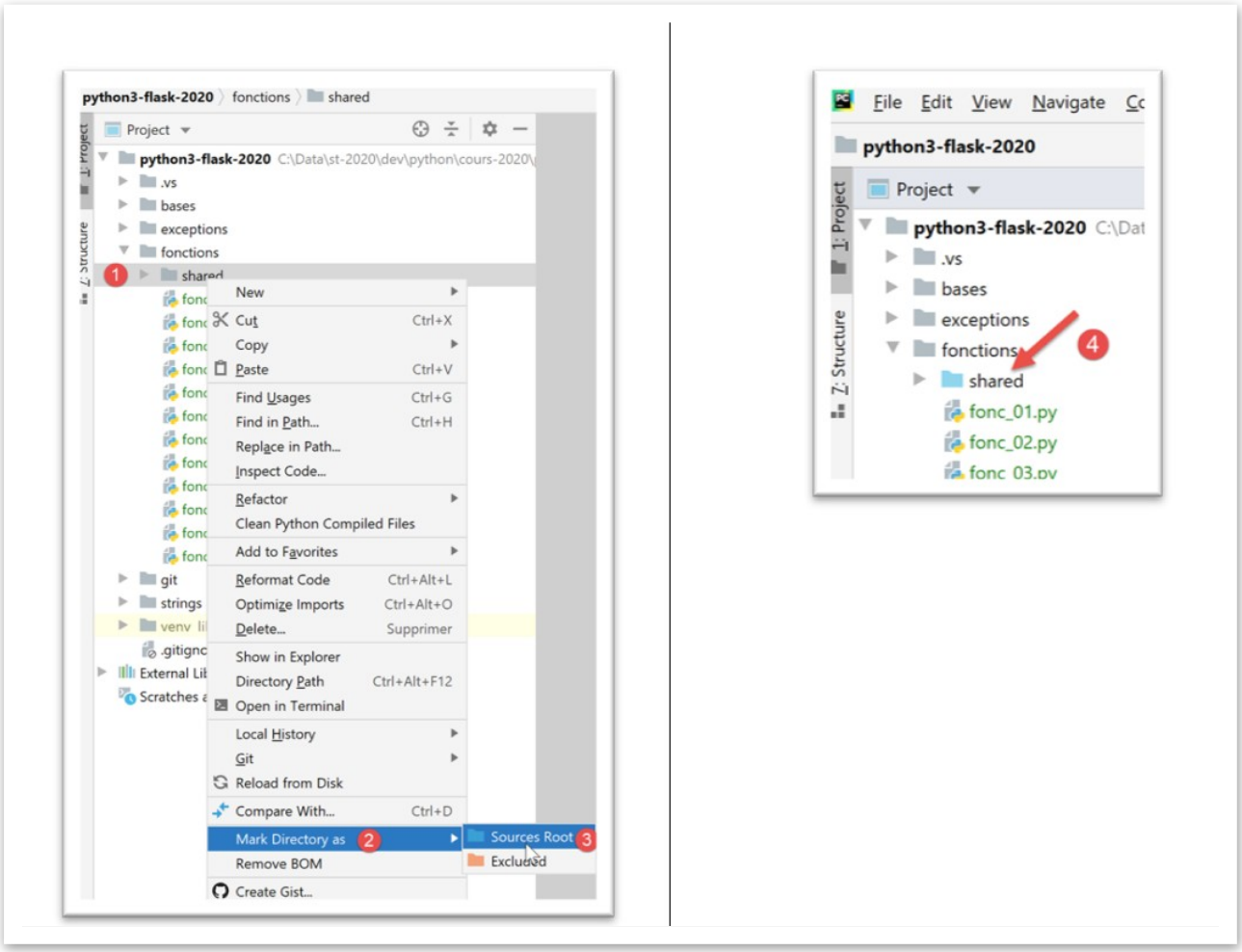
- les scripts ayant besoin de ces fonctions 'importe' le module qui les contient ;

Le script **[fonctions_module_01]** est le suivant :

```
1.  # fonction f2
2.  def f2(a, b):
3.      return f1(a + b)
4.
5.  # fonction f1
6.  def f1(a):
7.      return a + 10
```

Pour que les fonctions du script **[fonctions_module_01]** puissent être référencées par d'autres scripts, il y a différentes façons de faire. Elles diffèrent selon qu'on exécute ou non le script à l'intérieur de **[PyCharm]**.

Sous **[PyCharm]** les modules importés sont cherchés dans des dossiers précis appelés **[Sources Root]**. Il y a deux façons de faire d'un



dossier un **[Sources Root]** :

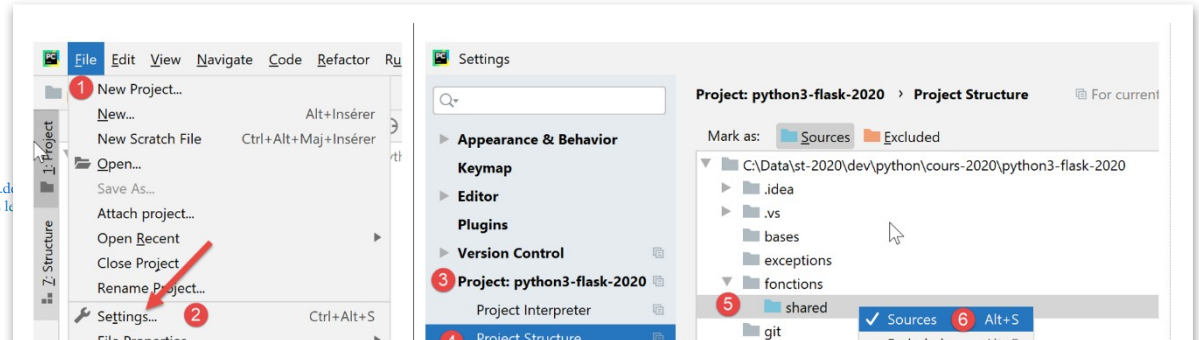
- en **[4]**, le dossier a changé de couleur ;

Après cette opération, le dossier **[fonctions/modules]** est reconnu comme un dossier source. On peut alors écrire dans un script :

```
5.      from fonctions_module_01 import f2
```

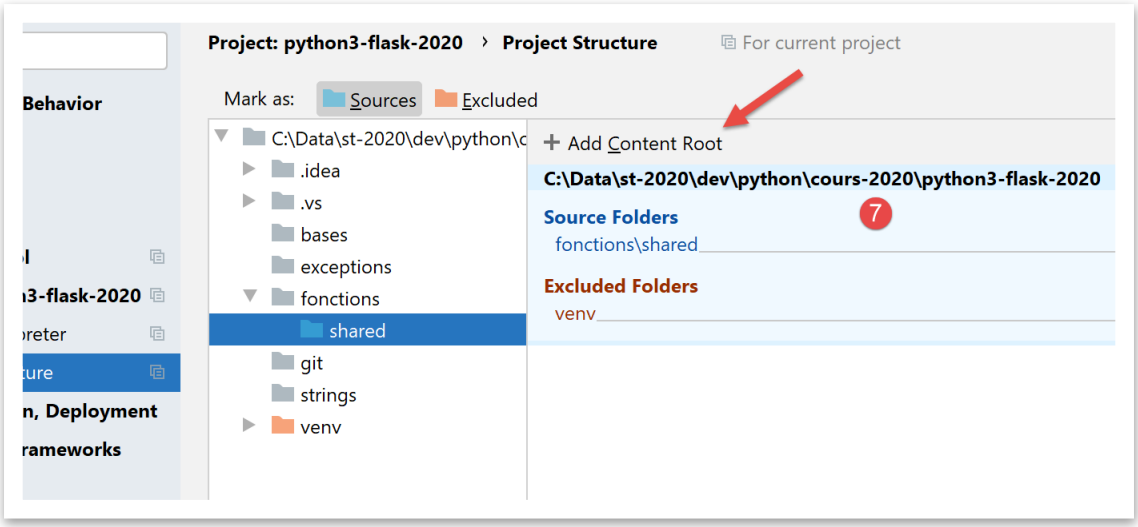
Pour importer / utiliser la fonction f2 définie dans le module **[fonctions_module_01.py]**.

Une autre méthode est de passer par les propriétés du projet :



- ci-dessus, la séquence **[1-6]** permet de faire du dossier **[shared]** un dossier où ranger des modules à importer ;

Nous n'allons pour l'instant déclarer aucun dossier comme **[Sources Root]** à part la racine du projet :



Ceci fait, on peut écrire le script **[fonc-07]** suivant :

```
1.  # utilisation de modules
2.  import sys
3.
4.  # ------------------------ main
5.  print(f"Python path={sys.path}")
6.  from fonctions.shared.fonctions_module_01 import f2
7.
8.  print(f2(100, 200))
```

- ligne 2 : on importe l'objet **[sys]** pour pouvoir utiliser ligne 5, son attribut **[path]** qui donne, ce qu'on appelle le **[Python Path]** : une liste de dossiers qui seront explorés à la recherche de modules importés ;
- ligne 6 : on importe la fonction f2 du module **[fonctions_module_01]**. Pour désigner ce module, on utilise le chemin qui mène de la racine du projet au module. Avec Pycharm, la racine du projet fait toujours partie des dossiers explorés lorsqu'est cherché un module importé dans un script. Ce dossier fait donc partie du **[Python Path]** du projet. C'est ce que la ligne 5 va nous permettre de vérifier ;
- ligne 6 : si on décrivait le chemin qui mène de la racine du projet au dossier **[fonctions_module_01]** on écrirait **[fonctions/shared/fonctions_module_01]**. Dans le chemin d'un module, le signe / est remplacé par le point. On écrit donc **[fonctions.modules.fonctions_module_01]** ;
- après la ligne 6, la fonction f2 est connue. On l'utilise ligne 8 ;

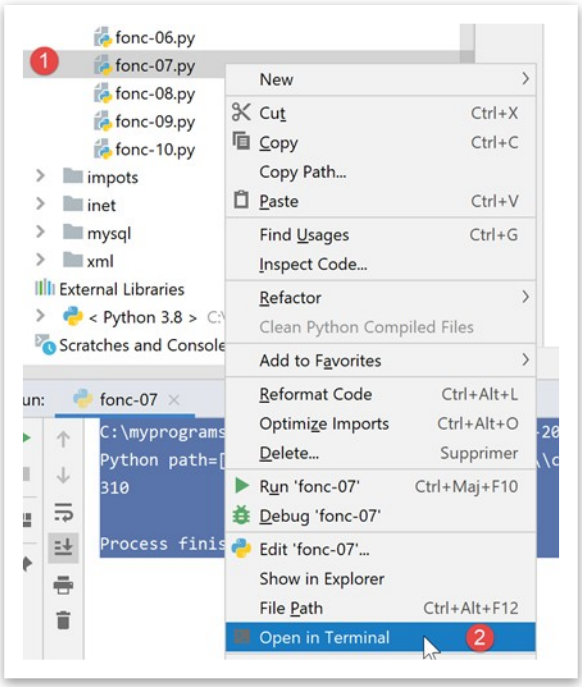
Résultats

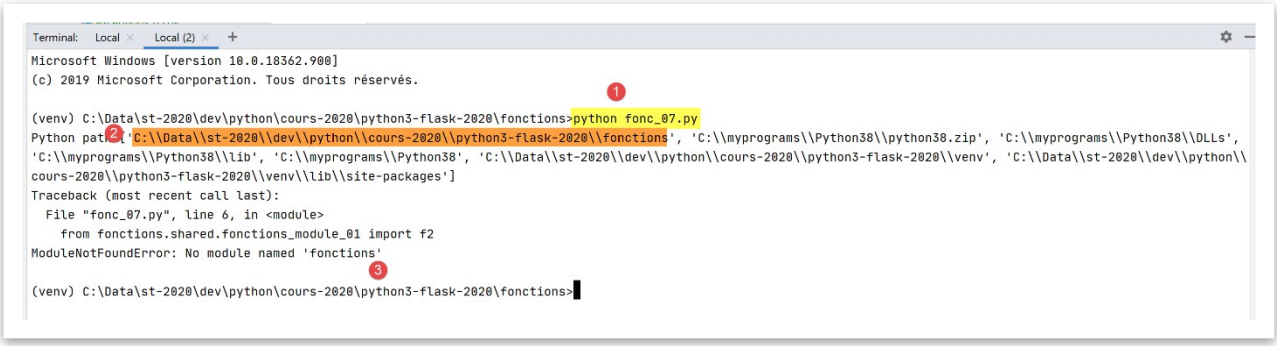
```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_07.py
2.  Python path=['C:\\Data\\st-2020\\dev\\python\\cours-2020\\python3-flask-2020\\fonctions', 'C:\\Data\
    \st-2020\\dev\\python\\cours-2020\\python3-flask-2020', 'C:\\Data\\st-2020\\dev\\python\\cours-2020\
    \python3-flask-2020\\fonctions\\shared', 'C:\\myprograms\\Python38\\python38.zip', 'C:\\myprograms\
    \Python38\\DLLs', 'C:\\myprograms\\Python38\\lib', 'C:\\myprograms\\Python38', 'C:\\Data\\st-2020\\dev\
    \python\\cours-2020\\python3-flask-2020\\venv', 'C:\\Data\\st-2020\\dev\\python\\cours-2020\\python3-
    flask-2020\\venv\\lib\\site-packages']
3.  310
4.
5.  Process finished with exit code 0
```

Ci-dessus :

- surlignée en vert, on voit que la racine du projet fait partie du **[Python Path]** ;
- surligné en jaune, on voit que le dossier du script exécuté fait également partie du **[Python Path]** ;
- les autres éléments du **[Python Path]** proviennent directement du dossier d'installation de Python ;

Que se passe-t-il lorsqu'on n'utilise pas PyCharm pour exécuter **[fonc-07]** ?





- en **[1]**, on exécute le script **[fonc-07]**. On est positionnés dans le dossier **[fonctions]** ;
- en **[2]**, on voit que le dossier d'exécution fait partie du **[Python Path]**. C'est toujours ainsi. On peut voir également que le dossier racine **[C:\\Data\\st-2020\\dev\\python\\cours-2020\\python3-flask-2020]** ne fait pas partie du **[Python Path]** ;
- en **[3]**, l'interpréteur Python déclare qu'il ne trouve pas le module **[fonctions]** ;

Pour chercher le module importé **[fonctions.shared.fonctions_module_01]**, l'interpréteur Python cherche dans les dossiers du **[Python Path]** un sous-dossier nommé **[fonctions]**. Il ne le trouve nulle part. En effet le sous-dossier **[fonctions]** est sous le dossier **[C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020]** qui ne fait pas partie du **[Python Path]**.

Le script **[fonc-08]** donne une solution possible à ce problème.

6.8 Script [fonc_08] : ajouter des dossiers au [Python Path]

Il est possible de modifier le **[Python Path]** par programmation comme le montre le script **[fonc-08]** :

```
1.  # utilisation de modules
2.  import os
3.  import sys
4.
5.  # dossier du script
6.  script_dir = os.path.dirname(os.path.abspath(__file__))
7.  # Python Path avant modification
8.  print(f"Python path avant={sys.path}")
9.  # on ajoute le dossier [shared] au Python Path
10. sys.path.append(f"{script_dir}/shared")
11. # Python Path après modification
```

```
12. print(f"Python path après={sys.path}")
13.
14. # import f2
15. from fonctions_module_01 import f2
16.
17. # ------------------------ main
18. print(f2(100, 200))
```

Notes

- ligne 4 : la variable spéciale **[__file__]** est le nom du script qui s'exécute. Selon le contexte d'exécution, ce nom peut être absolu (Pycharm) ou relatif (console). La fonction **[os.path.abspath]** donne le nom absolu du fichier dont on lui passe le nom. La fonction **[os.path.dirname]** donne le nom absolu du dossier contenant le fichier dont on lui passe le nom ;
- ligne 10 : **[sys.path]** est un tableau contenant les noms des dossiers à explorer lorsqu'un module est recherché. On ajoute à ce tableau la racine du projet définie ligne 4 ;
- on affiche le **[Python Path]** avant (ligne 8) et après (ligne 12) modification ;
- ligne 15 : on importe le module **[fonctions_module_01]** qui contient la fonction f2 ;

L'exécution dans PyCharm donne les résultats suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_08.py
2.  Python path avant=['C:\\Data\\st-2020\\dev\\python\\cours-2020\\python3-flask-2020\\fonctions', 'C:\\Data\
    \st-2020\\dev\\python\\cours-2020\\python3-flask-2020', 'C:\\Data\\st-2020\\dev\\python\\cours-2020\
    \python3-flask-2020\\fonctions\\shared', 'C:\\myprograms\\Python38\\python38.zip', 'C:\\myprograms\
    \Python38\\DLLs', 'C:\\myprograms\\Python38\\lib', 'C:\\myprograms\\Python38', 'C:\\Data\\st-2020\\dev\
    \python\\cours-2020\\python3-flask-2020\\venv', 'C:\\Data\\st-2020\\dev\\python\\cours-2020\\python3-
    flask-2020\\venv\\lib\\site-packages']
3.  Python path après=['C:\\Data\\st-2020\\dev\\python\\cours-2020\\python3-flask-2020\\fonctions', 'C:\\Data\
    \st-2020\\dev\\python\\cours-2020\\python3-flask-2020', 'C:\\Data\\st-2020\\dev\\python\\cours-2020\
    \python3-flask-2020\\fonctions\\shared', 'C:\\myprograms\\Python38\\python38.zip', 'C:\\myprograms\
    \Python38\\DLLs', 'C:\\myprograms\\Python38\\lib', 'C:\\myprograms\\Python38', 'C:\\Data\\st-2020\\dev\
    \python\\cours-2020\\python3-flask-2020\\venv', 'C:\\Data\\st-2020\\dev\\python\\cours-2020\\python3-
    flask-2020\\venv\\lib\\site-packages', 'C:\\Data\\st-2020\\dev\\python\\cours-2020\\python3-flask-2020\
    \fonctions/shared']
4.  310
5.
6.  Process finished with exit code 0
```

- ligne 3 : on voit que le dossier **[shared]** est présent deux fois dans le **[Python Path]**. On peut éviter cela mais ici ça ne gêne pas ;
- ligne 4 : la fonction f2 a bien été exécutée ;

Maintenant exécutons **[fonc-08]** dans un terminal :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\fonctions>python fonc_08.py
2.  Python path avant=['C:\\Data\\st-2020\\dev\\python\\cours-2020\\python3-flask-2020\\fonctions', 'C:\
    \myprograms\\Python38\\python38.zip', 'C:\\myprograms\\Python38\\DLLs', 'C:\\myprograms\\Python38\\lib',
    'C:\\myprograms\\Python38', 'C:\\Data\\st-2020\\dev\\python\\cours-2020\\python3-flask-2020\\venv', 'C:\
    \Data\\st-2020\\dev\\python\\cours-2020\\python3-flask-2020\\venv\\lib\\site-packages']
3.  Python path après=['C:\\Data\\st-2020\\dev\\python\\cours-2020\\python3-flask-2020\\fonctions', 'C:\
    \myprograms\\Python38\\python38.zip', 'C:\\myprograms\\Python38\\DLLs', 'C:\\myprograms\\Python38\\lib',
    'C:\\myprograms\\Python38', 'C:\\Data\\st-2020\\dev\\python\\cours-2020\\python3-flask-2020\\venv', 'C:\
    \Data\\st-2020\\dev\\python\\cours-2020\\python3-flask-2020\\venv\\lib\\site-packages', 'C:\\Data\\st-2020\
    \dev\\python\\cours-2020\\python3-flask-2020\\fonctions/shared']
19. 310
```

- ligne 2 : comme précédemment, le dossier **[shared]** n'est pas dans le **[Python Path]** ;
- ligne 3 : maintenant il y est ;
- ligne 4 : la fonction f2 a été trouvée ;

6.9 Script [fonc_09] : déclaration du type des paramètres

Le script **[fonc_09]** montre qu'on peut déclarer le type des paramètres d'une fonction ainsi que celui du résultat. Cependant cette déclaration n'est utile que pour la documentation de la fonction. L'interpréteur Python ne vérifie pas que les paramètres effectifs de la fonction ont bien le type attendu. Cependant, Pycharm signale les incohérences de types entre paramètres effectifs et formels. Cette seule raison suffit pour rendre les déclarations de type indispensables.

Le script est le suivant :

```
1.  # une fonction avec indication du type des paramètres
2.  # cela sert de documentation uniquement car l'interpréteur python n'en tient pas compte
3.
4.
5.  def show(param: int) -> int:
6.      print(f"param={param}, type(param)={type(param)}")
7.      return param + 1
8.
9.
10. # main -------------------------
11. print(show(4))
12. show("xyz")
```

Notes :

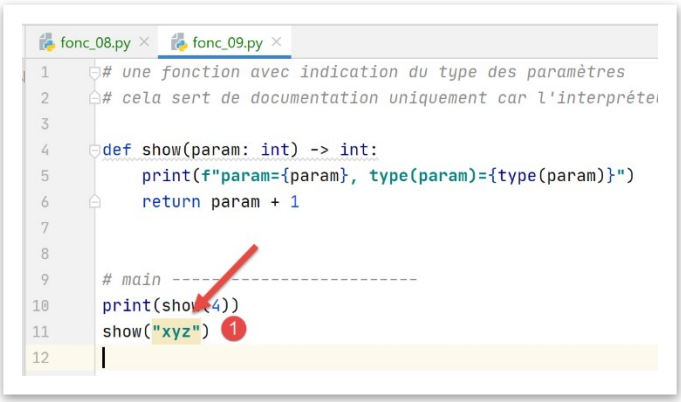
- ligne 5 : on déclare que le paramètre formel **[param]** est de type **[int]** et que le résultat de la fonction est également de type **[int]** ;
- ligne 11 : le paramètre effectif de la fonction **[show]** a le bon type ;
- ligne 12 : le paramètre effectif de la fonction **[show]** n'a pas le bon type ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_09.py
2.  param=4, type(param)=<class 'int'>
3.  5
4.  param=xyz, type(param)=<class 'str'>
5.  Traceback (most recent call last):
6.    File "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_09.py", line 11, in
    <module>
7.      show("xyz")
8.    File "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_09.py", line 6, in show
9.      return param + 1
10. TypeError: can only concatenate str (not "int") to str
11.
12. Process finished with exit code 1
```

- ligne 10 : le type du paramètre **[param]** est de type **[str]**. Lorsque ce message s'affiche, on est déjà entrés dans le code de la fonction **[show]**. L'interpréteur Python a donc accepté que le paramètre effectif da la fonction **[show]** soit de type **[str]** ;
- la ligne 7 du code provoque l'exception reflétée par les lignes 4-10 des résultats ;

PyCharm indique néanmoins qu'il y a une anomalie :



En **[1]**, PyCharm a surligné l'appel erroné.

6.10 Script [fonc_10] : paramètres nommés

Pour passer des paramètres à une fonction, on peut utiliser les noms des paramètres formels de celle-ci. Dans ce cas, on n'est pas obligés de respecter l'ordre des paramètres formels :

```
1.  # on peut désigner les paramètres effectifs par leurs noms formels
2.  def f(x, y):
3.      return x + y
4.
5.
```

```
6.  # main
7.  print(f(y=10, x=3))
```

Notes

- ligne 2 : la fonction f a deux paramètres formels x et y ;
- ligne 7 : lors de l'appel à la fonction f, on peut utiliser les noms des paramètres formels. Cette pratique peut être utile au moins dans deux cas :
- la fonction a beaucoup de paramètres dont la plupart ont une valeur par défaut. Lors de l'appel, la technique précédente permet d'initialiser les seuls paramètres dont on ne veut pas utiliser la valeur par défaut ;
- si les paramètres formels ont un nom significatif, alors l'utilisation de paramètres nommés dans l'appel de la fonction améliore la lisibilité du code ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_10.py
2.  13
3.
4.  Process finished with exit code 0
```

6.11 Script `[fonc_11]` : fonction récursive

Le script **`[fonc_11]`** est un exemple de fonction récursive (qui s'appelle elle-même) :

```
1.  # fonction récursive
2.  def fact(i: int) -> int:
3.      # factorielle(1) est 1
4.      # une fonction récursive doit se terminer à un certain moment
5.      if i == 1:
6.          return 1
7.      else:
8.          # factorielle(i)=i*factorielle(i-1)
9.          return i * fact(i - 1)
10.
11.
12. # ---------- main
13. print(f"fact(8)={fact(8)}")
```

Commentaires

- lignes 1-9 : la fonction factorielle ;
- ligne 9 : la fonction **`[factorielle]`** s'appelle elle-même ;
- lignes 5-6 : une fonction récursive doit toujours s'arrêter lorsqu'une condition est réalisée sinon on a une récursion infinie ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_11.py
2.  fact(8)=40320
3.
4.  Process finished with exit code 0
```

6.12 Script `[fonc_12]` : fonction récursive

La fonction **`[fonc_12]`** donne davantage de détails sur le fonctionnement de la récursivité :

```
1.  # fonction récursive
2.  # comportement du paramètre j
3.
4.
5.  def fact(i: int, j: int) -> int:
6.      # arrêt de la fonction récursive
7.      if i == 1:
8.          print(f"j={j}")
9.          return 1
10.     else:
11.         # on manipule j
12.         j += 1
```

```
13.         print(f"avant fact j={j}")
14.         # récursivité
15.         f = fact(i - 1, j)
16.         print(f"après fact j={j}")
17.         # résultat
18.         return i * f
19.
20.
21. # ---------- main
22. print(f"fact(8)={fact(8, 0)}")
```

Commentaires

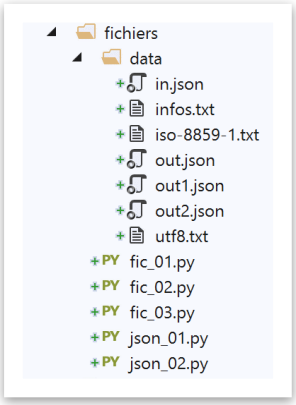
- ligne 5 : on s'intéresse toujours à la fonction factorielle. On lui ajoute le paramètre **[j]** ;
- ligne 12 : la variable j est régulièrement incrémentée à chaque factorielle. On affiche sa valeur avant (ligne 12) et après (ligne 16) la récursivité (ligne 15) ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fonctions/fonc_12.py
2.  avant fact j=1
3.  avant fact j=2
4.  avant fact j=3
5.  avant fact j=4
6.  avant fact j=5
7.  avant fact j=6
8.  avant fact j=7
9.  j=7
10. après fact j=7
11. après fact j=6
12. après fact j=5
13. après fact j=4
14. après fact j=3
15. après fact j=2
16. après fact j=1
17. fact(8)=40320
18.
19. Process finished with exit code 0
```

- lignes 2-8 : on voit que la valeur de **[j]** croît tant que la récursivité continue jusqu'à rencontrer la condition où la récursivité s'arrête. A partir de ce moment, le retour des appels à la fonction **[fact]** s'opère en sens inverse des appels ;
- lignes 10-16 : ces affichages traduisent les retours successifs de l'appel de la factorielle. La variable **[j]** retrouve ses valeurs jusqu'à sa valeur initiale 1 ;

7 Les fichiers texte



7.1 Script [fic_01] : lecture / écriture d'un fichier texte

Le script suivant illustre un exemple d'exploitation de fichiers texte :

```
1.  # imports
2.  import sys
3.
4.
5.  # création puis exploitation séquentielle d'un fichier texte
6.  # celui-ci est un ensemble de lignes de la forme login:pwd:uid:gid:infos:dir:shell
7.  # chaque ligne est mis dans un dictionnaire sous la forme login => uid:gid:infos:dir:shell
8.
9.  # ------------------------------------------------------------------------
10. def affiche_infos(dico: dict, clé: str):
11.     # affiche la valeur associée à clé dans le dictionnaire dico si elle existe
12.     if clé in dico.keys():
13.         # on affiche la valeur associée à la clé
14.         print(f"{clé} : {dico[clé]}")
15.     else:
16.         # clé n'est pas une clé du dictionnaire dico
17.         print(f"la clé [{clé}] n'existe pas")
18.
19.
20. # main ----------------------------------------------
21. # on fixe le nom du fichier
22. FILE_NAME = "./data/infos.txt"
23.
24. # création et remplissage du fichier texte
25. fic = None
26. try:
27.     # ouverture du fichier en écriture (w=write)
28.     fic = open(FILE_NAME, "w")
29.     # on génère un contenu arbitraire
30.     for i in range(1, 101):
31.         # une ligne
32.         ligne = f"login{i}:pwd{i}:uid{i}:gid{i}:infos{i}:dir{i}:shell{i}"
33.         # est écrite dans le fichier texte
34.         fic.write(f"{ligne}\n")
35. except IOError as erreur:
36.     print(f"Erreur d'exploitation du fichier {FILE_NAME} : {erreur}")
37.     sys.exit()
38. finally:
39.     # on ferme le fichier s'il a été ouvert
40.     if fic:
41.         fic.close()
42.
43. # on l'ouvre en lecture
44. fic = None
45. try:
46.     # ouverture du fichier en lecture
47.     fic = open(FILE_NAME, "r")
48.     # dictionnaire vide au départ
```

```
49.     dico = {}
50.     # chaque ligne est mis dans le dictionnaire [dico] sous la forme login => uid:gid:infos:dir:shell
51.     # lecture 1ère ligne en enlevant espaces de début et fin de ligne
52.     ligne = fic.readline().strip()
53.     # tant que la ligne n'est pas vide
54.     while ligne != '':
55.         # on met la ligne dans un tableau
56.         infos = ligne.split(":")
57.         # on récupère le login
58.         login = infos[0]
59.         # on néglige le pwd
60.         infos[0:2] = []
61.         # on crée une entrée dans le dictionnaire
62.         dico[login] = infos
63.         # lecture ligne suivante
64.         ligne = fic.readline().strip()
65. except IOError as erreur:
66.     print(f"Erreur d'exploitation du fichier {FILE_NAME} : {erreur}")
67.     sys.exit()
68. finally:
69.     # on ferme le fichier s'il a été ouvert
70.     if fic:
71.         fic.close()
72.
73. # exploitation du dictionnaire dico
74. affiche_infos(dico, "login10")
75. affiche_infos(dico, "X")
```

Notes :

- ligne 28 : ouverture du fichier en écriture (w=write). Si le fichier existe déjà, il sera écrasé ;
- lignes 30-34 : on génère 100 lignes dans le fichier texte ;
- ligne 34 : pour écrire une ligne dans le fichier texte. La méthode **[write]** ne rajoute pas la marque de fin de ligne. Il faut donc prévoir celle-ci dans le texte écrit ;
- lignes 35-37 : gestion de l'éventuelle exception ;
- ligne 37 : abandon de l'exécution du script (néanmoins après l'exécution de la clause finally) ;
- lignes 38-41 : dans tous les cas, erreur ou pas, on fermet le fichier s'il est ouvert ;
- ligne 47 : ouverture du fichier en lecture (r=read) ;
- ligne 49 : définition d'un dictionnaire vide ;
- ligne 52 : la méthode **[readline]** lit une ligne de texte, marque de fin de ligne incluse. La méthode **[strip]** supprime les "espaces" de début et fin de chaîne. Par "espace", il faut entendre caractère blanc, marque de fin de ligne, saut de page, tabulation, et quelques autres. Donc ici, **[ligne]** n'aura pas les caractères de fin de ligne **[\r\n]** (windows) ou **[\n]** (unix) ;
- ligne 54 : on exploite le fichier tant qu'on n'a pas récupéré une ligne vide ;
- lignes 54-64 : le fichier texte est transféré dans le dictionnaire **[dico]**. La clé est le champ **[login]**, la valeur les champs **[uid:gid:infos:dir:shell]** ;
- lignes 65-67 : gestion de l'éventuelle exception ;
- lignes 68-71 : fermeture du fichier dans tous les cas, erreur ou pas ;
- lignes 74-75 : exploitation du dictionnaire **[dico]** ;

Le fichier **[data/infos.txt]** :

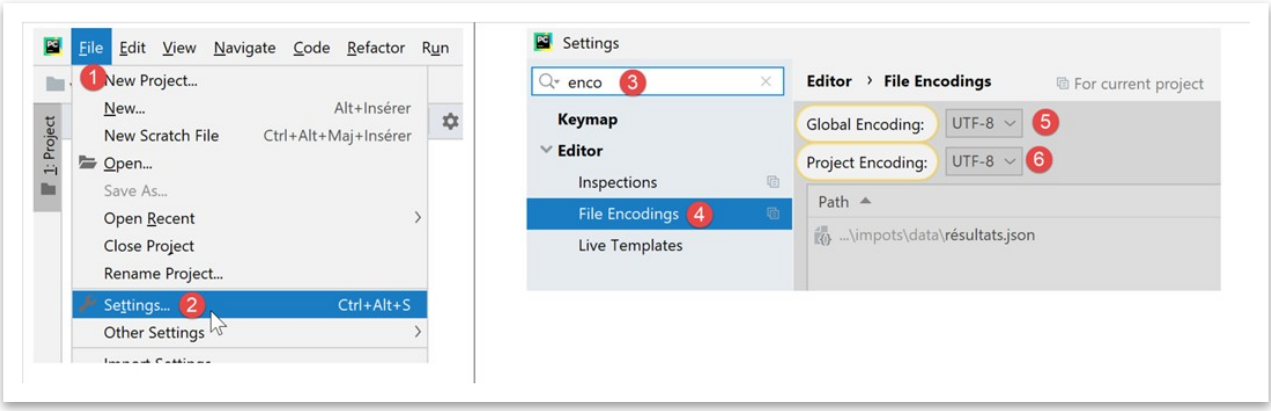
```
1.  login0:pwd0:uid0:gid0:infos0:dir0:shell0
2.  login1:pwd1:uid1:gid1:infos1:dir1:shell1
3.  login2:pwd2:uid2:gid2:infos2:dir2:shell2
4.  …
5.  login98:pwd98:uid98:gid98:infos98:dir98:shell98
6.  login99:pwd99:uid99:gid99:infos99:dir99:shell99
```

Les résultats écran :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fichiers/fic_01.py
2.  login10 : ['uid10', 'gid10', 'infos10', 'dir10', 'shell10']
3.  la clé [X] n'existe pas
4.
5.  Process finished with exit code 0
```

7.2 Script [fic_02] : gérer des fichiers texte encodés en UTF-8

Dans la suite du document, nous allons gérer des fichiers texte codés uniquement en UTF-8. Nous allons tout d'abord configurer PyCharm :



- en **[5-6]** : choisir l'encodage UTF-8 pour les fichiers du projet ;

Pour créer un fichier encodé en UTF-8, on pourra procéder comme suit (fic-02) :

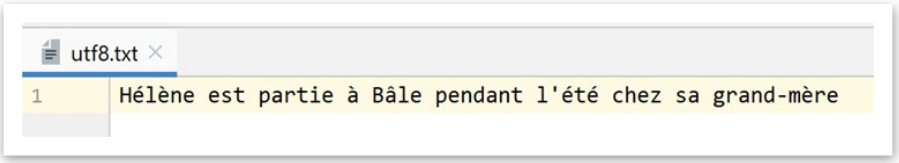
```
1.  # imports
2.  import codecs
3.
4.  # écriture utf8 dans un fichier texte
5.  # on ne gère pas les exceptions
6.  file=codecs.open("./data/utf8.txt","w","utf8")
7.  file.write("Hélène est partie à Bâle pendant l'été chez sa grand-mère")
8.  file.close()
```

Notes

- ligne 2 : pour gérer l'encodage des fichiers, on importe le module **[codecs]** ;
- ligne 6 : la méthode **[codecs.open]** s'utilise comme la classique fonction **[open]**. On peut cependant préciser l'encodage souhaité (création) ou existant (lecture). Après l'ouverture, l'objet **[file]** obtenu ligne 6, s'utilise comme un fichier classique ;
- ligne 7 : on a utilisé des caractères accentués qui ont la plupart du temps des représentations différentes selon le code de caractères utilisé ;

Résultats

Lorsqu'on ouvre le fichier **[data/utf8.txt]** obtenu (cf. ligne 6), on obtient le résultat suivant :

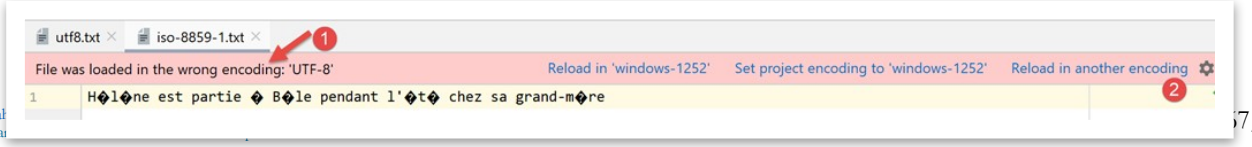


7.3 Script [fic_03] : gérer des fichiers texte encodés en ISO-8859-1

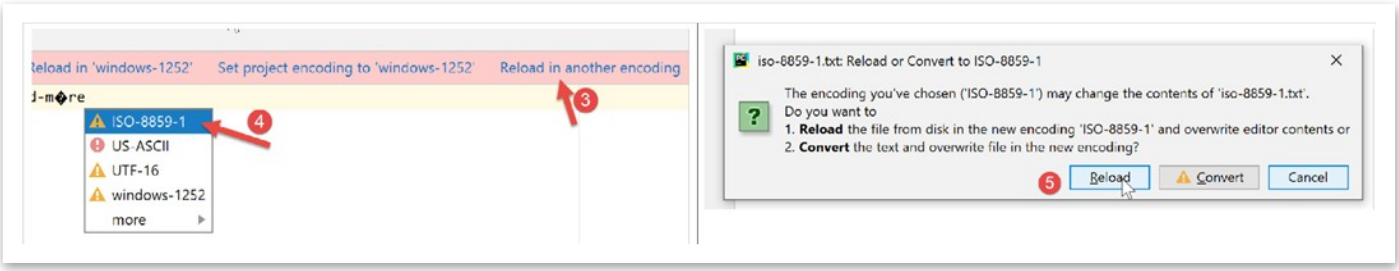
Le script **[fic_03]** fait la même chose que le script **[fic_02]** mais code le fichier texte en ISO-8859-1. On veut montrer la différence des fichiers obtenus :

```
1.  # imports
2.  import codecs
3.
4.  # écriture iso-8859-1 dans un fichier texte
5.  # on ne gère pas les exceptions
6.  file=codecs.open("./data/iso-8859-1.txt","w","iso-8859-1")
7.  file.write("Hélène est partie à Bâle pendant l'été chez sa grand-mère")
8.  file.close()
```

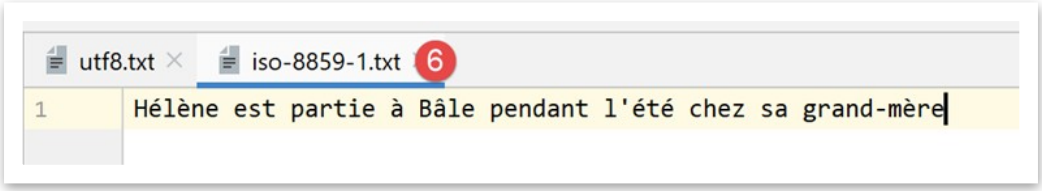
Lorsqu'on ouvre le fichier **[data/iso-8859-1]** créé ligne 6, on obtient le résultat suivant :



Parce que nous avons configuré le projet pour fonctionner avec des fichiers UTF-8, PyCharm a essayé d'ouvrir le fichier **[iso-8859-1.txt]** en UTF-8. Il est capable de voir **[1]** que le fichier n'est pas du UTF-8. Il propose alors **[2]** de recharger le fichier dans un autre encodage :

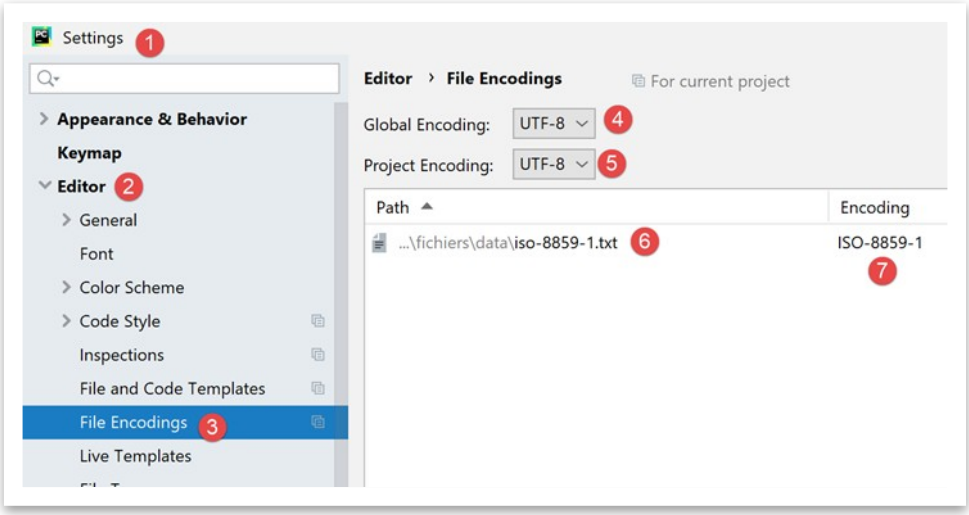


- en **[3-5]** : on recharge le fichier en utilisant un codage ISO-8859-1 ;



- en **[6]**, le même fichier mais affiché avec un encodage différent ;

Si on retourne dans les paramétrages du projet :

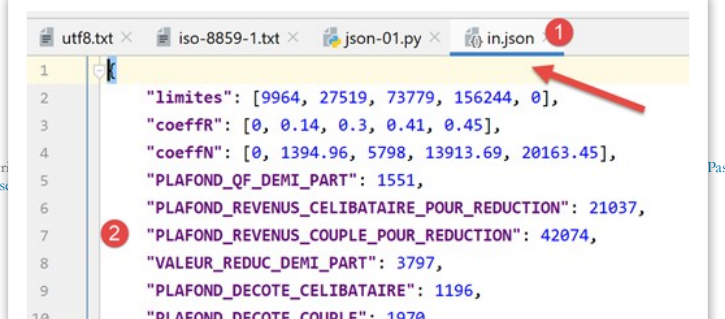


- on voit qu'en **[6-7]**, Pycharm a noté le fait que le fichier **[iso-8859-1.txt]** devait être ouvert avec un encodage ISO-8859-1. C'est donc une exception à la règle **[5]** ;

7.4 Script [json_01] : gestion d'un fichier jSON

JSON signifie **J**ava**S**cript **O**bject **N**otation. Comme son nom l'indique c'est un mode de représentation texte des objets du langage Javascript. Nous l'utiliserons ici avec des objets Python.

Le fichier jSON géré **[data/in.json]** sera le suivant :



- en **[2]**, on voit que le contenu texte du fichier **[in.json]** pourrait représenter un dictionnaire Python. PyCharm a mis en forme (Ctrl-Alt-L) ce texte mais il serait sur une ligne que ça ne changerait rien. La forme du texte n'a aucune importance tant qu'il représente syntaxiquement un objet Python ;

Le script **[json-01]** montre comment exploiter ce fichier :

```
1.  # imports
2.  import codecs
3.  import json
4.  import sys
5.
6.  # lecture / écriture d'un fichier jSON
7.  inFile=None
8.  outFile=None
9.  try:
10.     # ouverture du fichier jSON en lecture
11.     inFile = codecs.open("./data/in.json", "r", "utf8")
12.     # transfert du contenu dans un dictionnaire
13.     data = json.load(inFile)
14.     # affichage des données lues
15.     print(f"data={data}, type(data)={type(data)}")
16.     limites = data['limites']
17.     print(f"limites={limites}, type(limites)={type(limites)}")
18.     print(f"limites[1]={limites[1]}, type(limites[1])={type(limites[1])}")
19.     # transfert du dictionnaire [data] dans un fichier json
20.     outFile = codecs.open("./data/out.json", "w", "utf8")
21.     json.dump(data, outFile)
22. except BaseException as erreur:
23.     # on affiche l'erreur et on quitte
24.     print(f"L'erreur suivante s'est produite : {erreur}")
25.     sys.exit()
26. finally:
27.     # fermeture des fichiers s'ils sont ouverts
28.     if inFile:
29.         inFile.close()
30.     if outFile:
31.         outFile.close()
```

Notes

- ligne 3 : pour gérer du JSON, on importe le module **[json]** ;
- ligne 11 : nous allons gérer des fichiers jSON codés en UTF-8. Ici on ouvre le fichier **[data/in.json]** avec le module **[codecs]** ;
- ligne 13 : la méthode **[json.load]** lit le contenu du fichier jSON et le met dans la variable **[data]**. Le type de cette variable sera ici un dictionnaire ;
- lignes 15-18 : pour montrer qu'on a bien obtenu un dictionnaire Python, on fait quelques affichages d'éléments de celui-ci ;
- lignes 20-21 : on fait l'opération inverse : le dictionnaire **[data]** est mis dans un fichier codé en UTF-8 grâce à la méthode **[json.dump]** ;
- lignes 22-25 : gestion de l'éventuelle exception ;
- lignes 26-31 : dans tous les cas, erreur ou pas, on ferme les fichiers qui ont pu être ouverts ;

Résultats

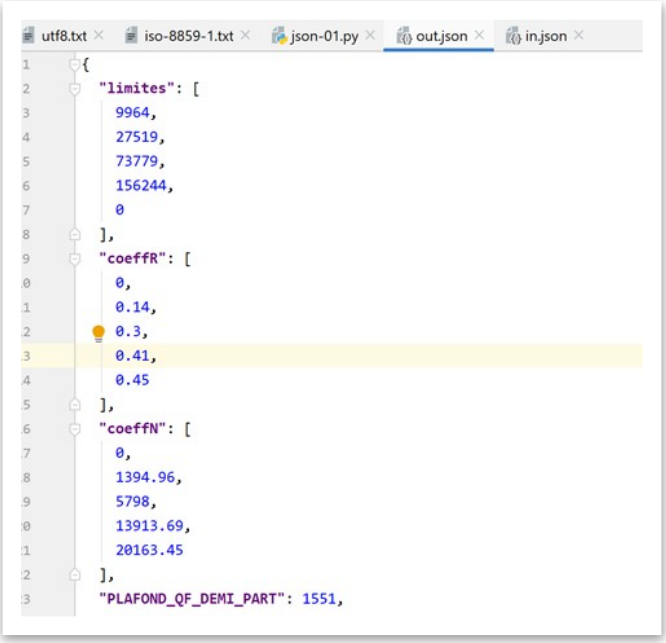
```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fichiers/json_01.py
2.  data={'limites': [9964, 27519, 73779, 156244, 0], 'coeffR': [0, 0.14, 0.3, 0.41, 0.45], 'coeffN': [0,
    1394.96, 5798, 13913.69, 20163.45], 'PLAFOND_QF_DEMI_PART': 1551,
    'PLAFOND_REVENUS_CELIBATAIRE_POUR_REDUCTION': 21037, 'PLAFOND_REVENUS_COUPLE_POUR_REDUCTION': 42074,
    'VALEUR_REDUC_DEMI_PART': 3797, 'PLAFOND_DECOTE_CELIBATAIRE': 1196, 'PLAFOND_DECOTE_COUPLE': 1970,
    'PLAFOND_IMPOT_COUPLE_POUR_DECOTE': 2627, 'PLAFOND_IMPOT_CELIBATAIRE_POUR_DECOTE': 1595,
    'ABATTEMENT_DIXPOURCENT_MAX': 12502, 'ABATTEMENT_DIXPOURCENT_MIN': 437}, type(data)=<class 'dict'>
3.  limites=[9964, 27519, 73779, 156244, 0], type(limites)=<class 'list'>
4.  limites[1]=27519, type(limites[1])=<class 'int'>
5.
6.  Process finished with exit code 0
```

- les lignes 2-4 montrent qu'on a correctement récupéré le dictionnaire présent dans le fichier jSON ;

Maintenant, regardons le contenu du fichier **[data/out.json]** :



Le texte du fichier est sur une ligne. Cependant PyCharm reconnaît les fichiers jSON et on peut les formater, comme les fichiers Python et d'autres par Ctrl-Alt-L. On obtient alors la chose suivante :



7.5 Script [json_02] : gestion des fichiers jSON codés en UTF-8

Un fichier jSON codé en UTF-8 peut avoir deux formes :

```
1.  # imports
2.  import codecs
3.  import json
4.  import sys
5.
6.  # dictionnaire
7.  data = {'marié': 'oui', 'impôt': 1340}
8.
9.  # écriture d'un fichier jSON
10. out_file1 = None
11. out_file2 = None
12. try:
13.     # transfert du dictionnaire [data] dans un fichier json
14.     out_file1 = codecs.open("./data/out1.json", "w", "utf8")
15.     json.dump(data, out_file1, ensure_ascii=True)
16.     # transfert du dictionnaire [data] dans un fichier json
17.     out_file2 = codecs.open("./data/out2.json", "w", "utf8")
18.     json.dump(data, out_file2, ensure_ascii=False)
19. except BaseException as erreur:
20.     # on affiche l'erreur et on quitte
21.     print(f"L'erreur suivante s'est produite : {erreur}")
22.     sys.exit()
23. finally:
24.     # fermeture des fichiers s'ils sont ouverts
25.     if out_file1:
26.         out_file1.close()
27.     if out_file2:
28.         out_file2.close()
29. …
```

- dans ce script, on écrit le dictionnaire **[data]** (ligne 7) dans deux fichiers jSON (lignes 14, 17) ;
- lignes 14, 17 : dans les deux cas, on crée un fichier texte UTF-8 ;
- lignes 15 : lors de l'écriture du dictionnaire, on utilise le paramètre nommé **[ensure_ascii=True]** ;
- lignes 18 : lors de l'écriture du dictionnaire, on utilise le paramètre nommé **[ensure_ascii=False]** ;

Voici les deux fichiers obtenus :



- dans le fichier **[out1.json]**, les caractères accentués ont été remplacés par une série de **caractères** représentant leur code UTF-8. On dit parfois qu'ils ont été 'échappés'. Techniquement, dans le binaire de **[out1.json]**, on trouve pour le caractère é de **[marié]** successivement les codes binaires UTF-8 des 6 caractères **[\u00e9]** ;
- dans le fichier **[out2.json]**, les caractères accentués ont été laissés tels quels. Cela signifie que dans le binaire de **[out2.json]** ces caractères sont représentés par leur code **binaire** UTF-8 (1 seul code UTF-8 donc au lieu de 6 pour out1). Pour le caractère é de **[marié]**, on trouvera ainsi le code binaire **[00e9]** sur 4 octets ;
- c'est la valeur du paramètre **[ensure_ascii]** de la méthode **[json.dump]** qui décide du format utilisé ;

Certaines applications utilisent de l'UTF-8 'échappé' pour leurs fichiers jSON. C'est la valeur **[ensure_ascii=True]** qui doit être alors utilisée. Cette valeur est en fait la valeur par défaut. Si donc on n'utilise pas le paramètre **[ensure_ascii]** on travaillera avec des fichiers jSON UTF-8 échappés.

Le script se poursuit de la façon suivante :

```
1.  # imports
2.  import codecs
3.  import json
4.  import sys
5.
6.  # dictionnaire
7.  data = {'marié': 'oui', 'impôt': 1340}
8.
9.  …
10.
11. # relecture des fichiers jSON
12. in_file1 = None
13. in_file2 = None
14. try:
15.     # transfert du fichier jSON 1 dans un dictionnaire
16.     in_file1 = codecs.open("./data/out1.json", "r", "utf8")
17.     dico1 = json.load(in_file1)
18.     # affichage
19.     print(f"dico1={dico1}")
20.     # transfert du fichier jSON 2 dans un dictionnaire
21.     in_file2 = codecs.open("./data/out2.json", "r", "utf8")
22.     dico2 = json.load(in_file2)
23.     # affichage
24.     print(f"dico2={dico2}")
25. except BaseException as erreur:
26.     # on affiche l'erreur et on quitte
27.     print(f"L'erreur suivante s'est produite : {erreur}")
28.     sys.exit()
29. finally:
30.     # fermeture des fichiers s'ils sont ouverts
31.     if in_file1:
32.         in_file1.close()
33.     if in_file2:
34.         in_file2.close()
```

Notes

- lignes 11-34 : relecture des deux fichiers **[out1.json, out2.json]** et affichage du dictionnaire lu dans chacun des cas ;

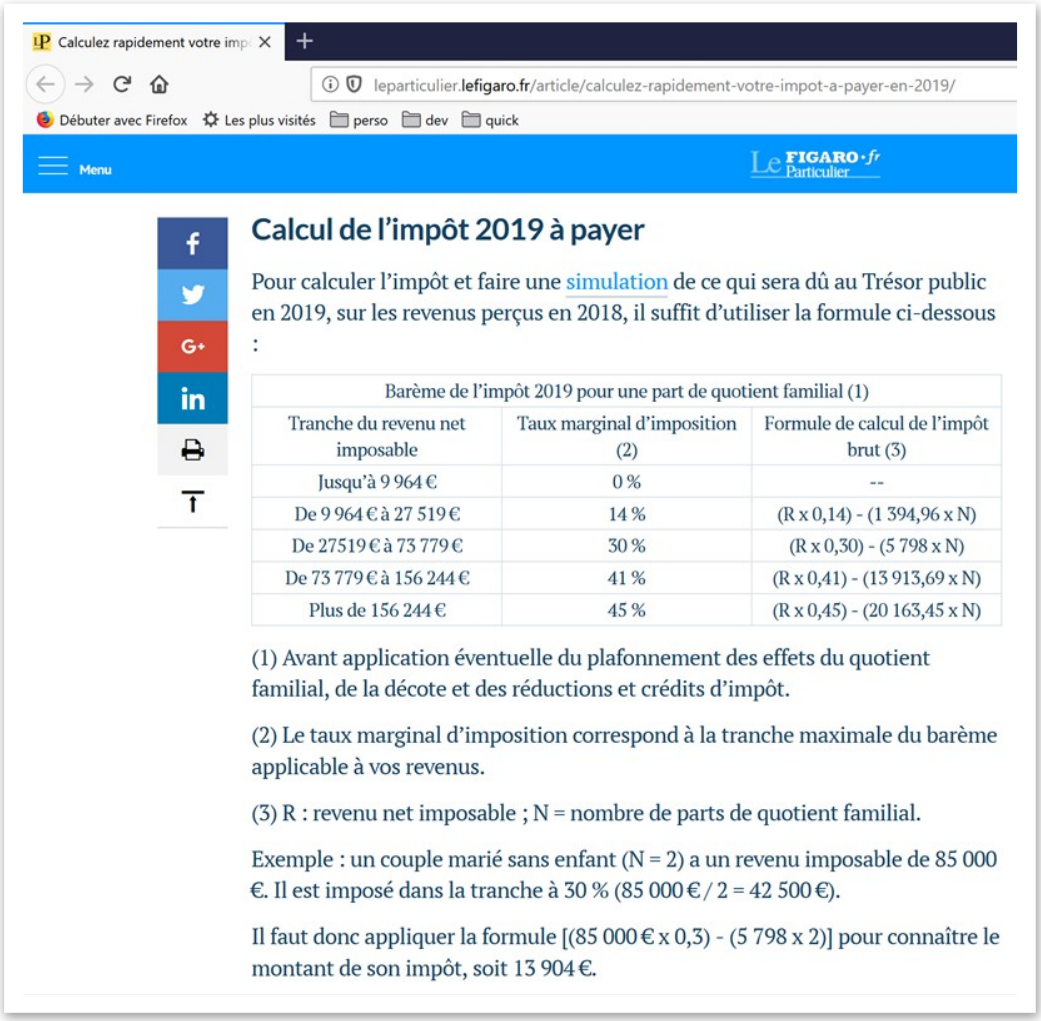
Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/fichiers/json_02.py
2.  dico1={'marié': 'oui', 'impôt': 1340}
3.  dico2={'marié': 'oui', 'impôt': 1340}
4.
5.  Process finished with exit code 0
```

De façon surprenante, on constate qu'on n'a pas eu besoin de préciser à la fonction **[json.load]** (lignes 17, 22) le type d'encodage (échappé ou non) de la chaîne jSON à lire. On récupère dans les deux cas le bon dictionnaire.

8 Exercice d'application – version 1

8.1 Le problème



Calculez rapidement votre impôt

leparticulier.lefigaro.fr/article/calculez-rapidement-votre-impot-a-payer-en-2019/

Débuter avec Firefox | Les plus visités | perso | dev | quick

Menu — Le FIGARO.fr Particulier

Calcul de l'impôt 2019 à payer

Pour calculer l'impôt et faire une simulation de ce qui sera dû au Trésor public en 2019, sur les revenus perçus en 2018, il suffit d'utiliser la formule ci-dessous :

Barème de l'impôt 2019 pour une part de quotient familial (1)		
Tranche du revenu net imposable	Taux marginal d'imposition (2)	Formule de calcul de l'impôt brut (3)
Jusqu'à 9 964 €	0 %	--
De 9 964 € à 27 519 €	14 %	(R x 0,14) - (1 394,96 x N)
De 27519 € à 73 779 €	30 %	(R x 0,30) - (5 798 x N)
De 73 779 € à 156 244 €	41 %	(R x 0,41) - (13 913,69 x N)
Plus de 156 244 €	45 %	(R x 0,45) - (20 163,45 x N)

(1) Avant application éventuelle du plafonnement des effets du quotient familial, de la décote et des réductions et crédits d'impôt.

(2) Le taux marginal d'imposition correspond à la tranche maximale du barème applicable à vos revenus.

(3) R : revenu net imposable ; N = nombre de parts de quotient familial.

Exemple : un couple marié sans enfant (N = 2) a un revenu imposable de 85 000 €. Il est imposé dans la tranche à 30 % (85 000 € / 2 = 42 500 €).

Il faut donc appliquer la formule [(85 000 € x 0,3) - (5 798 x 2)] pour connaître le montant de son impôt, soit 13 904 €.

Le tableau ci-dessus permet de calculer l'impôt dans le cas simplifié d'un contribuable n'ayant que son seul salaire à déclarer. Comme l'indique la note (1), l'impôt ainsi calculé est l'impôt avant trois mécanismes :

- le plafonnement du quotient familial qui intervient pour les hauts revenus ;
- la décote et la réduction d'impôts qui interviennent pour les faibles revenus ;

Ainsi le calcul de l'impôt comprend les étapes suivantes **[http://impotsurlerevenu.org/comprendre-le-calcul-de-l-impot/1217-calcul-de-l-impot-2019.php]** :



CALCUL DE L'IMPÔT 2019 SUR REVENU 2018

L'impôt est calculé en plusieurs étapes :

1. Détermination du **revenu net imposable** : Revenus - Charges déductibles ;
2. Détermination du **quotient familial (QF)** : Revenu net / Parts fiscales ;
3. Calcul de **l'impôt brut** : Application du QF au barème progressif de l'impôt ;
4. Calcul de **l'impôt net** : Application de la décote, du plafonnement du QF et déduction des crédits et réductions d'impôt.

On se propose d'écrire un programme permettant de calculer l'impôt d'un contribuable dans le cas simplifié d'un contribuable n'ayant que son seul salaire à déclarer :

8.1.1 Calcul de l'impôt brut

L'impôt brut peut être calculé de la façon suivante :

On calcule d'abord le nombre de parts du contribuable :
- chaque parent amène 1 part ;
- les deux premiers enfants amènent chacun 1/2 part ;
- les enfants suivants amènent une part chacun :

Le nombre de parts est donc :
- nbParts=1+nbEnfants*0,5+(nbEnfants-2)*0,5 si le salarié n'est pas marié ;
- nbParts=2+nbEnfants*0,5+(nbEnfants-2)*0,5 s'il est marié ;
- où *nbEnfants* est son nombre d'enfants ;
- on calcule le revenu imposable **R=0.9*S** où *S* est le salaire annuel ;
- on calcule le quotient familial **QF=R/nbParts ;**
- on calcule l'impôt brut **I** d'après les données suivantes (2019) :

9964	0	0
27519	0.14	1394.96
73779	0.3	5798
156244	0.4	13913.69
0	0.45	20163.45

Chaque ligne a 3 champs : *champ1*, *champ2*, *champ3*. Pour calculer l'impôt I, on recherche la première ligne où *QF*<=*champ1* et on prend les valeurs de cette ligne. Par exemple, pour un salarié marié avec deux enfants et un salaire annuel S de **50000** euros :

Revenu imposable : **R**=0,9*S=45000
Nombre de parts : **nbParts**=2+2*0,5=3
Quotient familial : **QF**=45000/3=15000

La 1re ligne où QF<=champ1 est la suivante :

```
27519   0.14    1394.96
```

L'impôt *I* est alors égal à **0.14*R – 1394,96*nbParts**=**[0,14*45000-1394,96*3]**=**2115**. L'impôt est arrondi à l'euro inférieur.

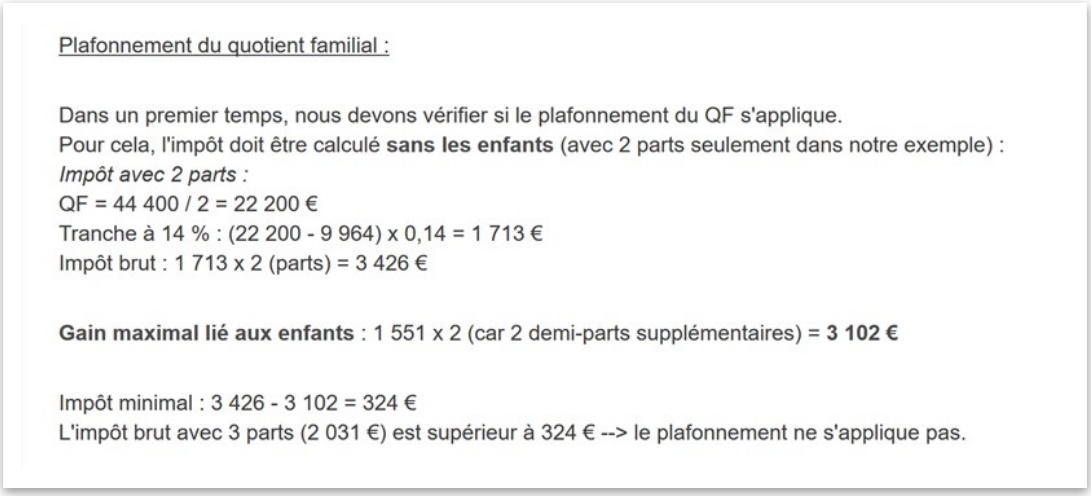
Si la relation *QF*<=*champ1* dès la 1re ligne, alors l'impôt est nul.

Si QF est tel que la relation *QF*<=*champ1* n'est jamais vérifiée, alors ce sont les coefficients de la dernière ligne qui sont utilisés. Ici :

```
0       0.45    20163.45
```

ce qui donne l'impôt brut I=**0.45*R – 20163,45*nbParts**.

8.1.2 Plafonnement du quotient familial


Plafonnement du quotient familial :

Dans un premier temps, nous devons vérifier si le plafonnement du QF s'applique.
Pour cela, l'impôt doit être calculé **sans les enfants** (avec 2 parts seulement dans notre exemple) :
Impôt avec 2 parts :
QF = 44 400 / 2 = 22 200 €
Tranche à 14 % : (22 200 - 9 964) x 0,14 = 1 713 €
Impôt brut : 1 713 x 2 (parts) = 3 426 €

Gain maximal lié aux enfants : 1 551 x 2 (car 2 demi-parts supplémentaires) = **3 102 €**

Impôt minimal : 3 426 - 3 102 = 324 €
L'impôt brut avec 3 parts (2 031 €) est supérieur à 324 € --> le plafonnement ne s'applique pas.

Pour savoir si le plafonnement du quotient familial QF s'applique, on refait le calcul de l'impôt brut sans les enfants. Toujours pour le salarié marié avec deux enfants et un salaire annuel S de **50000** euros :

Revenu imposable : **R**=0,9*S=45000
Nombre de parts : **nbParts**=2 (on ne compte plus les enfants)
Quotient familial : **QF**=45000/2=22500

La 1re ligne où QF<=champ1 est la suivante :

```
   27519    0.14     1394.96
```

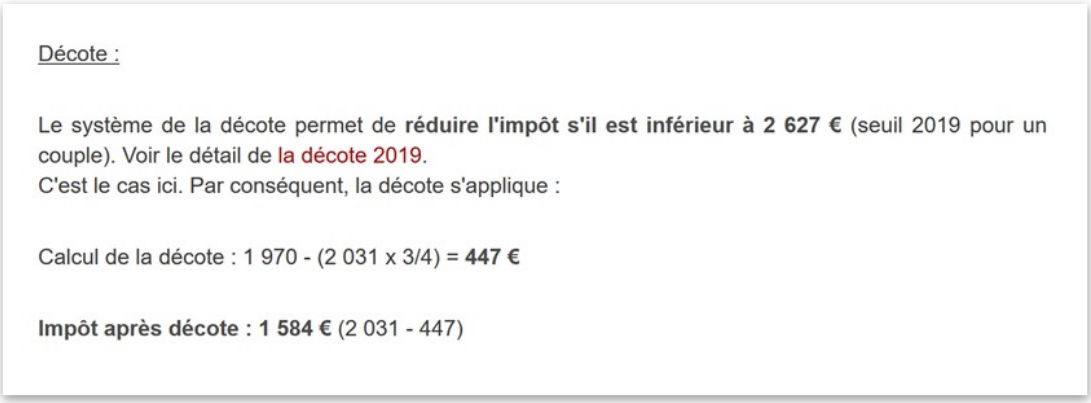
L'impôt *I* est alors égal à **0.14*R – 1394,96*nbParts**=`[0,14*45000-1394,96*2]`=**3510**.

Gain maximal lié aux enfants : 1551 * 2 = 3102 euros
Impôt minimal : 3510-3102 = 408 euros
L'impôt brut avec 3 parts déjà calculé 2115 euros est supérieur à l'impôt minimal 408 euros, donc le plafonnement familial ne s'applique pas ici.

De façon générale, l'impôt brut est **sup(impôt1, impôt2)** où :

- `[impôt1]` : est l'impôt brut calculé avec les enfants ;
- `[impôt2]` : est l'impôt brut calculé sans les enfants et diminué du gain maximal (ici 1551 euros par demi-part) lié aux enfants ;

8.1.3 Calcul de la décote


Décote :

Le système de la décote permet de **réduire l'impôt s'il est inférieur à 2 627 €** (seuil 2019 pour un couple). Voir le détail de la décote 2019.
C'est le cas ici. Par conséquent, la décote s'applique :

Calcul de la décote : 1 970 - (2 031 x 3/4) = **447 €**

Impôt après décote : 1 584 € (2 031 - 447)

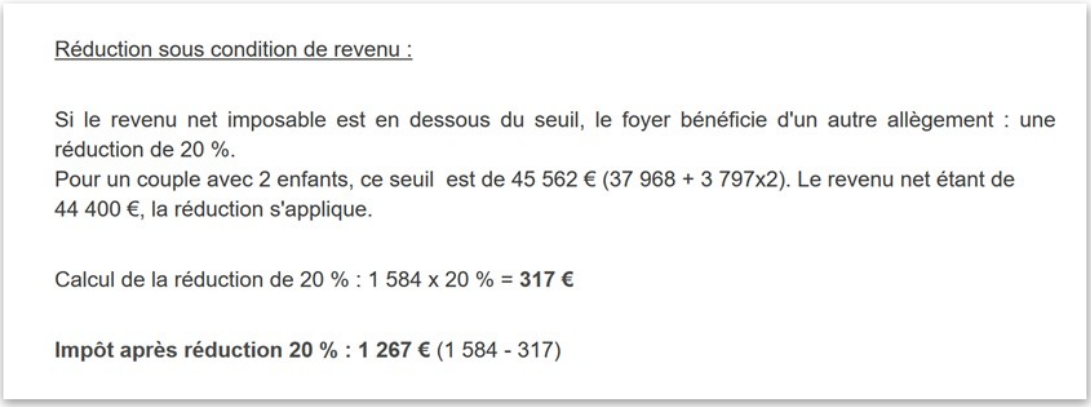
Toujours pour le salarié marié avec deux enfants et un salaire annuel S de **50000** euros :

L'impôt brut (2115) issu de l'étape précédente est inférieur à 2627 euros pour un couple (1595 euros pour un célibataire) : la décôte s'applique donc. Elle est obtenue avec le calcul suivant :

décôte= seuil (couple=1970/célibataire=1196)-0,75* Impôt brut

décôte=1970-0,75*2115=383,75 arrondi à **384** euros.
Nouvel Impôt brut= 2115-384= **1731** euros

8.1.4 Calcul de la réduction d'impôts


Réduction sous condition de revenu :

Si le revenu net imposable est en dessous du seuil, le foyer bénéficie d'un autre allègement : une réduction de 20 %.
Pour un couple avec 2 enfants, ce seuil est de 45 562 € (37 968 + 3 797x2). Le revenu net étant de 44 400 €, la réduction s'applique.

Calcul de la réduction de 20 % : 1 584 x 20 % = **317 €**

Impôt après réduction 20 % : 1 267 € (1 584 - 317)

Au-dessous d'un certain seuil, une réduction de 20 % est faite sur l'impôt brut issu des calculs précédents. En 2019, les seuils sont les suivants :

- célibataire : 21037 euros ;
- couple : 42074 euros ; (le chiffre 37968 utilisé dans l'exemple ci-dessus semble erroné) ;

Ce seuil est augmenté de la valeur : 3797 * (nombre de demi-parts amenées par les enfants).

Toujours pour le salarié marié avec deux enfants et un salaire annuel S de **50000** euros :

- son revenu imposable (45000 euros) est inférieur au seuil (42074+2*3797)=49668 euros ;
- il a donc droit à une réduction de 20 % de son impôt : 1731 * 0,2= 346,2 euros arrondi à 347 euros ;
- l'impôt brut du contribuable devient : 1731-347= **1384** euros ;

8.1.5 Calcul de l'impôt net

Notre calcul s'arrêtera là : l'impôt net à payer sera de **1384** euros. Dans la réalité, le contribuable peut bénéficier d'autres réductions notamment pour des dons à des organismes d'intérêt public ou général.

8.1.6 Cas des hauts revenus

Notre exemple précédent correspond à la majorité des cas de salariés. Cependant le calcul de l'impôt est différent dans le cas des hauts revenus.

8.1.6.1 Plafonnement de la réduction de 10 % sur les revenus annuels

Dans la plupart des cas, le revenu imposable est obtenu par la formule : R=0,9*S où S est le salaire annuel. On appelle cela la réduction des 10 %. Cette réduction est plafonnée. En 2019 :

- elle ne peut être supérieure à 12502 euros ;
- elle ne peut être inférieure à 437 euros ;

Prenons le cas d'un salarié non marié sans enfants et un salaire annuel de 200000 euros :

- la réduction de 10 % est de 20000 euros > 12502 euros. Elle est donc ramenée à 12502 euros ;

8.1.6.2 Plafonnement du quotient familial

Prenon un cas où le plafonnement familial présenté au paragraphe |Plafonnement du quotient familial|, intervient. Prenons le cas d'un couple avec trois enfants et des revenus annuels de 100000 euros. Reprenons les étapes du calcul :

- l'abattement de 10 % est de 10000 euros < 12502 euros. Le revenu imposable **R** est donc 100000-10000=90000 euros ;
- le couple a **nbParts**=2+0,5*2+1=**4** parts ;
- son quotient familial est donc **QF**= R/nbParts=90000/4=**22500** euros ;

- son impôt brut **I1 avec** enfants est I1=0,14*90000-1394,96*4= 7020 euros ;
- son impôt brut **I2 sans** enfants :
- **QF**=90000/2=45000 euros ;
- **I2**=0,3*90000-5798*2=15404 euros ;
- la règle du plafonnement du quotient familial dit que le gain amené par les enfants ne peut dépasser (1551*4 demi-parts)=6204 euros. Or ici, il est I2-I1=15404-7020= 8384 euros, donc supérieur à 6204 euros ;
- l'impôt brut est donc recalculé comme **I3**=**I2**-6204=15404-6204= **9200** euros ;

Ce couple n'aura ni décote, ni réduction et son impôt final sera de **9200** euros.

8.1.7 Chiffres officiels

Le calcul de l'impôt est complexe. Tout au long du document, les tests seront faits avec les exemples suivants. Les résultats sont ceux du simulateur de l'administration fiscale |https://www3.impots.gouv.fr/simulateur/calcul_impot/2019/simplifie/index.htm| :

Contribuable	Résultats officiels	Résultats de l'algorithme du document
Couple avec 2 enfants et des revenus annuels de 55555 euros	Impôt=**2815** euros Taux d'imposition=14 %	Impôt=**2814** euros Taux d'imposition=14 %
Couple avec 2 enfants et des revenus annuels de 50000 euros	Impôt=**1385** euros Décote=720 euros Réduction=0 euros Taux d'imposition=14 %	Impôt=**1384** euros Décote=384 euros Réduction=347 euros Taux d'imposition=14 %
Couple avec 3 enfants et des revenus annuels de 50000 euros	Impôt=**0** euro décote=384 euros Réduction=346 euros Taux d'imposition=14 %	Impôt=**0** euro décote=720 euros Réduction=0 euro Taux d'imposition=14 %
Célibataire avec 2 enfants et des revenus annuels de 100000 euros	Impôt=**19884** euros décote=0 euro Réduction=0 euro Taux d'imposition=41 %	Impôt=**19884** euros Surcote=4480 euros décote=0 euro Réduction=0 euro Taux d'imposition=41 %
Célibataire avec 3 enfants et des revenus annuels de 100000 euros	Impôt=**16782** euros décote=0 euro Réduction=0 euro Taux d'imposition=41 %	Impôt=**16782** euros Surcote=7176 euros décote=0 euro Réduction=0 euro Taux d'imposition=41 %
Couple avec 3 enfants et des revenus annuels de 100000 euros	Impôt=**9200** euros décote=0 euro Réduction=0 euro Taux d'imposition=30 %	Impôt=**9200** euros Surcote=2180 euros décote=0 euro Réduction=0 euro Taux d'imposition=30 %
Couple avec 5 enfants et des revenus annuels de 100000 euros	Impôt=**4230** euros décote=0 euro Réduction=0 euro Taux d'imposition=14 %	Impôt=**4230** euros décote=0 euro Réduction=0 euro Taux d'imposition=14 %
Célibataire sans enfants et des revenus annuels de 100000 euros	Impôt=**22986** euros décote=0 euro Réduction=0 euro Taux d'imposition=41 %	Impôt= **22986** euros Surcote=0 euro décote=0 euro Réduction=0 euro Taux d'imposition=41 %
Couple avec 2 enfants et des revenus annuels de 30000 euros	Impôt=**0** euro décote=0 euro Réduction=0 euro Taux d'imposition=0 %	Impôt=**0** euro décote=0 euro Réduction=0 euro Taux d'imposition=0 %
Célibataire sans enfants et des revenus annuels de 200000 euros	Impôt=**64211** euro décote=0 euro Réduction=0 euro Taux d'imposition=45 %	Impôt= **64210** euros Surcote=7498 euros décote=0 euro Réduction=0 euro

		Taux d'imposition=45 %
Couple avec 3 enfants et des revenus annuels de 200000 euros	Impôt=**42843** euro décote=0 euro Réduction=0 euro Taux d'imposition=41 %	Impôt=**42842** euros Surcote=17283 euros décote=0 euro Réduction=0 euro Taux d'imposition=41 %

Ci-dessus, on appelle surcote, ce que paient en plus les hauts revenus à cause de deux phénomènes :

- le plafonnement de l'abattement de 10 % sur les revenus annuels ;
- le plafonnement du quotient familial ;

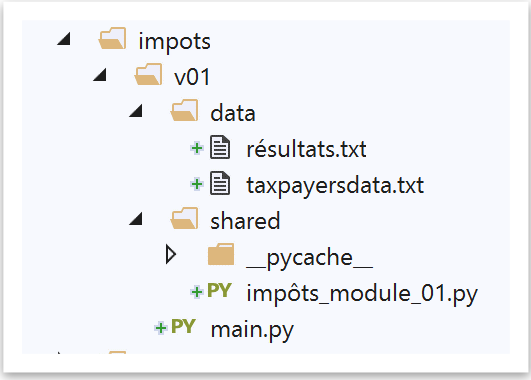
Cet indicateur n'a pu être vérifié car le simulateur de l'administration fiscale ne le donne pas.

On voit que l'algorithme du document donne un impôt juste à chaque fois, avec cependant une marge d'erreur de 1 euro. Cette marge d'erreur provient des arrondis. Toutes les sommes d'argent sont arrondies parfois à l'euro supérieur, parfois à l'euro inférieur. Comme je ne connaissais pas les règles officielles, les sommes d'argent de l'algorithme du document ont été arrondies :

- à l'euro supérieur pour les décotes et réductions ;
- à l'euro inférieur pour les surcotes et l'impôt final ;

Nous allons développer plusieurs versions de l'application de calcul de l'impôt.

8.2 Version 1



8.2.1 Le script principal

Nous présentons un premier programme où :

- les données nécessaires au calcul de l'impôt sont codées en dur dans le code sous forme de listes et de constantes ;
- les données des contribuables (marié, enfants, salaire) sont dans un premier fichier texte **[taxpayersdata.txt]** ;
- les résultats du calcul de l'impôt (marié, enfants, salaire, impôt) sont mémorisés dans un second fichier texte **[résultats.txt]** ;

Le script **[v-01/main.py]** est le suivant :

```
1.  # modules
2.  import sys
3.
4.  from impots.v01.shared.impôts_module_01 import *
5.
6.  # main ----------------------
7.
8.  # constantes
9.  # fichier des contribuables
10. DATA = "./data/taxpayersdata.txt"
11. # fichier des résultats
12. RESULTATS = "./data/résultats.txt"
13.
14. try:
```

```
15.     # lecture des données contribuables
16.     tax_payers = get_taxpayers_data(DATA)
17.     # liste des résultats
18.     results = []
19.     # on calcule l'impôt des contribuables
20.     for tax_payer in tax_payers:
21.         # le calcul de l'impôt renvoie un dictionnaire de clés
22.         # ['marié', 'enfants', 'salaire', 'impôt', 'surcôte', 'décôte', 'réduction', 'taux']
23.         result = calcul_impôt(tax_payer['marié'], tax_payer['enfants'], tax_payer['salaire'])
24.         # le dictionnaire est ajouté à la liste des résultats
25.         results.append(result)
26.     # on enregistre les résultats
27.     record_results(RESULTATS, results)
28. except BaseException as erreur:
29.     # il peut y avoir différentes erreurs : absence de fichier, contenu de fichier incorrect
30.     # on affiche l'erreur et on quitte l'application
31.     print(f"l'erreur suivante s'est produite : {erreur}]\n")
32.     sys.exit()
```

Notes

- ligne 4 : on utilise le module **[impots.v01.modules.impôts_module_01]**. On rappelle que ce chemin est mesuré à partir de la racine du projet PyCharm ;
- ligne 10 : le fichier **[data/taxpayersdata.txt]** est le suivant :

```
1.  oui,2,55555
2.  oui,2,50000
3.  oui,3,50000
4.  non,2,100000
5.  non,3,100000
6.  oui,3,100000
7.  oui,5,100000
8.  non,0,100000
9.  oui,2,30000
10. non,0,200000
11. oui,3,200000
```

Chaque ligne représente un tuple de trois éléments **[marié / pacsé ou pas, nombre d'enfants, salaire annuel en euros]**.

- ligne 12 : le fichier où on placera les résultats du calcul de l'impôt pour chacun des contribuables du fichier **[taxpayersdata.txt]**. Il aura le contenu suivant :

```
1.  {'marié': 'oui', 'enfants': 2, 'salaire': 55555, 'impôt': 2814, 'surcôte': 0, 'décôte': 0, 'réduction': 0,
    'taux': 0.14}
2.  {'marié': 'oui', 'enfants': 2, 'salaire': 50000, 'impôt': 1384, 'surcôte': 0, 'décôte': 384, 'réduction':
    347, 'taux': 0.14}
3.  {'marié': 'oui', 'enfants': 3, 'salaire': 50000, 'impôt': 0, 'surcôte': 0, 'décôte': 720, 'réduction': 0,
    'taux': 0.14}
4.  {'marié': 'non', 'enfants': 2, 'salaire': 100000, 'impôt': 19884, 'surcôte': 4480, 'décôte': 0,
    'réduction': 0, 'taux': 0.41}
5.  {'marié': 'non', 'enfants': 3, 'salaire': 100000, 'impôt': 16782, 'surcôte': 7176, 'décôte': 0,
    'réduction': 0, 'taux': 0.41}
6.  {'marié': 'oui', 'enfants': 3, 'salaire': 100000, 'impôt': 9200, 'surcôte': 2180, 'décôte': 0, 'réduction':
    0, 'taux': 0.3}
7.  {'marié': 'oui', 'enfants': 5, 'salaire': 100000, 'impôt': 4230, 'surcôte': 0, 'décôte': 0, 'réduction': 0,
    'taux': 0.14}
8.  {'marié': 'non', 'enfants': 0, 'salaire': 100000, 'impôt': 22986, 'surcôte': 0, 'décôte': 0, 'réduction':
    0, 'taux': 0.41}
9.  {'marié': 'oui', 'enfants': 2, 'salaire': 30000, 'impôt': 0, 'surcôte': 0, 'décôte': 0, 'réduction': 0,
    'taux': 0}
10. {'marié': 'non', 'enfants': 0, 'salaire': 200000, 'impôt': 64210, 'surcôte': 7498, 'décôte': 0,
    'réduction': 0, 'taux': 0.45}
11. {'marié': 'oui', 'enfants': 3, 'salaire': 200000, 'impôt': 42842, 'surcôte': 17283, 'décôte': 0,
    'réduction': 0, 'taux': 0.41}
```

- ligne 16 : on récupère les données des contribuables contenues dans **[taxpayersdata.txt]**. On récupère une liste de dictionnaires de clés **[marié, enfants, salaire]** chaque dictionnaire représentant un contribuable ;
- lignes 17-25 : on calcule l'impôt des contribuables de la liste **[taxPayers]**. On récupère une liste **[results]** dont chaque élément est de nouveau un dictionnaire de clés **[marié, enfants, salaire, impôt, surcôte, décôte, réduction, taux]** ;
- ligne 27 : la liste **[results]** des résultats est enregistrée dans le fichier **[résultats.txt]** sous la forme montrée ci-dessus ;
- lignes 28-32 : on arrête toutes les exceptions qui peuvent sortir du module **[impots.v01.modules.impôts_module_01]** ;

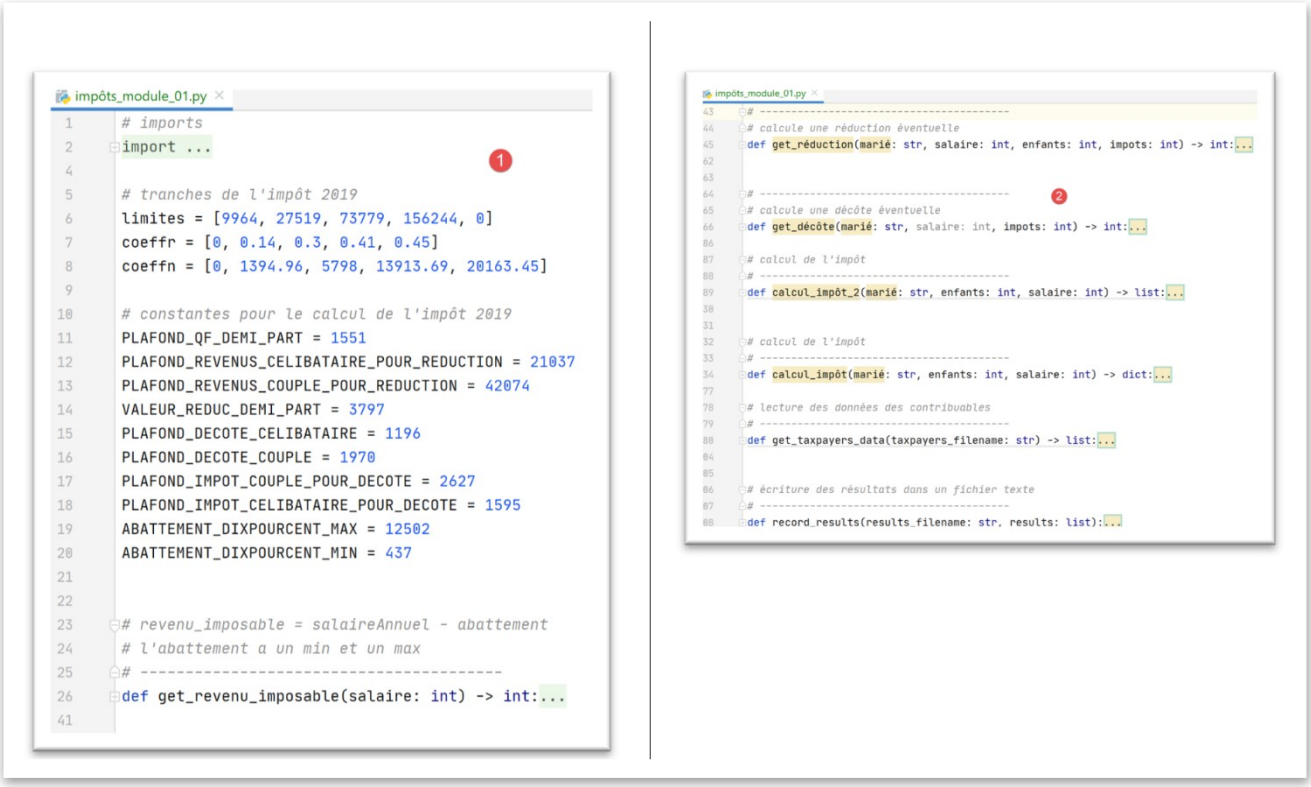
Nous allons détailler maintenant les trois fonctions utilisées par le script **[main]** :

- **[get_taxpayers_data]** : pour lire les données des contribuables ;
- **[calcul_impôt]** : pour calculer l'impôt de ceux-ci ;
- **[record_results]** : pour enregistrer les résultats dans un fichier texte ;

Toutes ces fonctions se trouvent dans le module **[impots.modules.impôts_module_01]**.

8.2.2 Le module [impots.v01.shared.impôts_module_01]

Les fonctions nécessaires au calcul de l'impôt ont été rassemblées dans le module **[impots.v01.shared.impôts_module_01]** :



- en **[1]** : définition des constantes du calcul de l'impôt ;
- en **[2]** : la liste des fonctions du module ;

8.2.3 La fonction [get_taxpayers_data]

La fonction **[get_taxpayers_data]** est la suivante :

```
1.  # imports
2.  import codecs
3.  …
4.
5.  # lecture des données des contribuables
6.  # ----------------------------------------
7.  def get_taxpayers_data(taxpayers_filename: str) -> list:
8.      # lecture des données contribuables
9.      file = None
10.     try:
11.         # la liste des contribuables
12.         taxpayers = []
13.         # ouverture du fichier
14.         file = codecs.open(taxpayers_filename, "r", "utf8")
15.         # on lit la première ligne du fichier des contribuables
16.         ligne = file.readline().strip()
17.         # tant qu'il reste une ligne à exploiter
18.         while ligne != '':
19.             # on récupère les 3 champs marié,enfants,salaire qui forment la ligne
20.             (marié, enfants, salaire) = ligne.split(",")
21.             # on les ajoute à la liste des contribuables
22.             taxpayers.append({'marié': marié.strip().lower(), 'enfants': int(enfants), 'salaire':
    int(salaire)})
```

```
23.             # on lit une nouvelle ligne du fichier des contribuables
24.             ligne = file.readline().strip()
25.         # on rend le résultat
26.         return taxpayers
27.     finally:
28.         # on ferme le fichier s'il a été ouvert
29.         if file:
30.             file.close()
```

Notes

- ligne 7 : **[taxpayers_filename]** est le nom du fichier à exploiter. La fonction rend une liste ;
- lignes 18-24 : la boucle d'exploitation des lignes **[marié, enfants, salaire]** du fichier texte ;
- ligne 20 : les trois éléments de la ligne sont récupérés. On suppose ici que la ligne est syntaxiquement correcte, ç-à-d qu'elle a bien les trois éléments attendus ;
- ligne 22 : on construit un dictionnaire avec les clés **[marié, enfants, salaire]** et ce dictionnaire est ajouté à la liste **[taxPayers]** ;
- ligne 26 : une fois le fichier exploité, on rend la liste **[taxPayers]** ;
- lignes 10-30 : on remarquera qu'on n'a pas mis de clause **[catch]** au **[try]** de la ligne 10. La clause **[catch]** n'est pas obligatoire. Ligne 27, on a mis une clause **[finally]** pour fermer le fichier texte dans tous les cas, erreur ou pas ;
- cette structure try / finally laisse échapper une éventuelle exception (il n'y a pas de catch). Cette exception va remonter au script principal **[main]** qui va arrêter et afficher l'exception (cf paragraphe |Le script principal|). Ce mécanisme a été utilisé pour la plupart des fonctions du module ;

8.2.4 La fonction [calcul_impôt]

La fonction **[calcul_impôt]** est la suivante :

```
1.  # imports
2.  import codecs
3.  import math
4.
5.  # tranches de l'impôt 2019
6.  limites = [9964, 27519, 73779, 156244, 0]
7.  coeffr = [0, 0.14, 0.3, 0.41, 0.45]
8.  coeffn = [0, 1394.96, 5798, 13913.69, 20163.45]
9.
10. # constantes pour le calcul de l'impôt 2019
11. PLAFOND_QF_DEMI_PART = 1551
12. PLAFOND_REVENUS_CELIBATAIRE_POUR_REDUCTION = 21037
13. PLAFOND_REVENUS_COUPLE_POUR_REDUCTION = 42074
14. VALEUR_REDUC_DEMI_PART = 3797
15. PLAFOND_DECOTE_CELIBATAIRE = 1196
16. PLAFOND_DECOTE_COUPLE = 1970
17. PLAFOND_IMPOT_COUPLE_POUR_DECOTE = 2627
18. PLAFOND_IMPOT_CELIBATAIRE_POUR_DECOTE = 1595
19. ABATTEMENT_DIXPOURCENT_MAX = 12502
20. ABATTEMENT_DIXPOURCENT_MIN = 437
21.
22. …
23. # calcul de l'impôt
24. # ----------------------------------------
25. def calcul_impôt(marié: str, enfants: int, salaire: int) -> dict:
26.     # marié : oui, non
27.     # enfants : nombre d'enfants
28.     # salaire : salaire annuel
29.     # limites, coeffr, coeffn : les tableaux des données permettant le calcul de l'impôt
30.     #
31.     # calcul de l'impôt avec enfants
32.     result1 = calcul_impôt_2(marié, enfants, salaire)
33.     impot1 = result1["impôt"]
34.     # calcul de l'impôt sans les enfants
35.     if enfants != 0:
36.         result2 = calcul_impôt_2(marié, 0, salaire)
37.         impot2 = result2["impôt"]
38.         # application du plafonnement du quotient familial
39.         if enfants < 3:
40.             # PLAFOND_QF_DEMI_PART euros pour les 2 premiers enfants
41.             impot2 = impot2 - enfants * PLAFOND_QF_DEMI_PART
42.         else:
43.             # PLAFOND_QF_DEMI_PART euros pour les 2 premiers enfants, le double pour les suivants
44.             impot2 = impot2 - 2 * PLAFOND_QF_DEMI_PART - (enfants - 2) * 2 * PLAFOND_QF_DEMI_PART
45.     else:
46.         impot2 = impot1
```

```
47.         result2 = result1
48.
49.     # on prend l'impôt le plus fort avec le taux et la surcôte qui vont avec
50.     if impot1 > impot2:
51.         impot = impot1
52.         taux = result1["taux"]
53.         surcôte = result1["surcôte"]
54.     else:
55.         surcôte = impot2 - impot1 + result2["surcôte"]
56.         impot = impot2
57.         taux = result2["taux"]
58.
59.     # calcul d'une éventuelle décôte
60.     décôte = get_décôte(marié, salaire, impot)
61.     impot -= décôte
62.     # calcul d'une éventuelle réduction d'impôts
63.     réduction = get_réduction(marié, salaire, enfants, impot)
64.     impot -= réduction
65.     # résultat
66.     return {"marié": marié, "enfants": enfants, "salaire": salaire, "impôt": math.floor(impot), "surcôte":
    surcôte,
67.             "décôte": décôte, "réduction": réduction, "taux": taux}
```

Notes

- lignes 6-8 : les tranches de l'impôt (cf. paragraphe |Calcul de l'impôt brut|) ;
- lignes 11-20 : les constantes du calcul de l'impôt ;
- on notera que les éléments initialisés aux lignes 5-20 seront globaux aux fonctions que nous allons décrire. Ils sont donc connus tant que la fonction qui les utilise ne déclare pas de variables de mêmes noms ;
- les chiffres des lignes 5-20 changent chaque année. Ici ce sont les chiffres 2019 ;
- ligne 25 : la fonction **[calcul_impôt]** reçoit trois paramètres :
  - **[marié]** : oui / non, indique si le contribuable est marié ou pacsé ;
  - **[enfants]** : son nombre d'enfants ;
  - **[salaire]** : son salaire annuel en euros ;
- lignes 31-33 : calcul de l'impôt en prenant en compte les enfants ;
- lignes 34-47 : ces lignes implémentent le plafonnement du quotient familial (cf. paragraphe |Plafonnement du quotient familial|) ;
- lignes 49-57 : ces lignes calculent le taux d'imposition du contribuable ainsi qu'une éventuelle surcote (cf paragraphe |Cas des hauts revenus|) ;
- lignes 59-61 : calcul d'une éventuelle décote (cf paragraphe |Calcul de la décôte|) ;
- lignes 62-64 : calcul d'une éventuelle réduction de l'impôt à payer (cf. paragraphe |Calcul de la réduction d'impôts|) ;

L'algorithme est assez complexe et nous ne le détaillerons pas plus que ce que disent les commentaires. L'algorithme implémente le mode de calcul de l'impôt tel que décrit au paragraphe |Le problème|.

8.2.5 La fonction [calcul_impôt_2]

La fonction **[calcul_impôt]** fait appel à la fonction **[calcul_impôt_2]** suivante :

```
1.  def calcul_impôt_2(marié: str, enfants: int, salaire: int) -> list:
2.      # marié : oui, non
3.      # enfants : nombre d'enfants
4.      # salaire : salaire annuel
5.      # limites, coeffr, coeffn : les tableaux des données permettant le calcul de l'impôt
6.      #
7.      # nombre de parts
8.      marié = marié.strip().lower()
9.      if marié == "oui":
10.         nb_parts = enfants / 2 + 2
11.     else:
12.         nb_parts = enfants / 2 + 1
13.
14.     # 1 part par enfant à partir du 3ième
15.     if enfants >= 3:
16.         # une demi-part de + pour chaque enfant à partir du 3ième
17.         nb_parts += 0.5 * (enfants - 2)
18.
19.     # revenu imposable
20.     revenu_imposable = get_revenu_imposable(salaire)
21.     # surcôte
22.     surcôte = math.floor(revenu_imposable - 0.9 * salaire)
23.     # pour des pbs d'arrondi
```

```
    if surcôte < 0:
        surcôte = 0

    # quotient familial
    quotient = revenu_imposable / nb_parts
    # est mis à la fin du tableau limites pour arrêter la boucle qui suit
    limites[len(limites) - 1] = quotient
    # calcul de l'impôt
    i = 0
    while quotient > limites[i]:
        i += 1
        # du fait qu'on a placé quotient à la fin du tableau limites, la boucle précédente
    # ne peut déborder du tableau limites

    # maintenant on peut calculer l'impôt
    impôt = math.floor(revenu_imposable * coeffr[i] - nb_parts * coeffn[i])
    # résultat
    return {"impôt": impôt, "surcôte": surcôte, "taux": coeffr[i]}
```

Cet algorithme a été décrit au paragraphe 8.1.1.

8.2.6 La fonction `[get_décôte]`

La fonction **`[get_décôte]`** implémente le calcul de l'éventuelle décote de l'impôt (paragraphe |Calcul de la décôte|) :

```
# calcule une décôte éventuelle
def get_décôte(marié: str, salaire: int, impots: int) -> int:
    # au départ, une décôte nulle
    décôte = 0
    # montant maximal d'impôt pour avoir la décôte
    plafond_impôt_pour_décôte = PLAFOND_IMPOT_COUPLE_POUR_DECOTE if marié == "oui" else
PLAFOND_IMPOT_CELIBATAIRE_POUR_DECOTE
    if impots < plafond_impôt_pour_décôte:
        # montant maximal de la décôte
        plafond_décôte = PLAFOND_DECOTE_COUPLE if marié == "oui" else PLAFOND_DECOTE_CELIBATAIRE
        # décôte théorique
        décôte = plafond_décôte - 0.75 * impots
        # la décôte ne peut dépasser le montant de l'ompôt
        if décôte > impots:
            décôte = impots

        # pas de décôte <0
        if décôte < 0:
            décôte = 0

    # résultat
    return math.ceil(décôte)
```

8.2.7 La fonction `[get_réduction]`

La fonction **`[get_réduction]`** implémente le calcul de l'éventuelle réduction de l'impôt à payer (paragraphe |Calcul de la réduction d'impôts|) :

```
# calcule une réduction éventuelle
def get_réduction(marié: str, salaire: int, enfants: int, impots: int) -> int:
    # le plafond des revenus pour avoir droit à la réduction de 20%
    plafond_revenu_pour_réduction = PLAFOND_REVENUS_COUPLE_POUR_REDUCTION if marié == "oui" else
PLAFOND_REVENUS_CELIBATAIRE_POUR_REDUCTION
    plafond_revenu_pour_réduction += enfants * VALEUR_REDUC_DEMI_PART
    if enfants > 2:
        plafond_revenu_pour_réduction += (enfants - 2) * VALEUR_REDUC_DEMI_PART

    # revenu imposable
    revenu_imposable = get_revenu_imposable(salaire)
    # réduction
    réduction = 0
    if revenu_imposable < plafond_revenu_pour_réduction:
        # réduction de 20%
        réduction = 0.2 * impots

    # résultat
    return math.ceil(réduction)
```

8.2.8 La fonction `[get_revenu_imposable]`

La fonction **`[get_revenu_imposable]`** calcule le revenu imposable à partir du salaire annuel :

```
# revenu_imposable = salaireAnnuel - abattement
# l'abattement a un min et un max
# ---------------------------------------
def get_revenu_imposable(salaire: int) -> int:
    # abattement de 10% du salaire
    abattement = 0.1 * salaire
    # cet abattement ne peut dépasser ABATTEMENT_DIXPOURCENT_MAX
    if abattement > ABATTEMENT_DIXPOURCENT_MAX:
        abattement = ABATTEMENT_DIXPOURCENT_MAX

    # l'abattement ne peut être inférieur à ABATTEMENT_DIXPOURCENT_MIN
    if abattement < ABATTEMENT_DIXPOURCENT_MIN:
        abattement = ABATTEMENT_DIXPOURCENT_MIN

    # revenu imposable
    revenu_imposable = salaire - abattement
    # résultat
    return math.floor(revenu_imposable)
```

8.2.9 La fonction `[record_results]`

La fonction **`[record_results]`** enregistre les résultats de calcul de l'impôt dans un fichier texte :

```
# écriture des résultats dans un fichier texte
# ---------------------------------------
def record_results(results_filename: str, results: list):
    # results_filename : le nom du fichier texte où placer les résultats
    # results : la liste des résultats sous la forme d'une liste de dictionnaires
    # chaque dictionnaire est écrit sur une ligne de texte
    résultats = None
    try:
        # ouverture du fichier des résultats
        résultats = codecs.open(results_filename, "w", "utf8")
        # exploitation des contribuables
        for result in results:
            # on inscrit le résultat dans le fichier des résultats
            résultats.write(f"{result}\n")
            # contribuable suivant
    finally:
        # on ferme le fichier s'il a été ouvert
        if résultats:
            résultats.close()
```

8.2.10 Les résultats

Comme il a déjà été dit, avec le fichier des contribuables **`[taxpayersdata.txt]`** suivant :

```
oui,2,55555
oui,2,50000
oui,3,50000
non,2,100000
non,3,100000
oui,3,100000
oui,5,100000
non,0,100000
oui,2,30000
non,0,200000
oui,3,200000
```

le script **`[main.py]`** crée le fichier **`[résultats.txt]`** suivant :

```
{'marié': 'oui', 'enfants': 2, 'salaire': 55555, 'impôt': 2814, 'surcôte': 0, 'décôte': 0, 'réduction': 0, 'taux': 0.14}
{'marié': 'oui', 'enfants': 2, 'salaire': 50000, 'impôt': 1384, 'surcôte': 0, 'décôte': 384, 'réduction': 347, 'taux': 0.14}
{'marié': 'oui', 'enfants': 3, 'salaire': 50000, 'impôt': 0, 'surcôte': 0, 'décôte': 720, 'réduction': 0, 'taux': 0.14}
{'marié': 'non', 'enfants': 2, 'salaire': 100000, 'impôt': 19884, 'surcôte': 4480, 'décôte': 0, 'réduction': 0, 'taux': 0.41}
```

```
5.  {'marié': 'non', 'enfants': 3, 'salaire': 100000, 'impôt': 16782, 'surcôte': 7176, 'décôte': 0,
    'réduction': 0, 'taux': 0.41}
6.  {'marié': 'oui', 'enfants': 3, 'salaire': 100000, 'impôt': 9200, 'surcôte': 2180, 'décôte': 0, 'réduction':
    0, 'taux': 0.3}
7.  {'marié': 'oui', 'enfants': 5, 'salaire': 100000, 'impôt': 4230, 'surcôte': 0, 'décôte': 0, 'réduction': 0,
    'taux': 0.14}
8.  {'marié': 'non', 'enfants': 0, 'salaire': 100000, 'impôt': 22986, 'surcôte': 0, 'décôte': 0, 'réduction':
    0, 'taux': 0.41}
9.  {'marié': 'oui', 'enfants': 2, 'salaire': 30000, 'impôt': 0, 'surcôte': 0, 'décôte': 0, 'réduction': 0,
    'taux': 0}
10. {'marié': 'non', 'enfants': 0, 'salaire': 200000, 'impôt': 64210, 'surcôte': 7498, 'décôte': 0,
    'réduction': 0, 'taux': 0.45}
11. {'marié': 'oui', 'enfants': 3, 'salaire': 200000, 'impôt': 42842, 'surcôte': 17283, 'décôte': 0,
    'réduction': 0, 'taux': 0.41}
```

Ces résultats sont conformes aux chiffres officiels du paragraphe |Chiffres officiels|.

Maintenant, exécutons cette version dans une fenêtre console :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\impots\v01>python main.py
2.  Traceback (most recent call last):
3.    File "main.py", line 4, in <module>
4.      from impots.v01.shared.impôts_module_01 import *
5.  ModuleNotFoundError: No module named 'impots'
```

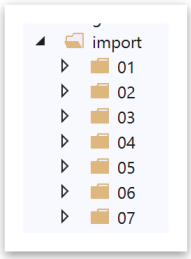
On retrouve une erreur déjà rencontrée : celle où un module n'est pas trouvé, ici le module **[impots]**. On rappelle que cela veut dire que :

- l'interpréteur Python a exploré un à un les dossiers du Python Path ;
- dans aucun d'eux, il n'a trouvé de dossier dans lequel il y aurait un sript **[impots.py]** ;

La version **[v02]** amènera une solution à ce problème.

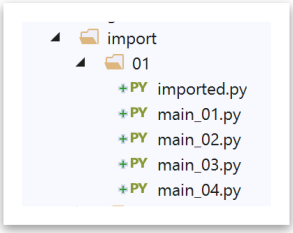
9 Les imports

L'erreur rencontrée dans la version 1 de l'exercice d'application nous amène à approfondir le rôle de l'instruction **[import]**.



9.1 Scripts [import_01]

Le script **[imported]** va être importé par différents scripts (appelés aussi modules) :




```
1.  # module importé
2.  # cette instruction sera exécutée à chaque fois que le module sera importé
3.  print("2")
4.  # variable appartenant au module importé
5.  x=4
```

Un module est **exécuté** lorsqu'il est importé. Ainsi lorsque le module **[imported]** sera importé :

- l'affichage de la ligne 3 aura lieu ;
- la variable x de la ligne 5 recevra sa valeur ;

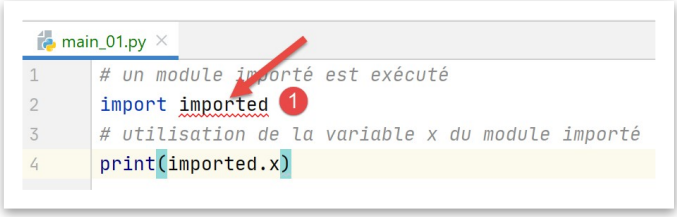
Le script **[main_01]** est le suivant :

```
1.  # un module importé est exécuté
2.  import imported
3.  # utilisation de la variable x du module importé
4.  print(imported.x)
```

- ligne 2 : le module **[imported]** est importé. Cela va provoquer son exécution :
  - la valeur 2 va être affichée ;
  - la variable x est crée avec la valeur 4 ;
- ligne 4 : on utilise la variable x du module importé ;

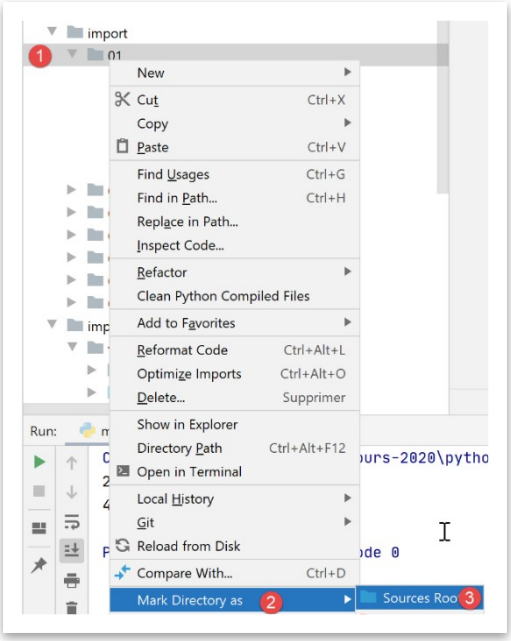
Dans PyCharm, une erreur est signalée :

En **[1]**, PyCharm indique qu'il ne connaît pas le module **[imported]**. En termes techniques, cela signifie que le dossier contenant le

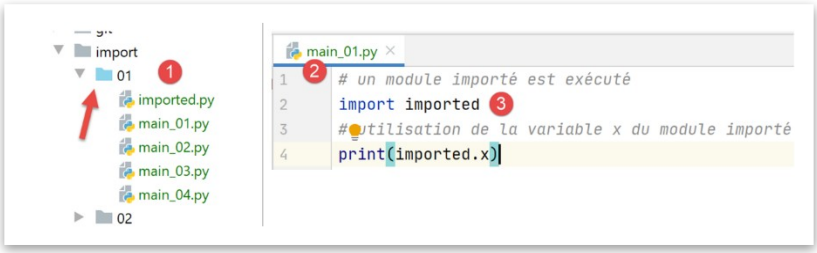


module **[imported]** n'est pas dans le Python Path de PyCharm. Le Python Path est l'ensemble des dossiers dans lesquels les modules

importés sont cherchés. Pour résoudre ce problème, il suffit de déclarer **[Sources root]** le dossier dans lequel se trouve le module **[imported]**, ici le dossier **[import/01]** :



Après cette opération, le dossier **[import/01]** est mis dans le Python Path de Pycharm et l'erreur disparaît :



- en **[1]**, le dossier **[01]** a changé de couleur ;
- en **[2-3]**, il n'y a plus d'erreur ;

Les résultats de l'exécution sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/import/01/main_01.py
2.  2
3.  4
4.
5.  Process finished with exit code 0
```

Commentaires

- la ligne 2 est le résultat de l'exécution du module importé ;
- la ligne 3 affiche la valeur de la variable x du module importé ;

On retiendra de cet exemple, le concept important qu'un module (ou un script) importé est **exécuté**.

Le script **[main_02]** est le suivant :

```
1.  # on importe la variable x du module importé
2.  from imported import x
3.  # on l'affiche
4.  print(x)
```

- ligne 2, on a une autre syntaxe de l'importation **[from module import objet1, objet2, …]**. Ici on importe la variable **[imported.x]**. Avec ectte syntaxe, la variable x devient une variable du script **[main_02]**. On n'a plus besoin de la préfixer par son module **[imported]** ;

- ligne 4 : on affiche la variable x de **[main_02]** ;

Les résultats de l'exécution sont les suivants :

```
C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/import/01/main_02.py
2
4

Process finished with exit code 0
```

Le script **[main_03]** est le suivant :

```
# on importe tout ce qui est visible du module importé
from imported import *
# on utilise la variable x du module importé
print(x)
```

La notation **[import *]** de la ligne 2 signifie qu'on importe tous les objets visibles du module importé (variables, fonctions).

Les résultats sont les suivants :

```
C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/import/01/main_03.py
2
4

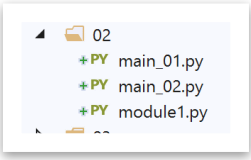
Process finished with exit code 0
```

Le script **[main_04]** est le suivant :

```
# on importe la variable x du module importé
# et on la renomme y
from imported import x as y
# on affiche la variable y
print(y)
```

La ligne 3 montre qu'on peut importer un objet du module importé et lui donner un alias. Ici la variable **[imported.x]** devient la variable **[main_04.y]**. Les résultats sont les mêmes qu'auparavant.

9.2 Script [import_02]



Le module importé **[module1.py]** est ici le suivant :

```
# une fonction
def f1():
    print("f1")
```

Le module importé définit une fonction, un cas fréquent.

Le script **[main_01]** est le suivant :

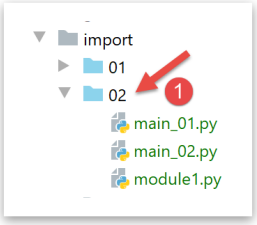
```
# import
import module1
# exécution f1
module1.f1()
```

- ligne 2, le module est importé. Il va être exécuté. Ici, il n'affiche rien ;
- ligne 4 : la fonction **[f1]** du module importé est exécutée ;

Les résultats de l'exécution sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/import/02/main_01.py
2.  f1
3.
4.  Process finished with exit code 0
```

Note : Pour éviter que PyCharm ne signale une erreur sur l'importation de la ligne 2, il mettre le dossier contenant **[module1]** dans les **[Sources Root]** de PyCharm :



En **[1]**, le dossier **[02]** mis dans les **[Sources Root]** est passé en bleu. On notera que l'erreur signalée n'empêche pas ici une exécution correcte des scripts. En effet, lors de l'exécution du script **[main_0x]**, le dossier du script est automatiquement mis dans le Python Path. Du coup **[module1]** est trouvé. Dorénavant, lorsque sur une copie d'écran un dossier est en bleu, c'est qu'il a été mis dans les **[Sources Root]** de PyCharm.

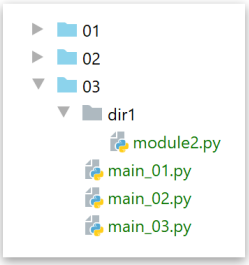
Le script **[main_02]** est le suivant :

```
1.  # import
2.  from module1 import f1
3.  # exécution f1
4.  f1()
```

- la ligne 2 importe la fonction **[f1]** du module **[module1]** ;
- ligne 4, on utilise la fonction f1 ;

Les résultats sont identiques à ceux du script **[main_01]**.

9.3 Scripts [import_03]



Note : **[03]** est dans les **[Sources Root]** du projet.

Les nouveaux scripts vont importer le module **[module2]** qui n'est pas dans le même dossier qu'eux.

Le script **[module2]** est le suivant :

```
1.  # une fonction
2.  def f2():
3.      print("f2")
```

Le script définit donc une fonction **[f2]**.

Le script **[main_01]** est le suivant :

```
1.  # import du module Class2
2.  import dir1.module2
```

```
3.  # exécution f2
4.  dir1.module2.f2()
```

- ligne 2 : on utilise une notation spéciale pour indiquer comment trouver le module **[module2]**. Il faut lire **[dir1.module2]** comme le chemin **[dir1/module2]** : pour trouver **[module2]**, on part du dossier du script courant **[main_01]**, puis on passe dans **[dir1]** et là on trouve **[module2]**. Il ne faut pas oublier ici que le point de départ du chemin est le dossier du script qui importe ;
- ligne 4 : pour exécuter la fonction **[f2]** de **[module2]** ;

Les résultats sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/import/03/main_01.py
2.  f2
3.
4.  Process finished with exit code 0
```

Ligne 2, le résultat de la fonction **[f2]**.

Le script **[main_02]** est le suivant :

```
1.  # import du module dir1.module2 qu'on renomme
2.  import dir1.module2 as module2
3.  # exécution f2
4.  module2.f2()
```

Ligne 2, on renomme le module **[dir1.module]** pour simplifier l'écriture de la ligne 4.

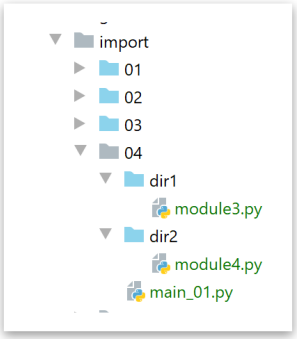
Le script **[main_03]** est le suivant :

```
1.  # importde la fonction f2 du module dir1.module2
2.  from dir1.module2 import f2
3.  # exécution f2
4.  f2()
```

Cette fois-ci, ligne 2, on n'importe que la fonction **[f2]** qui devient alors une fonction du script **[main_03]** (ligne 4).

Tous ces scripts fonctionnent aussi bien dans le contexte PyCharm que dans celui d'une console Python. La raison est que dans les deux cas, le dossier du script exécuté, ici le dossier **[03]** fait partie du Python Path. Du coup, le dossier **[dir1/module2]** est trouvé.

9.4 Scripts [import_04]



Ici, les dossiers **[dir1]** et **[dir2]** ont été mis dans les **[Sources Root]** du projet PyCharm.

Le premier module importé est **[module3]** :

```
1.  # une fonction
2.  def f3():
3.      print("f3")
```

Le second module importé est **[module4]** :

```
1.  from module3 import f3
```

```
2.
3.  # une fonction
4.  def f4():
5.      f3()
6.      print("f4")
```

- ligne 1, on importe la fonction **[f3]** de **[module3]**. Ici, **[module3]** est visible parce qu'on a mis son dossier **[dir1]** dans les **[Sources Root]** ;
- lignes 4-6 : on définit une fonction **[f4]** qui fait appel à la fonction **[f3]** de **[module3]** ;

Le script principal **[main_01]** est le suivant :

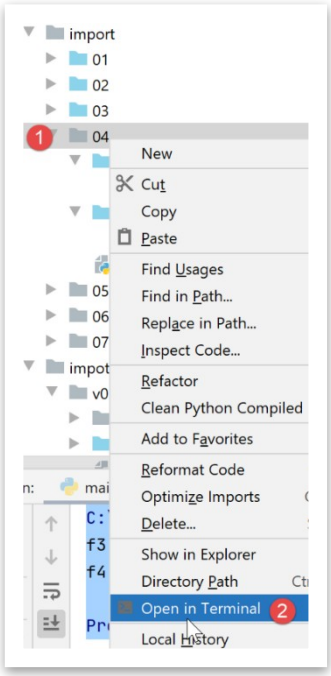
```
1.  # on importe module4
2.  from module4 import f4
3.  # exécution f4
4.  f4()
```

- ligne 2, on importe le module **[module4]**. Celui-ci est visible car on a mis son dossier **[dir2]** dans les **[Sources Root]** de PyCharm ;
- ligne 4 : exécution de la fonction **[f4]** de **[module4]** ;

Les résultats de l'exécution de **[main_01]** dans PyCharm sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/import/04/main_01.py
2.  f3
3.  f4
4.
5.  Process finished with exit code 0
```

Maintenant, exécutons **[main_01]** dans un terminal (console) Python :



Les résultats sont les suivants :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\import\04>python main_01.py
2.  Traceback (most recent call last):
3.    File "main_01.py", line 2, in <module>
4.      from module4 import f4
5.  ModuleNotFoundError: No module named 'module4'
```

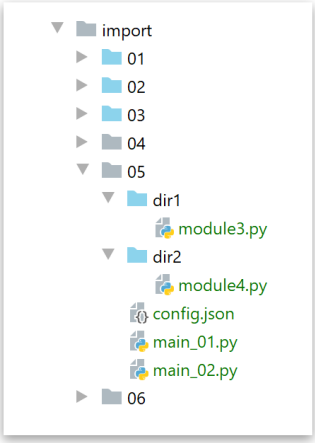
Que s'est-il passé ? Le terminal Python n'a aucune connaissance du Python Path et des **[Sources Root]** de PyCharm. Il a son propre Python Path. Dans celui-ci, on a toujours le dossier du script qui s'exécute, ici le script **[main_01]**. Il connaît donc le dossier **[import/04]**. Dans le script exécuté, il trouve la ligne :

```
1.  from module4 import f4
```

L'interpréteur Python cherche **[module4]** dans les dossiers de son Python Path. Or **[module4]** ne se trouve pas dans **[import/04]** qui se trouve bien dans le Python Path mais dans **[import/04/dir2]** qui ne s'y trouve pas. D'où l'erreur.

On a donc un problème déjà rencontré : un script s'exécutant correctement dans PyCharm peut planter dans le contexte d'un terminal Python. C'est un problème récurrent qu'il va nous falloir résoudre.

9.5 Scripts [import_05]



Note : les dossiers **[dir1]** et **[dir2]** sont mis dans le Python Path. Remarquons déjà qu'il y a là un conflit : **[module3]** et **[module4]** seront trouvés à deux endroits du Python Path de PyCharm :

- dans **[import/04/dir1]** et **[import/05/dir1]** pour **[module3]** ;
- dans **[import/04/dir2]** et **[import/05/dir2]** pour **[module4]** ;

On peut alors sortir **[import/04/dir1]** et **[import/04/dir2]** des **[Sources Root]** du projet PyCharm. Il se trouve qu'ici, **[import/05/dir1]** est une copie de **[import/04/dir1]** (idem pour **[dir2]**) et qu'il n'y a donc pas de problème. Mais notons néanmoins qu'à l'intérieur même de PyCharm, on doit faire attention à la liste des dossiers des **[Sources Root]** afin d'éviter les conflits.

Le script **[main_01]** devient le suivant :

```
1.  import sys
2.  # on modifie le sys.path pour y inclure les dossiers
3.  # contenant les classes à importer
4.  sys.path.append(".")
5.  sys.path.append("./dir1")
6.  sys.path.append("./dir2")
7.  # on importe module4
8.  from module4 import f4
9.  # exécution f4
10. f4()
```

On cherche à résoudre le problème du Python Path. On en veut un qui fonctionne aussi bien sous PyCharm que dans un terminal Python. Pour cela, on va le fixer nous-mêmes.

- lignes 4-6 : on ajoute les dossiers **[., ./dir1, ./dir2]** dans le Python Path. Pour que cela marche, il faut que le dossier courant au moment de l'exécution soit le dossier **[import/05]**. Ce sera vrai dans PyCharm mais pas forcément vrai dans un terminal Python comme nous le verrons ;
- ligne 8 : on importe **[module4]**. Suite à ce que nous venons de faire, il devrait être trouvé dans **[./dir2]** ;

L'exécution dans PyCharm donne les résultats suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/import/05/main_01.py
2.  f3
3.  f4
4.
5.  Process finished with exit code 0
```

Maintenant, dans un terminal Python :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\import\05>python main_01.py
2.  f3
3.  f4
```

Ligne 1, le dossier d'exécution est **[import/05]**.

Maintenant remontons d'un niveau dans l'arborescence de **[import/05]** :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\import\05>cd ..
2.
3.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\import>python 05/main_01.py
4.  Traceback (most recent call last):
5.    File "05/main_01.py", line 8, in <module>
6.      from module4 import f4
7.  ModuleNotFoundError: No module named 'module4'
```

- ligne 2, lorsque **[main_01]** est exécuté on n'est plus dans le dossier **[import/05]** mais dans **[import]**. Or nous avons écrit :

```
1.  sys.path.append(".")
2.  sys.path.append("./dir1")
3.  sys.path.append("./dir2")
```

Cela ajoute au Python Path les dossiers **[import, import/dir1, import/dir2]**, pas du tout ce qu'on veut. Notons qu'ajouter au Python Path des dossiers qui n'existent pas (import/dir1, import/dir2) ne provoque pas d'erreurs.

On a progressé mais ce n'est pas suffisant. Il faut ajouter au Python Path, non pas des chemins relatifs, mais des chemins absolus.

Le script **[main_02]** est une variante de **[main_01]** qui utilise un fichier de configuration **[config.json]** :

```
1.  {
2.    "dependencies": [
3.      "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/import/05/dir1",
4.      "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/import/05/dir2"
5.    ]
6.  }
```

La valeur de la clé **[dependencies]** est la liste des dossiers à ajouter au Python Path. A noter qu'ici on a mis des noms absolus et non des noms relatifs.

Le script **[main_02]** utilise le fichier **[config.json]** de la façon suivante :

```
1.  import codecs
2.  import json
3.  import sys
4.
5.  # fichier de configuration
6.  config_filename="C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/import/05/config.json"
7.  # lecture du fichier de configuration json
8.  with codecs.open(config_filename, "r", "utf-8") as file:
9.      config = json.load(file)
10.
11. # modification du sys.path
12. for directory in config['dependencies']:
13.     sys.path.append(directory)
14.
15. # on importe module4
16. from module4 import f4
17. # exécution f4
18. f4()
19.
```

- ligne 6 : notez qu'on a utilisé le nom absolu du fichier de configuration ;
- lignes 8-9 : le fichier de configuration est lu. Un dictionnaire **[config]** (ligne 9) est construit avec son contenu ;
- lignes 11-13 : on ajoute les éléments du tableau **[config['dependencies']]** au Python Path. Notons que puisqu'on a mis des noms absolus de dossiers dans **[config.json]**, on ajoute dans le Python Path des noms absolus ;
- ligne 16 : **[module4]** est importé. On devrait le trouver puisque maintenant **[dir2]** est dans le Python Path ;

L'exécution donne les mêmes résultats que pour **[main_02]** sauf que le script continue à fonctionner lorsque le dossier d'exécution n'est plus **[import/05]** :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\import>python 05/main_02.py
2.  f3
3.  f4
```

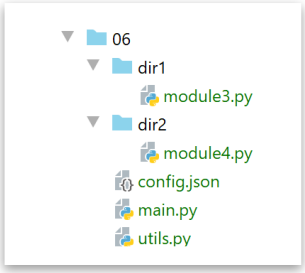
Ligne 1, le dossier d'exécution est **[import]**.

Nous avons progressé. Nous avons vu :

- qu'il nous fallait construire le Python Path nous-mêmes ;
- qu'il fallait y mettre les noms absolus de tous les dossiers contenant les modules importés par l'application ;

Néanmoins, mettre des noms absolus dans des scripts n'est pas une solution. Dès que le projet est transporté sur un autre site, il ne marche plus. Il nous faut trouver autre chose.

9.6 Scripts [import_06]



Note : les dossiers **[06, dir1, dir2]** ont été placés dans les **[Sources Root]** du projet PyCharm. Les dossiers **[dir1, dir2]** sont identiques à ceux des exemples précédents.

Le fichier **[config.json]** est le suivant :

```
1.  {
2.    "rootDir": "C:/Data/st-2020/dev/python/cours-2020/v-02/imports/06",
3.    "relativeDependencies": [
4.      "dir1",
5.      "dir2"
6.    ],
7.    "absoluteDependencies": [
8.     ]
9.  }
```

Nous introduisons deux types de chemins :

- des chemins absolus, lignes 7-8 ;
- des chemins relatifs, lignes 3-6. Ils sont relatifs à la racine de la ligne 2. Ainsi lorsque le projet bouge d'emplacement, seule cette ligne doit être modifiée ;

Le script **[utils.py]** exploite le fichier **[config.json]** et construit le Python Path :

```
1.  # imports
2.  import codecs
3.  import json
4.  import os
5.  import sys
6.
7.  # configuration de l'application
8.  def config_app(config_filename: str) -> dict:
9.      # config_filename : nom du fichier de configuration
10.     # on laisse remonter les exceptions
11.
12.     # exploitation du fichier de configuration
13.     with codecs.open(config_filename, "r", "utf-8") as file:
14.         config = json.load(file)
15.     # ajout des dépendances au sys.path
16.     rootDir = config['rootDir']
17.     # on ajoute les dépendances relatives du projet au syspath
```

```
18.     for directory in config['relativeDependencies']:
19.         # on ajoute la dépendance au début du syspath
20.         sys.path.insert(0, f"{rootDir}/{directory}")
21.     # on lui ajoute les dépendances absolues du projet au syspath
22.     for directory in config['absoluteDependencies']:
23.         # on ajoute la dépendance au début du syspath
24.         sys.path.insert(0, directory)
25.     # on rend le dictionnaire de la configuration
26.     return config
27.
28. # dossier du sript exécuté
29. def get_scriptdir():
30.     return os.path.dirname(os.path.abspath(__file__))
```

- ligne 8 : la fonction **[config_app]** reçoit en paramètre le nom du fichier de configuration ;
- lignes 12-14 : le fichier de configuration est exploité pour créer le dictionnaire **[config]** ;
- ligne 20 : **[sys.path]** est la liste des dossiers du Python Path ;
- lignes 17-20 : les dépendances relatives du fichier de configuration sont ajoutées au Python Path. Elles sont ajoutées au début du tableau **[sys.path]**, ligne 20. En effet, lorsque Python cherche un module il explore les dossiers du **[sys.path]** dans l'ordre. Or dans ce document, des modules de mêmes noms vont se trouver dans des dossiers différents du **[sys.path]**. En mettant les dépendances de l'application au début du tableau **[sys.path]**, on s'assure que celles-ci seront explorées avant d'autres dossiers du **[sys.path]** qui pourraient contenir des modules de mêmes noms ;
- lignes 21-24 : les dépendances absolues du fichier de configuration sont ajoutées au Python Path ;
- ligne 26 : on rend la configuration de l'application ;
- lignes 29-30 : la fonction **[get_scriptdir]** rend le nom absolu du dossier du script en cours d'exécution (celui où se trouve l'appel à la fonction) ;

Le script principal **[main]** est le suivant :

```
1.  # imports
2.  import sys
3.
4.  from utils import config_app
5.
6.
7.  def affiche_path(msg: str):
8.      # message
9.      print(f"{msg}------------------------------")
10.     # sys.path
11.     for path in sys.path:
12.         print(path)
13.
14.
15. # main -------------
16. try:
17.     # le sys.path est configuré
18.     affiche_path("avant....")
19.     config = config_app(f"{get_scriptdir()}/config.json")
20.     affiche_path("après....")
21.     # on importe module4
22.     from module4 import f4
23.     # exécution f4
24.     f4()
25. except BaseException as erreur:
26.     print(f"L'erreur suivante s'est produite : {erreur}")
27. finally:
28.     print("done")
```

- ligne 4 : la fonction **[config_app]** est importée. On notera que puisque **[utils]** et **[main]** sont dans le même dossier, cet **[import]** fonctionne tout le temps. En effet, le dossier du script principal est automatiquement ajouté au python Path ;
- lignes 7-12 : la fonction **[affiche_path]** affiche la liste des dossiers du Python Path ;
- ligne 19 : on configure l'application. Noter qu'on passe à la fonction **[config_app]** le nom absolu du fichier de configuration. Après cette instruction, le Python Path a été reconstruit ;
- ligne 22 : on importe **[module4]**. Grâce à la reconstruction du Python Path, ce module va être trouvé ;
- ligne 24 : on exécute la fonction **[f4]** ;

Dans le contexte PyCharm, les résultats de l'exécution sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/import/06/main.py
2.  avant....------------------------------
3.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\import\06
```

```
4.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020
5.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\fonctions\shared
6.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\impots\v01\shared
7.  …
8.  C:\Program Files\Python38\python38.zip
9.  C:\Program Files\Python38\DLLs
10. C:\Program Files\Python38\lib
11. C:\Program Files\Python38
12. C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv
13. C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\lib\site-packages
14. après....------------------------------
15. C:/Data/st-2020/dev/python/cours-2020/v-02/imports/06/dir2
16. C:/Data/st-2020/dev/python/cours-2020/v-02/imports/06/dir1
17. C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\import\06
18. C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020
19. C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\fonctions\shared
20. C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\impots\v01\shared
21. ….
22. C:\Program Files\Python38\python38.zip
23. C:\Program Files\Python38\DLLs
24. C:\Program Files\Python38\lib
25. C:\Program Files\Python38
26. C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv
27. C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\lib\site-packages
28. f3
29. f4
30. done
31.
32. Process finished with exit code 0
```

Commentaires

- lignes 2-13 : le Python Path de PyCharm. On y retrouve tous les dossiers mis dans les **[Sources Root]** du projet ;
- lignes 14-29 : le Python Path construit par la fonction **[config_app]**. On y trouve aux lignes 15-16, les deux dépendances que nous avons ajoutées ;
- lignes 22-27 : les dossiers système de l'interpréteur Python qui a exécuté le script ;
- lignes 28-29 : l'exécution se passe normalement ;

Maintenant, plaçons-nous dans le contexte qui jusqu'à maintenant provoquait une erreur d'exécution :

- on se place dans un terminal Python ;
- on se met dans un dossier autre que celui contenant le script exécuté ;

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\import>python 06/main.py
2.  avant....------------------------------
3.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\import\06
4.  C:\Program Files\Python38\python38.zip
5.  C:\Program Files\Python38\DLLs
6.  C:\Program Files\Python38\lib
7.  C:\Program Files\Python38
8.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv
9.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\lib\site-packages
10. après....------------------------------
11. C:/Data/st-2020/dev/python/cours-2020/v-02/imports/06/dir2
12. C:/Data/st-2020/dev/python/cours-2020/v-02/imports/06/dir1
13. C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\import\06
14. C:\Program Files\Python38\python38.zip
15. C:\Program Files\Python38\DLLs
16. C:\Program Files\Python38\lib
17. C:\Program Files\Python38
18. C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv
19. C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\lib\site-packages
20. f3
21. f4
22. done
```

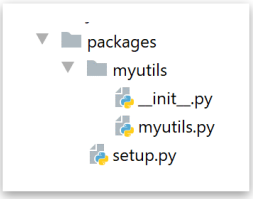
Cette fois-ci c'est bon (lignes 20-21). On notera que le **[sys.path]** ne contient pas les mêmes dossiers que lorsque l'exécution se passe sous PyCharm.

9.7 Scripts [import_07]

Nous améliorons la solution précédente de deux façons :

- nous remplaçons le fichier de configuration **[config.json]** par un script **[config.py]**. En effet, le fichier jSON pose un problème important : il ne peut pas être commenté. Le dictionnaire **[config.json]** peut être remplacé par un dictionnaire Python qui a l'avantage de pouvoir être commenté ;
- nous utilisons un module visible de tous les projets Python de la machine ;

9.7.1 Installation d'un module de portée machine



Ci-dessus, nous créons un dossier **[packages/myutils]** dans le projet PyCharm (les noms n'importent pas).

Le script **[myutils.py]** est le suivant :

```
1.  # imports
2.  import sys
3.  import os
4.
5.
6.  def set_syspath(absolute_dependencies: list):
7.      # absolute_dependencies : une liste de noms absolus de dossiers
8.
9.      # on ajoute au syspath les dépendances absolues du projet
10.     for directory in absolute_dependencies:
11.         # on vérifie l'existence du dossier
12.         existe = os.path.exists(directory) and os.path.isdir(directory)
13.         if not existe:
14.             # on lève une exception
15.             raise BaseException(f"[set_syspath] le dossier du Python Path [{directory}] n'existe pas")
16.         else:
17.             # on joute le dossier au début du syspath
18.             sys.path.insert(0, directory)
```

- lignes 6-18 : la fonction **[set_syspath]** crée un Python Path avec la liste des dossiers qu'on lui transmet en paramètre ;
- lignes 12-15 : on vérifie que le dossier à mettre dans le Python Path existe ;

Le script **[__init.py__]** (deux soulignés devant et derrière le nom. Celui-ci est imposé) est le suivant :

```
1.  from .myutils import set_syspath
```

On importe la fonction **[set_syspath]** du script **[myutils]**. La notation **[.myutils]** désigne le chemin **[./myutils]**, donc le script **[myutils]** se trouvant dans le même dossier que **[__init.py]**. On aurait pu utiliser la notation **[myutils]**. Seulement, nous allons créer un module **[myutils]** de portée machine. Si bien que la notation **[from myutils import set_syspath]** deviendrait alors ambigüe. S'agit-il d'importer le script **[myutils]** du dossier courant ou le script **[myutils]** de portée machine ? La notation **[.myutils]** lève cette ambiguïté.

Le script **[setup.py]** (là également le nom est imposé) est le suivant :

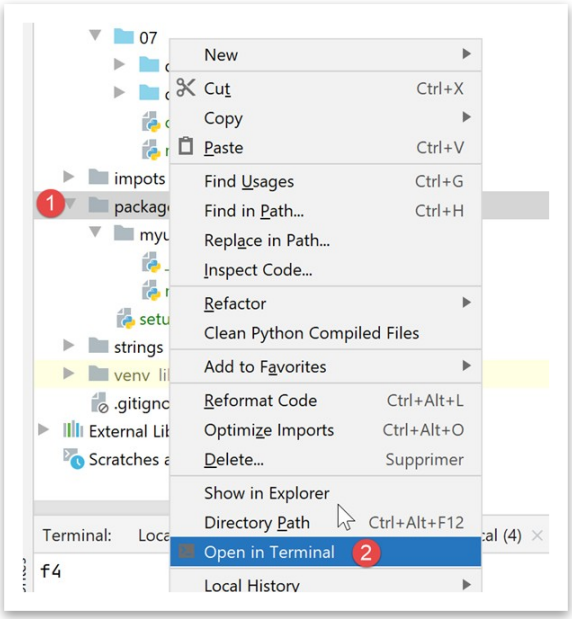
```
1.  from setuptools import setup
2.
3.  setup(name='myutils',
4.        version='0.1',
5.        description='Utilitaire fixant le Python Path',
6.        url='#',
7.        author='st',
8.        author_email='st@gmail.com',
9.        license='MIT',
10.       packages=['myutils'],
11.       zip_safe=False)
```

Dans ce script, on décrit le module qu'on va créer. Ici, nous allons le créer localement. Mais le même processus est utilisé pour créer un module distribué officiellement (cf. |pypi|). Les points importants sont ici les suivants :

- ligne 3 : le nom du module créé ;

- ligne 4 : la version du module ;
- ligne 5 : sa description ;
- lignes 7-8 : l'auteur du module ;

Pour installer ce module avec une portée machine, on procède de la façon suivante :

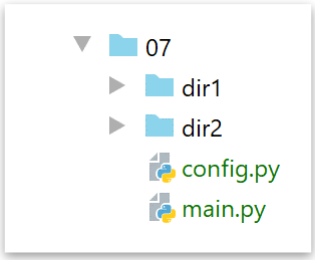


Puis dans le terminal Python, on tape la commande suivante **[pip install .]** :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\packages>pip install .
2.  Processing c:\data\st-2020\dev\python\cours-2020\python3-flask-2020\packages
3.  Using legacy setup.py install for myutils, since package 'wheel' is not installed.
4.  Installing collected packages: myutils
5.    Attempting uninstall: myutils
6.      Found existing installation: myutils 0.1
7.      Uninstalling myutils-0.1:
8.        Successfully uninstalled myutils-0.1
9.      Running setup.py install for myutils ... done
10. Successfully installed myutils-0.1
```

A partir de maintenant, tout script de la machine peut importer le module **[myutils]** sans que celui-ci soit dans les codes du projet.

9.7.2 Le script [config.py]



Le script **[config.py]** assure la configuration de l'application :

```
1.  def configure():
2.      import os
3.
4.      # nom absolu du dossier du script de configuration
5.      script_dir = os.path.dirname(os.path.abspath(__file__))
6.      # chemins absolus des dossiers à mettre dans le syspath
7.      absolute_dependencies = [
8.          # dossiers locaux
9.          f"{script_dir}/dir1",
10.         f"{script_dir}/dir2",
11.     ]
```

```
12.     # mise à jour du syspath
13.     from myutils import set_syspath
14.     set_syspath(absolute_dependencies)
15.
16.     # on retourne la config
17.     return {}
```

- ligne 1 : la fonction **[configure]** assure la configuration de l'application ;
- lignes 7-10 : le dictionnaire qui était auparavant dans **[config.json]** ;
- lignes 9-10 : parce qu'on est dans un script, on peut avoir directement les noms absolus des dossiers **[dir1, dir2]** ;
- lignes 12-14 : on utilise la fonction **[set_syspath]** du module **[myutils]** que nous venons de créer pour définir le Python Path de la configuration ;
- ligne 20 : on rend le dictionnaire de la configuration de l'application. Ici, il est vide ;

9.7.3 Le script [main.py]

Le script principal **[main]** est le suivant :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from module4 import f4
8.
9.  # main -------------
10. try:
11.     f4()
12. except BaseException as erreur:
13.     print(f"L'erreur suivante s'est produite : {erreur}")
14. finally:
15.     print("done")
```

- lignes 2-4 : on configure l'application à l'aide du module **[config.py]**. Celui-ci est accessible car il est dans le même dossier que le script principal. Or le dossier du script principal fait toujours partie du Python Path ;
- lorsqu'on arrive à la ligne 6, le Python Path a été construit avec dedans le dossier du module **[module4]**. On peut donc importer celui-ci ligne 7 ;
- lignes 10-15 : il ne reste plus qu'à exécuter la fonction **[f4]** ;

Les résultats de l'exécution dans PyCharm sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/import/07/main.py
2.  f3
3.  f4
4.  done
5.
6.  Process finished with exit code 0
```

Dans un terminal Python et ailleurs que dans le dossier du script principal, les résultats sont les suivants :

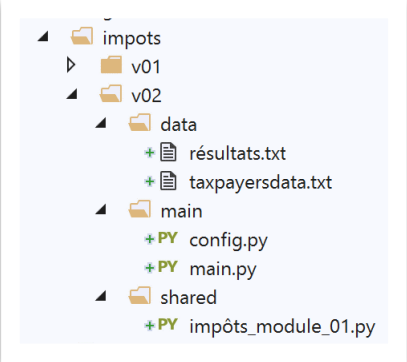
```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\import>python 07/main.py
2.  f3
3.  f4
4.  done
```

Désormais nous procèderons toujours de la même façon pour configurer une application :

- présence d'un script **[config.py]** dans le dossier du script principal. Ce script contient une fonction **[configure]** qui a deux objectifs :
  - construire le Python Path pour l'application. Pour cela, **[config.py]** déclare tous les dossiers contenant les modules utilisés par l'application et construit le Python Path avec leurs noms absolus ;
  - construire le dictionnaire **[config]** de la configuration de l'application ;

Nous appliquons ce schéma à la seconde version de l'exercice d'application. On se souvient en effet que la version 1 fonctionnait dans l'environnement PyCharm mais pas dans un terminal Python. Le problème venait du Python Path.

10 Exercice d'application : version 2



Cette nouvelle version introduit le fichier **[config.py]** suivant :

```
1.  def configure():
2.      import os
3.
4.      # chemin absolu du dossier de ce script
5.      script_dir = os.path.dirname(os.path.abspath(__file__))
6.
7.      # racine à partir de laquelle vont être mesurés certains chemins relatifs
8.      root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots"
9.
10.     # dépendances de l'application
11.     absolute_dependencies = [
12.         f"{root_dir}/v01/shared",
13.     ]
14.
15.     # configuration de l'application
16.     config = {
17.         # chemin absolu du fichier des contribuables
18.         "taxpayersFilename": f"{script_dir}/../data/taxpayersdata.txt",
19.         # chemins absolu du fichier des résultats
20.         "resultsFilename": f"{script_dir}/../data/résultats.txt"
21.     }
22.
23.     # mise à jour du syspath
24.     from myutils import set_syspath
25.     set_syspath(absolute_dependencies)
26.
27.     # on rend la config
28.     return config
```

Commentaires

- ligne 5 : on récupère le nom absolu du dossier qui contient le script qui s'exécute, ici le script **[config.py]**. On obtient donc le nom absolu du dossier **[main]**. C'est également ce dossier qui contient le script principal **[main.py]** ;
- ligne 8 : lorsqu'un fichier référencé n'appartient pas au dossier de l'application, on n'utilisera pas **[script_dir]** pour le localiser mais **[root_dir]**. Cette ligne devra être changée dès que l'application change de place dans le système de fichiers ;
- lignes 11-13 : on liste les noms absolus de tous les dossiers qui doivent se trouver dans le Python Path pour que l'application fonctionne. Ligne 12, on référence le dossier **[shared]** de la version 1 de l'exercice d'application ;
- lignes 16-21 : définissent la configuration de l'application dans un dictionnaire **[config]**. Ici on inscrit les chemins absolus des fichiers texte manipulés par l'application. Pour cela, on utilise **[script_dir]** qui rappelons-le, désigne ici le dossier **[main]** ;
- lignes 24-25 : on fixe le Python Path nécessaire à l'application ;

Le script principal **[main.py]** est le suivant :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from impôts_module_01 import calcul_impôt, record_results, get_taxpayers_data
8.
9.  # fichier des contribuables
```

```
10. taxpayers_filename = config['taxpayersFilename']
11. # fichier des résultats
12. results_filename = config['resultsFilename']
13.
14. # code
15. try:
16.
17.     # lecture des données contribuables
18.     taxpayers = get_taxpayers_data(taxpayers_filename)
19.     # liste des résultats
20.     results = []
21.     # on calcule l'impôt des contribuables
22.     for taxpayer in taxpayers:
23.         # le calcul de l'impôt renvoie un dictionnaire de clés
24.         # ['marié', 'enfants', 'salaire', 'impôt', 'surcôte', 'décôte', 'réduction', 'taux']
25.         result = calcul_impôt(taxpayer['marié'], taxpayer['enfants'], taxpayer['salaire'])
26.         # le dictionnaire est ajouté à la liste des résultats
27.         results.append(result)
28.     # on enregistre les résultats
29.     record_results(results_filename, results)
30. except BaseException as erreur:
31.     # il peut y avoir différentes erreurs : absence de fichier, contenu de fichier incorrect
32.     # on affiche l'erreur et on quitte l'application
33.     print(f"L'erreur suivante s'est produite : {erreur}]\n")
34. finally:
35.     print("Travail terminé...")
```

Commentaires

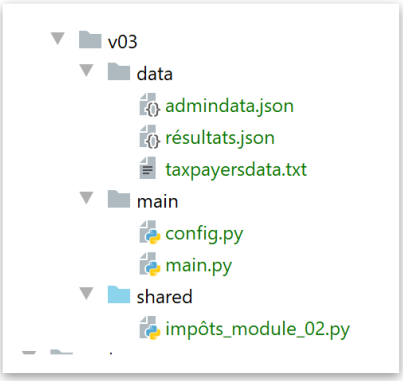
- lignes 1-4 : l'application est configurée ;
- ligne 7 : on sait qu'après la configuration, le Python Path est correct et contient notamment le dossier **[shared]** qui contient le script **[impôts_module_01]**. De ce script, on importe les fonctions dont on a besoin ;
- lignes 9-12 : les noms des fichiers utilisés sont trouvés dans la configuration. Ce sont des noms absolus ;
- lignes 14-35 : on retrouve le code de la version 1 ;

La version 1 ne marchait pas dans une console Python. Dans cette même console, la version 2 donne les résultats suivants :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\impots\v02>python main/main.py
2.  Travail terminé...
```

Aucune erreur n'est désormais rencontrée.

11 Exercice d'application : version 3



Cette nouvelle version introduit deux changements :

- les données nécessaires au calcul de l'impôt et fournies par l'administration fiscale sont placées dans un fichier jSON **[admindata.json]**:

```
{
    "limites": [9964, 27519, 73779, 156244, 0],
    "coeffR": [0, 0.14, 0.3, 0.41, 0.45],
    "coeffN": [0, 1394.96, 5798, 13913.69, 20163.45],
    "PLAFOND_QF_DEMI_PART": 1551,
    "PLAFOND_REVENUS_CELIBATAIRE_POUR_REDUCTION": 21037,
    "PLAFOND_REVENUS_COUPLE_POUR_REDUCTION": 42074,
    "VALEUR_REDUC_DEMI_PART": 3797,
    "PLAFOND_DECOTE_CELIBATAIRE": 1196,
    "PLAFOND_DECOTE_COUPLE": 1970,
    "PLAFOND_IMPOT_COUPLE_POUR_DECOTE": 2627,
    "PLAFOND_IMPOT_CELIBATAIRE_POUR_DECOTE": 1595,
    "ABATTEMENT_DIXPOURCENT_MAX": 12502,
    "ABATTEMENT_DIXPOURCENT_MIN": 437
}
```

- les résultats du calcul de l'impôt seront eux également placés dans un fichier jSON **[résultats.json]** :

```
[
  {
    "marié": "oui",
    "enfants": 2,
    "salaire": 55555,
    "impôt": 2814,
    "surcôte": 0,
    "décôte": 0,
    "réduction": 0,
    "taux": 0.14
  },
  {
    "marié": "oui",
    "enfants": 2,
    "salaire": 50000,
    "impôt": 1384,
    "surcôte": 0,
    "décôte": 384,
    "réduction": 347,
    "taux": 0.14
  },
  …
  {
    "marié": "oui",
    "enfants": 3,
    "salaire": 200000,
    "impôt": 42842,
    "surcôte": 17283,
    "décôte": 0,
    "réduction": 0,
    "taux": 0.41
  }
```

```
33. ]
```

11.1 Le script de configuration [config.py]

Le script de configuration sera le suivant :

```
1.  def configure():
2.      import os
3.
4.      # chemin absolu du dossier de ce script
5.      script_dir = os.path.dirname(os.path.abspath(__file__))
6.      # dépendances de l'application
7.      absolute_dependencies = [
8.          f"{script_dir}/../shared",
9.      ]
10.     # configuration de l'application
11.     config = {
12.         # chemin absolu du fichier des contribuables
13.         "taxpayersFilename": f"{script_dir}/../data/taxpayersdata.txt",
14.         # chemin absolu du fichier des résultats
15.         "resultsFilename": f"{script_dir}/../data/résultats.json",
16.         # chemin absolu du fichier des données de l'administration fiscale
17.         "admindataFilename": f"{script_dir}/../data/admindata.json"
18.     }
19.     # mise à jour du syspath
20.     from myutils import set_syspath
21.
22.     set_syspath(absolute_dependencies)
23.
24.     # on rend la config
25.     return config
```

- ligne 8 : on met le dossier **[shared]** dans le Python Path. Ce dossier contient le module **[impôts_module_02]** utilisé par le script principal ;

11.2 Script principal [main.py]

Le script principal de la version 3 est le suivant :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from impôts_module_02 import calcul_impôt, get_admindata, get_taxpayers_data, record_results_in_json_file
8.
9.  # fichier des contribuables
10. taxpayers_filename = config['taxpayersFilename']
11. # fichier des résultats
12. results_filename = config['resultsFilename']
13. # fichier des données de l'administration fiscale
14. admindata_filename = config['admindataFilename']
15. # code
16. try:
17.
18.     # lecture des données de l'administration fiscale
19.     admindata = get_admindata(admindata_filename)
20.     # lecture des données contribuables
21.     taxpayers = get_taxpayers_data(taxpayers_filename)
22.     # liste des résultats
23.     results = []
24.     # on calcule l'impôt des contribuables
25.     for taxpayer in taxpayers:
26.         # le calcul de l'impôt renvoie un dictionnaire de clés
27.         # ['marié', 'enfants', 'salaire', 'impôt', 'surcôte', 'décôte', 'réduction', 'taux']
28.         result = calcul_impôt(admindata, taxpayer['marié'], taxpayer['enfants'], taxpayer['salaire'])
29.         # le dictionnaire est ajouté à la liste des résultats
30.         results.append(result)
31.     # on enregistre les résultats
32.     record_results_in_json_file(results_filename, results)
33. except BaseException as erreur:
34.     # il peut y avoir différentes erreurs : absence de fichier, contenu de fichier incorrect
35.     # on affiche l'erreur et on quitte l'application
```

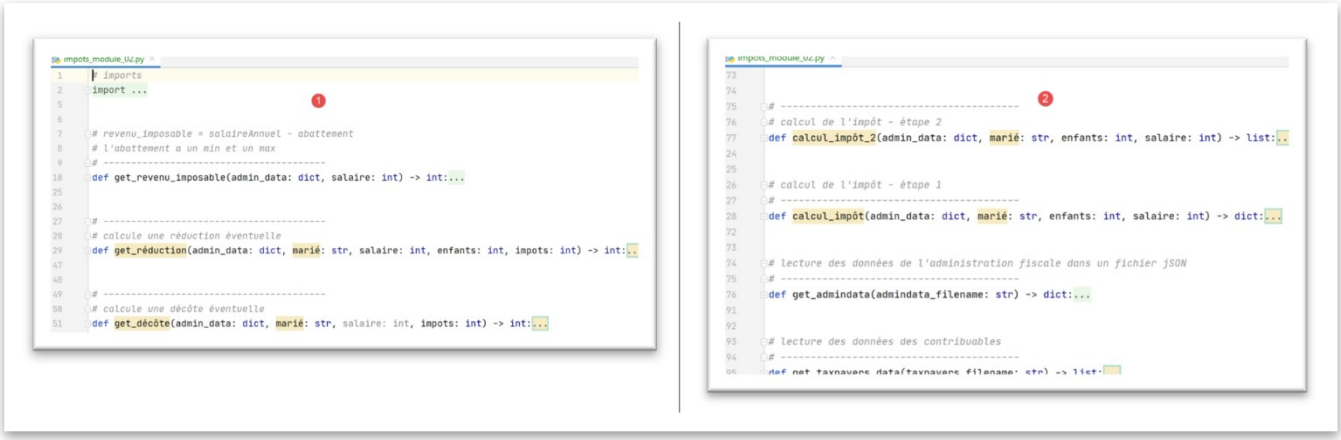
```
36.     print(f"L'erreur suivante s'est produite : {erreur}]\n")
37. finally:
38.     print("Travail terminé...")
```

Notes

- lignes 2-4, on configure l'application notamment son Python Path ;
- ligne 7 : on importe les fonctions dont on a besoin dans **[main.py]** ;
- lignes 9-14 : les noms des fichiers exploités par l'application sont récupérés dans la configuration ;
- le script principal de la version 3 présente trois différences vis à vis de celui des versions 1 et 2 :
  - ligne 21 : les données de l'administration fiscale sont prises dans le fichier jSON **[./data/admindata.json]** ;
  - ligne 32 : les résultats du calcul de l'impôt sont placés dans le fichier jSON **[./data/résultats.json]** ;
  - ligne 7 : les fonctions de la version 3 sont trouvées dans le module **[impots.modules.impôts_module_02]** ;

11.3 Le module [impots.v02.modules.impôts_module_02]

Le module **[impots.v02.modules.impôts_module_02]** a la structure suivante :



- on retrouve dans le module des fonctions déjà présentes dans le module utilisé par la version 1 avec cependant une différence. Lorsque le module de la version 2 reprend une fonction présente dans le module de la version 1, elle le fait avec un paramètre supplémentaire : **[adminData]** (lignes 29, 51, 77, 127). Ce paramètre représente le dictionnaire des données fiscales issues du fichier jSON **[adminData.json]**. Dans le module de la version 1, ces données n'avaient pas besoin d'être passées aux fonctions car elles étaient définies globalement à celles-ci ce qui faisait que les fonctions les connaissaient ;

11.4 Lecture des données de l'administration fiscale

La fonction **[get_admindata]** est la suivante :

```
1.  # lecture des données de l'administration fiscale dans un fichier jSON
2.  # ----------------------------------------
3.  def get_admindata(admindata_filename: str) -> dict:
4.      # lecture des données de l'administration fiscale
5.      # on laisse remonter les éventuelles exceptions : absence du fichier, contenu jSON incorrect
6.      file = None
7.      try:
8.          # ouverture du fichier jSON en lecture
9.          file = codecs.open(admindata_filename, "r", "utf8")
10.         # transfert du contenu dans un dictionnaire
11.         admin_data = json.load(file)
12.         # on rend le résultat
13.         return admin_data
14.     finally:
15.         # fermeture du fichier s'il a été ouvert
16.         if file:
17.             file.close()
```

- ligne 9 : on récupère le dictionnaire image du fichier jSON lu ;

11.5 Enregistrement des résultats

La fonction **[record_results_in_json_file]** est la suivante :

```
1.  # écriture des résultats dans un fichier jSON
2.  # ----------------------------------------
3.  def record_results_in_json_file(results_filename: str, results: list):
4.      file = None
5.      try:
6.          # ouverture du fichier des résultats
7.          file = codecs.open(results_filename, "w", "utf8")
8.          # écriture en bloc
9.          json.dump(results, file, ensure_ascii=False)
10.     finally:
11.         # on ferme le fichier s'il a été ouvert
12.         if file:
13.             file.close()
```

- ligne 7 : on crée un fichier encodé en UTF-8 ;
- ligne 9 : on écrit la liste **[results]** dans le fichier jSON. Les caractères UTF-8 ne sont pas échappés (`ensure_ascii=False`) ;

11.6 Modification des fonctions

Certaines fonctions reçoivent désormais un paramètre `[admin_data]` supplémentaire. Cela modifie un peu leur écriture. Prenons par exemple la fonction **[calcul_impôt]** :

```
1.  # calcul de l'impôt - étape 1
2.  # ----------------------------------------
3.  def calcul_impôt(admin_data: dict, marié: str, enfants: int, salaire: int) -> dict:
4.      # marié : oui, non
5.      # enfants : nombre d'enfants
6.      # salaire : salaire annuel
7.      # limites, coeffr, coeffn : les tableaux des données permettant le calcul de l'impôt
8.      #
9.      # calcul de l'impôt avec enfants
10.     result1 = calcul_impôt_2(admin_data, marié, enfants, salaire)
11.     impot1 = result1["impôt"]
12.     # calcul de l'impôt sans les enfants
13.     if enfants != 0:
14.         result2 = calcul_impôt_2(admin_data, marié, 0, salaire)
15.         impot2 = result2["impôt"]
16.         # application du plafonnement du quotient familial
17.         if enfants < 3:
18.             # PLAFOND_QF_DEMI_PART euros pour les 2 premiers enfants
19.             impot2 = impot2 - enfants * admin_data['plafond_qf_demi_part']
20.         else:
21.             # PLAFOND_QF_DEMI_PART euros pour les 2 premiers enfants, le double pour les suivants
22.             impot2 = impot2 - 2 * admin_data['plafond_qf_demi_part'] - (enfants - 2) * 2 * admin_data[
23.                 'plafond_qf_demi_part']
24.     else:
25.         impot2 = impot1
26.         result2 = result1
27.
28.     # on prend l'impôt le plus fort avec le taux et la surcôte qui vont avec
29.     if impot1 > impot2:
30.         impot = impot1
31.         taux = result1["taux"]
32.         surcôte = result1["surcôte"]
33.     else:
34.         surcôte = impot2 - impot1 + result2["surcôte"]
35.         impot = impot2
36.         taux = result2["taux"]
37.
38.     # calcul d'une éventuelle décôte
39.     décôte = get_décôte(admin_data, marié, salaire, impot)
40.     impot -= décôte
41.     # calcul d'une éventuelle réduction d'impôts
42.     réduction = get_réduction(admin_data, marié, salaire, enfants, impot)
43.     impot -= réduction
44.     # résultat
45.     return {"marié": marié, "enfants": enfants, "salaire": salaire, "impôt": math.floor(impot), "surcôte":
    surcôte,
46.             "décôte": décôte, "réduction": réduction, "taux": taux}
```

Notes

- là où **[calcul_impôt]** appelle d'autres fonctions, elle passe **[admin_data]** en 1er paramètre (lignes 10, 14, 39, 42) ;
- là où **[calcul_impôt]** utilise des constantes fiscales, elle passe désormais par le dictionnaire **[admin_data]** (lignes 19, 22) ;

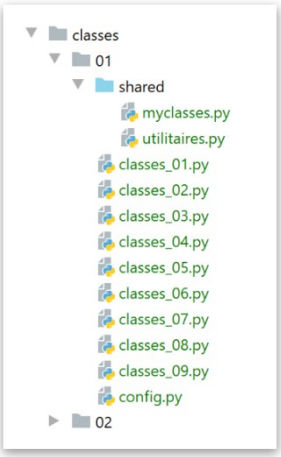
Toutes les fonctions recevant **[admin_data]** comme paramètre, subissent ces mêmes types de modifications.

11.7 Résultats

Les résultats obtenus sont ceux présentés au début du paragraphe 8.3.

12 Les classes et objets

La **classe** est le moule à partir duquel sont fabriqués des **objets**. On dit de l'objet que c'est l'**instance** d'une classe.



Note : le dossier **[shared]** a été placé dans les **[Sources Root]** du projet.

12.1 script [classes_01] : une classe Objet

Le script **[classes_01]** montre une utilisation obsolète des classes :

```
1.  # une classe
2.  class Objet(object):
3.      """une classe Objet vide"""
4.
5.
6.  # toute variable de type [Objet] peut avoir des attributs par construction
7.  obj1 = Objet()
8.  obj1.attr1 = "un"
9.  obj1.attr2 = 100
10. # affiche l'objet
11. print(f"objet1=[{obj1}, {type(obj1)},{id(obj1)},{obj1.attr1},{obj1.attr2}]")
12. # modifie l'objet
13. obj1.attr2 += 100
14. # affiche l'objet
15. print(f"objet1=[{obj1.attr1},{obj1.attr2}]")
16. # affecte la référence obj1 à obj2
17. obj2 = obj1
18. # modifie l'objet pointé par obj2
19. obj2.attr2 = 0
20. # affiche les deux objets - obj1 pointe maintenant sur un objet modifié
21. print(f"objet1=[{obj1.attr1},{obj1.attr2}]")
22. print(f"objet2=[{obj2.attr1},{obj2.attr2}]")
23. # obj1 et obj2 pointent sur le même objet
24. print(f"objet1=[{obj1}, {id(obj1)},{obj1.attr1},{obj1.attr2}]")
25. print(f"objet2=[{obj2}, {id(obj2)},{obj2.attr1},{obj2.attr2}]")
26. print(obj1 == obj2)
27. # type de l'instance obj1
28. print(f"type(obj1)={type(obj1)}")
29. print(f"isinstance(obj1,Objet)={isinstance(obj1, Objet)}, isinstance(obj1,object)={isinstance(obj1,
    object)}")
30. # tout type est un objet en Python
31. print(f"type(4)={type(4)}")
32. print(f"isinstance(4, int)={isinstance(4, int)}, isinstance(4, object)={isinstance(4, object)}")
```

Notes :

- lignes 2-3 : une classe **[Objet]** vide ;
- ligne 2 : la classe peut être déclarée sous trois formes :
  - class Objet ;
  - class Objet() ;
  - class Objet(object) ;
- ligne 3 : une autre forme de commentaire. Celui-ci précédé de trois " peut s'étaler sur plusieurs lignes ;
- ligne 7 : instanciation de la classe *Objet*. Le résultat est une adresse comme le montreront les lignes 24-26 ;

- lignes 8-9 : initialisation directe de deux attributs de l'objet ;
- ligne 17 : copie de références. Les variables *obj1* et *obj2* sont deux pointeurs (références) sur un même objet ;
- ligne 19 : on modifie l'objet pointé par **[obj2]**. Comme **[obj1]** et **[obj2]** pointent sur le même objet, les affichages des objets **[obj1, obj2]** des lignes 21 et 22 vont montrer que l'objet pointé par **[obj1]** a changé ;
- lignes 24-26 : ces lignes visent à montrer l'égalité des variables **[obj1]** et **[obj2]**. L'affichage de la ligne 26 va le montrer. Dans cette comparaison, ce sont les adresses **[obj1]** et **[obj2]** qui sont égales ;
- chaque objet Python est identifié par un n° unique qu'on obtient avec l'expression **[id(objet)]**. Les lignes 24 et 25 vont montrer que les n°s des objets pointés par **[obj1]** et **[obj2]** sont identiques, montrant par là que ces deux références pointent sur le même objet ;
- lignes 27-29 : la fonction **[isinstance(expr,Type)]** rend le booléen True si l'expression **[expr]** est de type **[Type]**. Ici, on va voir que **[obj1]** est de type **[Objet]**, ce qui semble naturel, mais également de type **[object]**. La classe **[object]** est la **classe parent** de toutes les classes Python. Par propriété de l'héritage de classes, une **classe fille F** a toutes les propriétés de sa **classe parent P** et la fonction **[isinstance(instance de F, P)]** rend True ;
- lignes 30-32 : montrent que type **[int]** est lui aussi un type **[object]**. Tous les types de Python dérivent de la classe **[object]** ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/01/classes_01.py
2.  objet1=[<__main__.Objet object at 0x0000025C3F469BB0>, <class '__main__.Objet'>,2595221838768,un,100]
3.  objet1=[un,200]
4.  objet1=[un,0]
5.  objet2=[un,0]
6.  objet1=[<__main__.Objet object at 0x0000025C3F469BB0>, 2595221838768,un,0]
7.  objet2=[<__main__.Objet object at 0x0000025C3F469BB0>, 2595221838768,un,0]
8.  True
9.  type(obj1)=<class '__main__.Objet'>
10. isinstance(obj1,Objet)=True, isinstance(obj1,object)=True
11. type(4)=<class 'int'>
12. isinstance(4, int)=True, isinstance(4, object)=True
13.
14. Process finished with exit code 0
```

12.2 Script [classes_02] : une classe Personne

Le script **[classes_02]** montre que les attributs d'une classe sont publics : ils sont accessibles directement de l'extérieur de la classe. On a là encore une utilisation des classes déconseillée. On la donne néanmoins parce qu'on peut parfois trouver ce type de code (Python l'accepte) et il faut alors être capable de le comprendre.

```
1.  # classe Personne
2.  class Personne:
3.      # attributs de la classe
4.      # non déclarés - peuvent être créés dynamiquement
5.
6.      # méthode
7.      def identité(self: object) -> str:
8.          # a priori, utilise des attributs inexistants
9.          return f"[{self.prénom},{self.nom},{self.âge}]"
10.
11.
12. # ---------------------------------- main
13. # les attributs sont publics et peuvent être créés dynamiquement
14. # instanciation classe
15. p = Personne()
16. # initialisation directe d'attributs de la classe
17. p.prénom = "Paul"
18. p.nom = "de la Hûche"
19. p.âge = 48
20. # appel d'une méthode de la classe
21. print(f"personne={p.identité()}")
22. # type de l'instance p
23. print(f"type(p)={type(p)}")
24. print(f"isinstance(Personne)={isinstance(p, Personne)}, isinstance(object)={isinstance(p, object)}")
```

Notes :

- lignes 2-9 : une classe avec une méthode ;
- ligne 7 : toute méthode d'une classe doit avoir pour premier paramètre, l'objet ***self*** qui désigne l'objet courant. La méthode **[identité]** rend une chaîne de caractères ;

- ligne 15 : instanciation d'un objet **[Personne]** ;
- lignes 16-19 : montrent que les attributs de l'objet peuvent être créés dynamiquement (ils n'existent pas dans la définition de la classe) ;
- ligne 9 : les attributs de la classe sont désignés par la notation **[self.attribut]** ;
- les lignes 23-24 vont montrer que l'objet **[p]** est à la fois une instance de la classe **[Personne]** et de la classe **[object]** ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/01/classes_02.py
2.  personne=[Paul,de la Hûche,48]
3.  type(p)=<class '__main__.Personne'>
4.  isinstance(Personne)=True, isinstance(object)=True
5.
6.  Process finished with exit code 0
```

12.3 Script [classes_03] : la classe Personne avec un constructeur

Le script **[classes_03]** montre l'utilisation normale d'une classe :

```
1.  # classe Personne
2.  class Personne:
3.      # constructeur - initialise trois attributs
4.      def __init__(self: object, prénom: str, nom: str, âge: int):
5.          # prénom : prénom de la personne
6.          # nom : nom de la personne
7.          # âge : âge de la personne
8.
9.          self.prénom = prénom
10.         self.nom = nom
11.         self.âge = âge
12.
13.     # rend les attributs de la classe sous la forme d'une chaîne de caractères
14.     def identité(self: object) -> str:
15.         return f"[{self.prénom},{self.nom},{self.âge}]"
16.
17.
18. # ---------------------------------- main
19. # un objet Personne
20. p = Personne("Paul", "de la Hûche", 48)
21. # appel d'une méthode
22. print(f"personne={p.identité()}")
```

Notes :

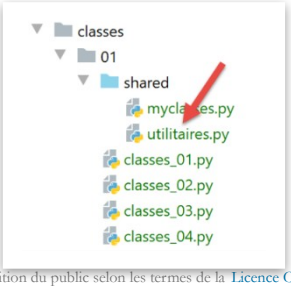
- ligne 4 : le constructeur d'une classe s'appelle **__init__**. Comme pour les autres méthodes, son premier paramètre est **self ;**
- ligne 20 : un objet *Personne* est construit avec le constructeur de la classe ;
- lignes 13-15 : la méthode **[identité]** rend une chaîne de caractères représentant le contenu de l'objet ;
- ligne 22 : affichage de l'identité de la personne ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/01/classes_03.py
2.  personne=[Paul,de la Hûche,48]
3.
4.  Process finished with exit code 0
```

12.4 Script [classes_04] : méthode statiques

Nous définissons dans le dossier **[modules]**, la classe **[Utils]** suivante (utils.py) :



```
1.  class Utils:
2.      # méthode statique
3.      @staticmethod
4.      def is_string_ok(string: str) -> bool:
5.          # string est-elle une chaîne
6.          erreur = not isinstance(string, str)
7.          if not erreur:
8.              # la chaîne est-elle vide ?
9.              erreur = string.strip() == ''
10.         # résultat
11.         return not erreur
```

Notes

- ligne 3 : l'annotation **[@staticmethod]** indique que la méthode ainsi annotée est une méthode de classe et non une méthode d'instance. Cela se voit au fait que le premier paramètre de la méthode ainsi annotée n'est pas le mot clé **[self]**. Ainsi la méthode statique n'a pas accès aux attributs de l'objet. Au lieu d'écrire :

```
1.  u=Utils()
2.  print(u.is_string_ok("abcd")
```

on écrit

```
1.  print(Utils.is_string_ok("abcd")
```

Parce que ci-dessus on a écrit **[Utils.is_string_ok]**, la méthode **[is_string_ok]** est dite une méthode de classe (la classe Utils ici). Pour cela, la méthode **[Utils.is_string_ok]** doit être annotée avec le mot clé **[@staticmethod]**.

La méthode statique **[Utils.is_string_ok]** permet ici de vérifier qu'une donnée est une chaîne de caractères non vide.

Le script **[classes_04]** utilise la classe **[Utils]** de la façon suivante :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from utilitaires import Utils
8.
9.  print(Utils.is_string_ok("  "))
10. print(Utils.is_string_ok(47))
11. print(Utils.is_string_ok(" q "))
```

- lignes 1-4 : on utilise un script de configuration ;

Le script de configuration **[config.py]** est le suivant :

```
1.  def configure():
2.      import os
3.
4.      # dossier du fichier de configuration
5.      script_dir = os.path.dirname(os.path.abspath(__file__))
6.
7.      # chemins absolus des dossiers à mettre dans le syspath
8.      absolute_dependencies = [
9.          f"{script_dir}/shared",
10.         "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/venv/lib/site-packages",
11.         "C:/myprograms/Python38/lib",
12.         "C:/myprograms/Python38/DLLs"
13.     ]
14.
15.     # mise à jour du syspath
16.     from myutils import set_syspath
17.     set_syspath(absolute_dependencies)
18.
19.     # on retourne la config
20.     return {}
```

- ligne 9 : le dossier **[shared]** va être placé dans le Python Path ;

Le résultat de l'exécution est le suivant :

```
C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/01/classes_04.py
False
False
True

Process finished with exit code 0
```

12.5 Script `[classes_05]` : contrôles de validité des attributs

Le script **`[classes_05]`** introduit de nouvelles notions :

- définition d'un type d'exception propriétaire ;
- définition de la méthode **`[_str_]`** qui est la méthode d'identité par défaut des classes ;
- définition de propriétés ;

```
# on configure l'application
import config

config = config.configure()

# le syspath est configuré - on peut faire les imports
from utilitaires import Utils


# une classe d'exception propriétaire dérivant de [BaseException]
class MyException(BaseException):
    # on ne fait rien : classe vide
    pass


# classe Personne
class Personne:
    # constructeur
    def __init__(self: object, prénom: str = "x", nom: str = "y", âge: int = 0):
        # prénom : prénom de la personne
        # nom : nom de la personne
        # âge : âge de la personne

        # mémorisation des paramètres
        # les initialisations seront faites via les setters
        self.prénom = prénom
        self.nom = nom
        self.âge = âge

    # méthode toString de la classe
    def __str__(self: object) -> str:
        return f"[{self.__prénom},{self.__nom},{self.__âge}]"

    # getters
    @property
    def prénom(self) -> str:
        return self.__prénom

    @property
    def nom(self) -> str:
        return self.__nom

    @property
    def âge(self) -> int:
        return self.__âge

    # setters
    @prénom.setter
    def prénom(self, prénom: str):
        # le prénom doit être non vide
        if Utils.is_string_ok(prénom):
            self.__prénom = prénom.strip()
        else:
            raise MyException("Le prénom doit être une chaîne de caractères non vide")

    @nom.setter
```

```
57.     def nom(self, nom: str):
58.         # le prénom doit être non vide
59.         if Utils.is_string_ok(nom):
60.             self.__nom = nom.strip()
61.         else:
62.             raise MyException("Le nom doit être une chaîne de caractères non vide")
63.
64.     @âge.setter
65.     def âge(self, âge: int):
66.         # l'âge doit être un entier >=0
67.         erreur = False
68.         if isinstance(âge, int):
69.             if âge >= 0:
70.                 self.__âge = âge
71.             else:
72.                 erreur = True
73.         else:
74.             erreur = True
75.         # erreur ?
76.         if erreur:
77.             raise MyException("L'âge doit être un entier >=0")
78.
79.
80. # ---------------------------------- main
81. # un objet Personne
82. try:
83.     # instanciation classe Personne
84.     p = Personne("Paul", "de la Hûche", 48)
85.     # affichage objet p
86.     print(f"personne={p}")
87. except MyException as erreur:
88.     # affichage erreur
89.     print(erreur)
90.
91. # un autre objet Personne
92. try:
93.     # instanciation classe Personne
94.     p = Personne("xx", "yy", "zz")
95.     # affichage objet p
96.     print(f"personne={p}")
97. except MyException as erreur:
98.     # affichage erreur
99.     print(erreur)
100.
101.# une autre personne sans paramètres cette fois
102.try:
103.    # instanciation classe Personne
104.    p = Personne()
105.    # affichage objet p
106.    print(f"personne={p}")
107.except MyException as erreur:
108.    # affichage msg d'erreur
109.    print(erreur)
110.
111.# on ne peut pas accéder aux attributs privés __attr de la classe
112.p.__prénom = "Gaëlle"
113.print(f"p.prénom={p.prénom}")
114.print(f"p.__prénom={p.__prénom}")
115.p.prénom = "Sébastien"
116.print(f"p.prénom={p.prénom}")
117.print(f"p.__prénom={p.__prénom}")
```

Notes :

- lignes 10-13 : une classe *MyException* dérivée de la classe *BaseException* (nous verrons ce point un peu plus loin). Elle n'ajoute aucune fonctionnalité à cette dernière. Elle n'est là que pour avoir une exception propriétaire ;
- ligne 19 : le constructeur a des valeurs par défaut pour ses paramètres. Ainsi l'opération *p=Personne()* est équivalente à *p=Personne("x","y",0)* ;
- lignes 34-45 : les propriétés de la classe. Ce sont des méthodes annotées avec le mot clé **`[@property]`**. Elles sont utilisées pour rendre la valeur des attributs ;
- lignes 47-77 : les setters de la classe. Ce sont des méthodes annotées avec le mot clé **`[@attributsetter]`**. Elles sont utilisées pour fixer la valeur des attributs ;
- lignes 48-54 : le setter de l'attribut **`[prénom]`**. Cette méthode sera appelée chaque fois qu'on affectera une valeur à l'attribut **`[prénom]`** :

```
1.  p=Personne(…)
2.  p.prénom=valeur
```

La ligne 2 provoquera l'appel **[p.prénom(valeur)]**. L'intérêt de passer par un setter pour affecter une valeur à un attribut est que le setter étant une fonction, on peut vérifier la validité de la valeur affectée à l'attribut ;

- ligne 51 : on vérifie que la valeur affectée à l'attribut **[prénom]** est une chaîne de caractères non vide. On utilise pour cela la méthode statique **[Utils.isStringOk]** vue précédemment ;
- ligne 52 : la valeur affectée à l'attribut **[prénom]** est débarrassée de ces "blancs" de début / fin de chaîne et affectée à l'attribut **[self.__prénom]**. Ce n'est donc pas l'attribut **[prénom]** qui est utilisé ici. On ne pouvait pas sinon on aurait eu un appel récursif infini. On pouvait utiliser n'importe quel nom d'attribut. Le fait d'avoir utilisé l'attribut **[__prénom]** avec deux underscores en début de l'identificateur a une signification spéciale : les attributs précédés de deux underscores sont **privés** à la classe. Cela signifie qu'ils ne sont pas visibles de l'extérieur de celle-ci. On ne peut donc écrire :

```
1.  p=Personne(…)
2.  p.__prénom=valeur
```

En fait on verra bientôt qu'on peut l'écrire mais que ça ne modifie pas le prénom. Ça fait autre chose ;

- lignes 53-54 : si la valeur affectée au prénom n'est pas correct, on lance une exception. Ainsi le code appelant saura que son appel est incorrect ;
- lignes 35-37 : la propriété **[prénom]**. Elle sera appelée à chaque fois qu'on écrira **[p.prénom]** dans une expression. C'est alors la méthode **[p.prénom()]** qui sera appelée. Ligne 37, on rend la valeur de l'attribut **[__prénom]** puisqu'on a vu que le setter de l'attribut **[prénom]** affectait sa valeur à l'attribut privé **[__prénom]** ;
- lignes 56-62 : le setter de l'attribut **[nom]** est construit de façon analogue à celui de l'attribut **[prénom]**. Il en est de même pour celui de l'attribut **[âge]** aux lignes 64-77 ;
- bien que les propriétés **[prénom, nom, valeur]** ne soient pas les véritables attributs qui sont en réalité **[__prénom, __nom, __âge]**, on continuera à les appeler les attributs de la classe, car elles s'utilisent comme tels ;
- lignes 19-28 : le constructeur de la classe utilise de façon implicite les setters des attributs **[prénom, nom, âge]**. En effet, en écrivant, ligne 26, **[self.prénom = prénom]**, c'est implicitement la méthode **[prénom(self, prénom)]** qui va être appelée. La validité du paramètre **[prénom]** va alors être vérifiée. Il va en être de même pour les deux autres attributs **[nom, âge]** ;
- avec ce modèle, on ne peut attribuer de valeurs incorrectes aux attributs **[prénom, nom, âge]** de la classe ;
- lignes 30-32 : la fonction *__str__* remplace la méthode qui s'appelait *identité* précédemment. Le nom **[__str__]** (2 underscores devant et derrière) n'est pas anodin. On va le voir par la suite ;
- lignes 83-86 : instanciation d'une personne puis affichage de son identité ;
- ligne 84 : instanciation ;
- ligne 86 : affichage. L'opération demande d'afficher la personne *p* sous la forme d'une chaîne de caractères. L'interpréteur Python appelle alors **automatiquement** la méthode *p.__str__()* si elle existe. Cette méthode joue le même rôle que la méthode *toString()* en Java ou dans les langages .NET ;
- lignes 87-89 : gestion d'une éventuelle exception de type *MyException*. Affiche alors l'erreur ;
- lignes 91-99 : idem pour une deuxième personne instanciée avec des paramètres erronés ;
- lignes 102-109 : idem pour une troisième personne instanciée avec les paramètres par défaut : on ne passe aucun paramètre. Ce sont alors les valeurs par défaut de ces paramètres dans le constructeur qui sont ici utilisées ;
- lignes 112-117 : on a dit que l'attribut **[__prénom]** était privé donc normalement non accessible de l'extérieur de la classe. On veut le vérifier ;
- lignes 112-114 : on affecte une valeur à l'attribut **[__prénom]** puis on vérifie la valeur des attributs **[__prénom]** et **[prénom]** qui normalement sont les mêmes ;
- lignes 115-117 : on recommence l'opération en initialisant cette fois l'attribut **[prénom]** ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/01/classes_05.py
2.  personne=[Paul,de la Hûche,48]
3.  L'âge doit être un entier >=0
4.  personne=[x,y,0]
5.  p.prénom=x
6.  p.__prénom=Gaëlle
7.  p.prénom=Sébastien
8.  p.__prénom=Gaëlle
9.
10. Process finished with exit code 0
```

Notes

- lignes 5-6 : on voit que l'affectation **[p.__prénom = "Gaëlle"]** n'a pas changé la valeur de l'attribut **[prénom]**, ligne 5 ;
- lignes 7-8 : on voit que l'affectation **[p.prénom = "Sébastien"]** n'a pas changé la valeur de l'attribut **[__prénom]**, ligne 8 ;

Que faut-il en déduire ? Que probablement, l'opération **[p.__prénom = "Gaëlle"]** a créé un attribut public **[__prénom]** à la classe mais que celui-ci est différent de l'attribut privé **[__prénom]** manipulé au sein de celle-ci ;

12.6 Script [classes_06] : ajout d'une méthode d'initialisation de l'objet

Le script **[classes_06]** ajoute une méthode à la classe **[Personne]** :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from utilitaires import Utils
8.
9.
10. # une classe d'exception propriétaire dérivant de [BaseException]
11. class MyException(BaseException):
12.     # on ne fait rien : classe vide
13.     pass
14.
15.
16. # classe Personne
17. class Personne:
18.     # constructeur
19.     def __init__(self: object, prénom: str = "x", nom: str = "y", âge: int = 0):
20.         # prénom : prénom de la personne
21.         # nom : nom de la personne
22.         # âge : âge de la personne
23.
24.         # mémorisation des paramètres
25.         # les initialisations seront faites via les setters
26.         self.prénom = prénom
27.         self.nom = nom
28.         self.âge = âge
29.
30.     # autre méthode d'initialisation
31.     def init_with_personne(self: object, p: object):
32.         # initialise l'objet courant avec une personne p
33.         self.__init__(p.prénom, p.nom, p.âge)
34.
35.     # méthode toString de la classe
36.     def __str__(self: object) -> str:
37.         return f"[{self.__prénom},{self.__nom},{self.__âge}]"
38.
39.     # getters
40.     …
41.
42.     # setters
43.     …
44.
45.
46. # ---------------------------------- main
47. # un objet Personne
48. try:
49.     # instanciation classe Personne
50.     p = Personne("Paul", "de la Hûche", 48)
51.     # affichage objet p
52.     print(f"personne={p}")
53. except MyException as erreur:
54.     # affichage msg d'erreur
55.     print(erreur)
56.
57. # un autre objet Personne
58. try:
59.     # instanciation classe Personne
60.     p = Personne("xx", "yy", "zz")
61.     # affichage objet p
62.     print(f"p={p}")
63. except MyException as erreur:
64.     # affichage msg d'erreur
65.     print(erreur)
66.
67. # une autre personne sans paramètres cette fois
68. try:
69.     # instanciation classe Personne
```

```
70.     p = Personne()
71.     # affichage objet p
72.     print(f"p={p}")
73. except MyException as erreur:
74.     # affichage msg d'erreur
75.     print(erreur)
76.
77. # une autre Personne obtenue par recopie
78. try:
79.     # instanciation classe Personne
80.     p2 = Personne()
81.     p2.init_with_personne(p)
82.     # affichage objet p2
83.     print(f"p2={p2}")
84. except MyException as erreur:
85.     # affichage msg d'erreur
86.     print(erreur)
```

Notes :

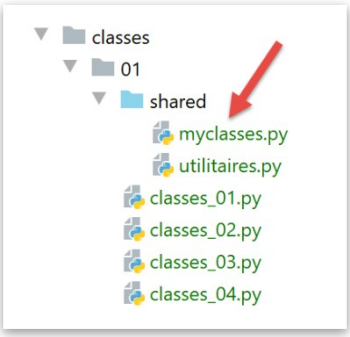
- la différence avec le script précédent est en lignes 30-33. On a rajouté la méthode *initWithPersonne*. Celle-ci fait appel au constructeur *__init__*. Il n'y a pas possibilité d'avoir, comme dans les langages typés, des méthodes de même nom différenciées par la nature de leurs paramètres ou de leur résultat. Il n'y a donc pas possibilité d'avoir plusieurs constructeurs qui construiraient l'objet à partir de paramètres différents, ici un objet de type *Personne* ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/01/classes_06.py
2.  personne=[Paul,de la Hûche,48]
3.  L'âge doit être un entier >=0
4.  p=[x,y,0]
5.  p2=[x,y,0]
6.
7.  Process finished with exit code 0
```

12.7 Script `[classes_07]` : une liste d'objets Personne

Nous mettons désormais les classes **`[MyException]`** et **`[Personne]`** dans un module pour pouvoir les utiliser sans avoir à recopier leur



code :

Les deux classes seront dans le module **`[myclasses.py]`** ci-dessus.

Le script **`[classes_07]`** montre qu'on peut avoir une liste d'objets :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from myclasses import Personne
8.
9.  # --------------------------------- main
10. # création d'une liste d'objets personne
11. groupe = [Personne("Paul", "Langevin", 48), Personne("Sylvie", "Lefur", 70)]
12. # identité de ces personnes
```

```
13. for i in range(len(groupe)):
14.     print(f"groupe[{i}]={groupe[i]}")
```

Notes :

- ligne 7 : on importe la classes **[Personne]** ;
- ligne 11 : un liste d'objets de type **[Personne]** ;
- lignes 13-14 : on parcourt cette liste pour afficher chacun de ses éléments ;
- ligne 14 : la fonction **[print]** va afficher la chaîne représentant l'objet **[groupe[i]]**. Par défaut, c'est la méthode **[__str__]** de ceux-ci qui sera appelée ;

Résultats

```
1.  C:\Users\serge\.virtualenvs\cours-python-v02\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/v-02/classes/01/classes_07.py
2.  groupe[0]=[Paul,Langevin,48]
3.  groupe[1]=[Sylvie,Lefur,70]
4.
5.  Process finished with exit code 0
```

12.8 Script [classes_08] : création d'une classe dérivée de la classe Personne

Nous définissons dans le module **[myclasses]**, la classe **[Enseignant]** suivante :

```
1.  # classe Enseignant
2.  class Enseignant(Personne):
3.      # constructeur
4.      def __init__(self, prénom: str = "x", nom: str = "x", âge: int = 0, discipline: str = "x"):
5.          # prénom : prénom de la personne
6.          # nom : nom de la personne
7.          # âge : âge de la personne
8.          # discipline : discpline enseignée
9.
10.         # initialisation du parent
11.         Personne.__init__(self, prénom, nom, âge)
12.         # autres initialisations
13.         self.discipline = discipline
14.
15.     # toString
16.     def __str__(self) -> str:
17.         return f"enseignant[{super().__str__()},{self.discipline}]"
18.
19.     # propriétés
20.     @property
21.     def discipline(self) -> str:
22.         return self.__discipline
23.
24.     @discipline.setter
25.     def discipline(self, discipline: str):
26.         # la discipline doit être une chaîne non vide
27.         if Utils.is_string_ok(discipline):
28.             self.__discipline = discipline
29.         else:
30.             raise MyException("La discipline doit être une chaîne de caractères non vide")
```

- ligne 2 : déclare la classe *Enseignant* comme étant une classe dérivée de la classe *Personne*. Une classe dérivée a toutes les propriétés (attributs et méthodes) de sa classe parent plus celles qui lui sont propres ;
- ligne 13 : la classe **[Enseignant]** définit un nouvel attribut **[discipline]** ;
- ligne 11 : le constructeur de la classe dérivée *Enseignant* doit appeler le constructeur de la classe parent *Personne* en lui transmettant les paramètres qu'il attend ;
- ligne 17 : la fonction **[super()]** rend la classe parent. Ici on appelle la fonction **[__str__]** de la classe parent ;
- lignes 19-30 : on définit le getter et setter du nouvel attribut **[discipline]** ;

Le script **[classes_08]** utilise la classe **[Enseignant]** de la façon suivante :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
```

```
7.  from myclasses import Personne, Enseignant
8.
9.  # ---------------------------------- main
10. # création d'un tableau d'objets Personne et dérivés
11. groupe = [Enseignant("Paul", "Langevin", 48, "anglais"), Personne("Sylvie", "Lefur", 70)]
12. # identité de ces personnes
13. for i in range(len(groupe)):
14.     print(f"groupe[{i}]={groupe[i]}")
```

Notes :

- ligne 7 : on importe les classes **[Personne]** et **[Enseignant]** définies dans le fichier **[myclasses.py]** ;
- lignes 11-14 : on définit un groupe de personnes dont on affiche ensuite l'identité ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/01/classes_08.py
2.  groupe[0]=enseignant[[Paul,Langevin,48],anglais]
3.  groupe[1]=[Sylvie,Lefur,70]
4.
5.  Process finished with exit code 0
```

12.9 Script [classes_09] : seconde classe dérivée de la classe Personne

Le script **[classes_09]** introduit la classe **[Etudiant]** dérivée de la classe **[Personne]**. Celle-ci est définie de la façon suivante dans le module **[myclasses]** :

```
1.  # classe Etudiant
2.  class Etudiant(Personne):
3.      # constructeur
4.      def __init__(self: object, prénom: str = "x", nom: str = "y", âge: int = 0, formation: str = "x"):
5.          Personne.__init__(self, prénom, nom, âge)
6.          self.formation = formation
7.
8.      # toString
9.      def __str__(self: object) -> str:
10.         return f"étudiant[{super().__str__()},{self.formation}]"
11.
12.     # propriétés
13.     @property
14.     def formation(self) -> str:
15.         return self.__formation
16.
17.     @formation.setter
18.     def formation(self, formation: str):
19.         # la formation doit être une chaîne non vide
20.         if Utils.is_string_ok(formation):
21.             self.__formation = formation
22.         else:
23.             raise MyException("La formation doit être une chaîne de caractères non vide")
```

Le script **[classes_09]** utilise la classe **[Etudiant]** de la façon suivante :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from myclasses import Personne, Enseignant, Etudiant
8.
9.  # ---------------------------------- main
10. # création d'un tableau d'objets Personne et dérivés
11. groupe = [Enseignant("Paul", "Langevin", 48, "anglais"), Personne("Sylvie", "Lefur", 70),
12.           Etudiant("Steve", "Boer", 22, "iup2 qualité")]
13. # identité de ces personnes
14. for personne in groupe:
15.     # affichage personne
16.     print(personne)
```

Notes :

- ce script est analogue au précédent.

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/01/classes_09.py
2.  enseignant[[Paul,Langevin,48],anglais]
3.  [Sylvie,Lefur,70]
4.  étudiant[[Steve,Boer,22],iup2 qualité]
5.
6.  Process finished with exit code 0
```

12.10 Script `[classes_10]` : la propriété `[__dict__]`

Le script **`[classes_10]`** présente la propriété **`[__dict__]`** que nous allons souvent utiliser par la suite :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from myclasses import Etudiant
8.
9.  # ---------------------------------- main
10. # création d'un étudiant
11. étudiant=Etudiant("Steve", "Boer", 22, "iup2 qualité")
12. # dictionnaire des propriétés
13. print(étudiant.__dict__)
```

Commentaires

- lignes 1-4 : l'application est configurée ;
- ligne 7 : la classe **`[Etudiant]`** est importée ;
- ligne 11 : instanciation d'un étudiant ;
- ligne 13 : utilisation de la méthode prédéfinie **`[__dict__]`** (2 caractères soulignés avant et après l'identifiant) ;

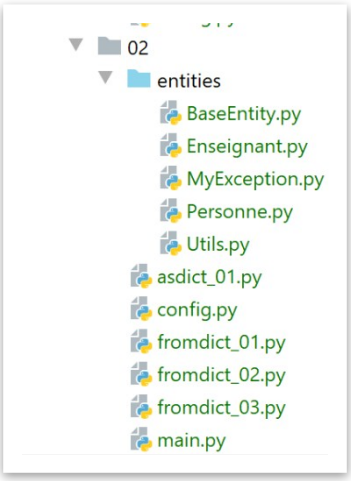
Les résultats sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/01/classes_10.py
2.  {'_Personne__prénom': 'Steve', '_Personne__nom': 'Boer', '_Personne__âge': 22, '_Etudiant__formation':
    'iup2 qualité'}
3.
4.  Process finished with exit code 0
```

- ligne 2, on obtient un dictionnaire dont les clés sont les propriétés de l'objet préfixées par le nom de la classe à laquelle elles appartiennent. Nous utiliserons ce dictionnaire pour faire une passerelle entre objet et dictionnaire ;

13 Les classes génériques `[BaseEntity]` et `[MyException]`

Nous définissons maintenant deux classes que nous utiliserons régulièrement par la suite.



13.1 La classe MyException

La classe **`[MyException]`** (MyException.py) fournit une classe d'exceptions propriétaire :

```
1.  # une classe d'exception propriétaire dérivant de [BaseException]
2.  class MyException(BaseException):
3.      # constructeur
4.      def __init__(self: object, code: int, message: str):
5.          # parent
6.          BaseException.__init__(self, message)
7.          # code erreur
8.          self.code = code
9.
10.     # toString
11.     def __str__(self):
12.         return f"MyException[{self.code}, {super().__str__()}]"
13.
14.     # getter
15.     @property
16.     def code(self) -> int:
17.         return self.__code
18.
19.     # setter
20.     @code.setter
21.     def code(self, code: int):
22.         # le code d'erreur doit être un entier positif
23.         if isinstance(code, int) and code > 0:
24.             self.__code = code
25.         else:
26.             # exception
27.             raise BaseException(f"code erreur {code} incorrect")
```

Notes

- ligne 2 : la classe **`[MyException]`** dérive de la classe prédéfinie **`[BaseException]`** ;
- ligne 4 : le constructeur accepte deux paramètres :
  - **`[code]`** : un code d'erreur entier ;
  - **`[message]`** : un message d'erreur ;
- ligne 6 : on passe le message d'erreur à la classe parent ;
- lignes 14-27 : l'attribut **`[code]`** est manipulé via un getter / setter ;
- lignes 23-24 : la validité de l'attribut **`[code]`** est vérifiée : il faut que ce soit un entier >0 ;

13.2 La classe `[BaseEntity]`

La classe **`[BaseEntity]`** sera la classe parent de la plupart des classes que nous créerons pour encapsuler des informations sur un objet. Dans la suite nous utiliserons principalement deux types de classes :

- des classes qui n'ont pour but que d'encapsuler au même endroit des informations sur un même objet. Celles-ci n'auront pas de comportements (méthodes) autres que des getters / setters et une fonction d'affichage (__str__). S'il y a N objets à gérer, ces classes sont instanciées N fois. **`[BaseEntity]`** sera la classe parent de ce type de classes ;
- des classes dont le rôle principal est d'encapsuler des méthodes et très peu d'informations. Ces classes ne seront instanciées qu'une fois (**singleton**). Leur rôle est d'implémenter les algorithmes d'une application ;

La classe **`[BaseEntity]`** est la suivante :

```
1.  # imports
2.  import json
3.  import re
4.
5.  from MyException import MyException
6.
7.
8.  class BaseEntity(object):
9.      # propriétés exclues de l'état de la classe
10.     excluded_keys = []
11.
12.     # propriétés de la classe
13.     @staticmethod
14.     def get_allowed_keys() -> list:
15.         # id : identifiant de l'objet
16.         return ["id"]
17.
18.     # toString
19.     def __str__(self) -> str:
20.         return self.asjson()
21.
22.     # getter
23.     @property
24.     def id(self) -> int:
25.         return self.__id
26.
27.     #  setter
28.     @id.setter
29.     def id(self, id):
30.         # l'id doit être un entier >=0
31.         try:
32.             id = int(id)
33.             erreur = id < 0
34.         except:
35.             erreur = True
36.         # erreur ?
37.         if erreur:
38.             raise MyException(1, f"L'identifiant d'une entité {self.__class__} doit être un entier >=0")
39.         else:
40.             self.__id = id
41.
42.     def fromdict(self, state: dict, silent=False):
43.         …
44.
45.     def set_value(self, key: str, value, new_attributes) -> dict:
46.         …
47.
48.     def asdict(self, included_keys: list = None, excluded_keys: list = []) -> dict:
49.         …
50.
51.     def asjson(self, excluded_keys: list = []) -> str:
52.         …
53.
54.     def fromjson(self, json_state: str):
55.         …
```

Commentaires

- l'objectif de la classe **`[BaseEntity]`** est de faciliter les conversions Objet / Dictionnaire et Objet / jSON. On offre ainsi les méthodes suivantes :

    - **[asdict]** : rend le dictionnaire des propriétés de l'objet ;
    - **[fromdict]** : construit un objet à partir d'un dictionnaire ;
    - **[asjson]** : rend la chaîne jSON de l'objet comme le fait la fonction **[__str__]** ;
    - **[fromjson]** : construit un objet à partir de sa chaîne jSON ;
- la classe **[BaseEntity]** est destinée à être dérivée et non à être utilisée telle quelle ;
- lignes 22-25 : la classe **[BaseEntity]** n'a qu'une propiété, l'entier **[id]**. Cette propriété est l'identifiant de l'objet. Dans la pratique, il est souvent utile de pouvoir différentier les instances d'une même classe. Nous le ferons avec cette propriété, unique pour une instance. Par ailleurs, les objets proviennent souvent de bases de données où ils sont identifiés par une clé primaire, un entier généralement. Dans ces cas, **[id]** sera la clé primaire ;
- lignes 27-40 : le setter de la propriété **[id]**. On vérifie que c'est un entier >=0. Si ce n'est pas le cas, une exception de type **[MyException]** est lancée (ligne 39) ;
- ligne 10 : **[excluded_keys]** est un attribut de classe et non d'instance. Ainsi on écrira **[BaseEntity.excluded_keys]**. Cet attribut de classe est une liste contenant les propriétés de classe qui ne participent pas aux conversions Objet / Dictionnaire et Objet / jSON ;
- lignes 12-16 : **[get_allowed_keys]** rend la liste des propriétés de la classe. Dans une conversion Dictionnaire -> Objet ou jSON -> Objet, on n'acceptera que les clés présentes dans cette liste. Chaque classe dérivant la classe **[BaseEntity]** aura à redéfinir cette liste ;

Il faut comprendre ici que les propriétés et fonctions de la classe **[BaseEntity]** sont accessibles aux classes dérivées de **[BaseEntity]**. C'est le point important à comprendre.

Nous allons détailler le code de la classe **[BaseEntity]**. Il est assez avancé. Le lecteur débutant pourra se contenter de lire le rôle de chaque fonction sans s'apesantir sur son code.

13.2.1 La méthode [BaseEntity.fromdict]

13.2.1.1 Définition

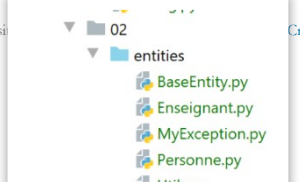
La méthode **[fromdict]** permet d'initialiser un objet **[BaseEntity]** ou dérivé à partir d'un dictionnaire :

```
1.  def fromdict(self, state: dict, silent=False):
2.          # on met à jour l'objet
3.          # clés autorisées
4.          allowed_keys = self.__class__.get_allowed_keys()
5.          # parcourt des clés de state
6.          for key, value in state.items():
7.              # la clé est-elle autorisée ?
8.              if key not in allowed_keys:
9.                  if not silent:
10.                     raise MyException(2, f"la clé {key} n'est pas autorisée")
11.             else:
12.                 # on essaie d'affecter la valeur à la clé
13.                 # on laisse remonter l'éventuelle exception
14.                 setattr(self, key, value)
15.         # on rend l'objet
16.         return self
```

Commentaires

- ligne 1 : la fonction reçoit en paramètre le dictionnaire **[state]** à partir duquel l'objet courant va être initialisé ;
- ligne 4 : on fait appel à la fonction statique **[get_allowed_keys]** de la classe qui a appelé la fonction **[fromdict]**. Si on a affaire à une classe dérivée de **[BaseEntity]** et que cette classe dérivée a redéfini la fonction statique **[get_allowed_keys]** alors c'est la fonction **[get_allowed_keys]** qui est appelée. Chaque classe dérivée redéfinit cette fonction statique pour y déclarer ses propriétés ;
- ligne 6 : on parcourt les clés et valeurs du dictionnaire **[state]** ;
- ligne 8 : si la clé **[key]** ne fait pas partie des propriétés de la classe, alors soit :
    - on l'ignore ;
    - on lance une exception (ligne 10). Le développeur indique ce qu'il souhaite en passant le bon paramètre **[silent]** (ligne 1). La valeur par défaut de **[silent]** fait qu'une exception est lancée si on tente d'initialiser l'objet avec une propriété qu'il n'a pas ;
- ligne 14 : si la clé fait partie des propriétés de l'objet, alors on l'affecte à l'objet **[self]** à l'aide de la fonction prédéfinie **[setattr]** ;
- ligne 16 : la fonction rend l'objet initialisé ;

13.2.1.2 Exemples



13.2.1.2.1 La classe [Utils]

La classe **[Utils]** (Utils.py) est la suivante :

```
1.  class Utils:
2.      # méthode statique
3.      @staticmethod
4.      def is_string_ok(string: str) -> bool:
5.          # string est-elle une chaîne
6.          erreur = not isinstance(string, str)
7.          if not erreur:
8.              # la chaîne est-elle vide ?
9.              erreur = string.strip() == ''
10.         # résultat
11.         return not erreur
```

Elle définit aux lignes 3-11, une méthode statique qui rend un booléen vrai si son paramètre **[str]** est une chaîne de caractères non vide ;

13.2.1.2.2 La classe [Personne]

La classe **[Personne]** (Personne.py) dérive de la classe **[BaseEntity]** :

```
1.  # imports
2.  from BaseEntity import BaseEntity
3.  from MyException import MyException
4.  from Utils import Utils
5.
6.
7.  # classe Personne
8.  class Personne(BaseEntity):
9.      # propriétés exclues de l'état de la classe
10.     excluded_keys = []
11.
12.     # propriétés de la classe
13.     # id : identifiant de la personne
14.     # prénom : prénom de la personne
15.     # nom : nom de la personne
16.     # âge : âge de la personne
17.     @staticmethod
18.     def get_allowed_keys() -> list:
19.         # id : identifiant de l'objet
20.         return BaseEntity.get_allowed_keys() + ["nom", "prénom", "âge"]
21.
22.     # getters
23.     @property
24.     def prénom(self) -> str:
25.         return self.__prénom
26.
27.     @property
28.     def nom(self) -> str:
29.         return self.__nom
30.
31.     @property
32.     def âge(self) -> int:
33.         return self.__âge
34.
35.     # setters
36.     @prénom.setter
37.     def prénom(self, prénom: str):
38.         # le prénom doit être non vide
39.         if Utils.is_string_ok(prénom):
40.             self.__prénom = prénom.strip()
41.         else:
42.             raise MyException(11, "Le prénom doit être une chaîne de caractères non vide")
43.
44.     @nom.setter
45.     def nom(self, nom: str):
46.         # le prénom doit être non vide
47.         if Utils.is_string_ok(nom):
48.             self.__nom = nom.strip()
49.         else:
50.             raise MyException(12, "Le nom doit être une chaîne de caractères non vide")
51.
52.     @âge.setter
53.     def âge(self, âge: int):
54.         # l'âge doit être un entier >=0
```

```
55.         erreur = False
56.         if isinstance(âge, int):
57.             if âge >= 0:
58.                 self.__âge = âge
59.             else:
60.                 erreur = True
61.         else:
62.             erreur = True
63.         # erreur ?
64.         if erreur:
65.             raise MyException(13, "L'âge doit être un entier >=0")
```

- ligne 8 : la classe **[Personne]** dérive de la classe **[BaseEntity]** ;
- lignes 8-65 : on a gardé l'essentiel de la classe **[Personne]** déjà rencontrée. Les différences sont les suivantes :
  - la classe n'a plus de constructeur ;
  - la classe utilise l'exception **[MyException]**, exemple ligne 65 ;
  - elle a une méthode statique, **[get_allowed_keys]**, lignes 17-20, qui définit la liste de ses propriétés. Les propriétés propres à la classe **[Personne]** sont ajoutées à celles de la classe parent **[BaseEntity]** ;
  - elle a une liste statique **[excluded_keys]** sur laquelle on reviendra ;

13.2.1.2.3 La classe [Enseignant]

La classe **[Enseignant]** (Enseignant.py) dérive de la classe **[Personne]** :

```
1.  # imports
2.  from MyException import MyException
3.  from Personne import Personne
4.  from Utils import Utils
5.
6.
7.  # classe Enseignant
8.  class Enseignant(Personne):
9.      # propriétés exclues de l'état de la classe
10.     excluded_keys = []
11.
12.     # propriétés de la classe
13.     # id : identifiant de la personne
14.     # prénom : prénom de la personne
15.     # nom : nom de la personne
16.     # âge : âge de la personne
17.     # discipline : discpline enseignée
18.     @staticmethod
19.     def get_allowed_keys() -> list:
20.         # id : identifiant de l'objet
21.         return Personne.get_allowed_keys() + ["discipline"]
22.
23.     # propriétés
24.     @property
25.     def discipline(self) -> str:
26.         return self.__discipline
27.
28.     @discipline.setter
29.     def discipline(self, discipline: str):
30.         # la discipline doit être une chaîne non vide
31.         if Utils.is_string_ok(discipline):
32.             self.__discipline = discipline
33.         else:
34.             raise MyException(21, "La discipline doit être une chaîne de caractères non vide")
35.
36.     # méthode show
37.     def show(self):
38.         print(f"Enseignant[{self.id}, {self.prénom}, {self.nom}, {self.âge}]")
```

- ligne 8 : la classe **[Enseignant]** étend (ou dérive) la classe **[Personne]** ;
- lignes 18-21 : définissent la liste des propriétés de la classe ;
- lignes 37-38 : la méthode **[show]** affiche l'identité de l'enseignant ;

13.2.1.2.4 La configuration [config]

Les scripts d'exemples utilisent la configuration **[config]** suivante :

```
1.  def configure():
2.      import os
3.
```

```
4.      # dossier du fichier de configuration
5.      script_dir = os.path.dirname(os.path.abspath(__file__))
6.
7.      # chemins absolus des dossiers à mettre dans le syspath
8.      absolute_dependencies = [
9.          # la classe BaseEntity
10.         f"{script_dir}/entities",
11.     ]
12.
13.     # mise à jour du syspath
14.     from myutils import set_syspath
15.     set_syspath(absolute_dependencies)
16.
17.     # on rend la configuration
18.     return {}
```

- lignes 8-10 : les dossiers contenant les dépendances du projet ;
- lignes 14-15 : le Python Path est construit ;
- ligne 18 : on rend un dictionnaire vide (il n'y a pas d'autres configuration que celle du syspath) ;

13.2.1.2.5 Le script [fromdict_01]

Le script **[fromdict_01]** est le suivant :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from Enseignant import Enseignant
8.
9.  # un enseignant
10. enseignant1 = Enseignant().fromdict({"id": 1, "nom": "lourou", "prénom": "paul", "âge": 56})
11. enseignant1.show()
```

- ligne 10 : on crée un objet **[Enseignant]** à partir d'un dictionnaire. Pour cela, on utilise le constructeur par défaut de la classe pour créer un objet **[Enseignant]** à laquel on applique la méthode **[fromdict]**. Il faut comprendre qu'ici la méthode **[fromdict]** exécutée est celle de la classe parent **[BaseEntity]**. En effet :
  - la méthode **[fromdict]** est d'abord cherchée dans la classe **[Enseignant]**. **Elle n'existe pas** ;
  - elle est ensuite cherchée dans la classe parent **[Personne]**. **Elle n'existe pas** ;
  - elle est ensuite cherchée dans la classe parent **[BaseEntity]**. **Elle existe** ;
- ligne 11 : on affiche l'objet **[Enseignant]** ;

Les résultats sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/02/fromdict_01.py
2.  Enseignant[1, paul, lourou, 56]
3.
4.  Process finished with exit code 0
```

13.2.1.2.6 Le script [fromdict_02]

Le script **[fromdict_02]** est le suivant :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from Enseignant import Enseignant
8.
9.  # un enseignant
10. enseignant1 = Enseignant().fromdict({"id": 1, "nom": "lourou", "prénom": "", "âge": 56})
11. enseignant1.show()
```

- ligne 10 : on crée un enseignant avec un prénom vide. Cela doit créer une exception car la classe **[Personne]** n'accepte pas des prénoms vides. Cet exemple montre la différence entre un dictionnaire et un objet. Ce dernier peut vérifier la validité de ses propriétés, pas le dictionnaire ;

Les résultats sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/02/fromdict_02.py
2.  Traceback (most recent call last):
3.    File "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/02/fromdict_02.py", line 10, in
    <module>
4.      enseignant1 = Enseignant().fromdict({"id": 1, "nom": "lourou", "prénom": "", "âge": 56})
5.    File "C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\classes\02/entities\BaseEntity.py", line
    55, in fromdict
6.      setattr(self, key, value)
7.    File "C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\classes\02/entities\Personne.py", line 42,
    in prénom
8.      raise MyException(11, "Le prénom doit être une chaîne de caractères non vide")
9.  MyException.MyException: MyException[11, Le prénom doit être une chaîne de caractères non vide]
10.
11. Process finished with exit code 1
```

13.2.1.2.7 Le script [fromdict_03]

Le script **[fromdict_03]** est le suivant :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from Enseignant import Enseignant
8.
9.  # un enseignant
10. enseignant1 = Enseignant().fromdict({"id": 1, "nom": "lourou", "prénom": "albert", "âge": 56, "sexe": "M"})
11. enseignant1.show()
```

- ligne 10 : on crée un enseignant à partir d'un dictionnaire contenant une clé (sexe) qui n'appartient pas à la classe **[Enseignant]**. Une exception devrait alors être levée ;

Les résultats sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/02/fromdict_03.py
2.  Traceback (most recent call last):
3.    File "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/02/fromdict_03.py", line 10, in
    <module>
4.      enseignant1 = Enseignant().fromdict({"id": 1, "nom": "lourou", "prénom": "albert", "âge": 56, "sexe":
    "M"})
5.    File "C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\classes\02/entities\BaseEntity.py", line
    51, in fromdict
6.      raise MyException(2, f"la clé [{key}] n'est pas autorisée")
7.  MyException.MyException: MyException[2, la clé [sexe] n'est pas autorisée]
8.
9.  Process finished with exit code 1
```

13.2.1.2.8 Le script [fromdict_04]

Le script **[fromdict_04]** est une copie de **[fromdict_03]** à un détail près :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from Enseignant import Enseignant
8.
9.  # un enseignant
10. enseignant1 = Enseignant().fromdict({"id": 1, "nom": "lourou", "prénom": "albert", "âge": 56, "sexe": "M"},
    silent=True)
11. enseignant1.show()
```

- ligne 10 : on a utilisé le paramètre **[silent=True]** pour indiquer que si une clé du dictionnaire n'est pas une propriété de la classe **[Enseignant]**, elle doit simplement être ignorée. Dans ce cas, aucune exception ne sera lancée ;

Les résultats sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/02/fromdict_04.py
2.  Enseignant[1, albert, lourou, 56]
3.
4.  Process finished with exit code 0
```

13.2.2 La méthode `[BaseEntity.asdict]`

13.2.2.1 Définition

La méthode **`[BaseEntity.asdict]`** rend un dictionnaire dont les clés sont les propriétés de l'objet :

```
1.      def asdict(self, included_keys: list = None, excluded_keys: list =[]) -> dict:
2.          # attributs de l'objet
3.          attributes = self.__dict__
4.          # les nouveaux attributs
5.          new_attributes = {}
6.          # on parcourt les attributs
7.          for key, value in attributes.items():
8.              # si la clé est explicitement demandée
9.              if included_keys and key in included_keys:
10.                 self.set_value(key, value, new_attributes)
11.             # sinon, si la clé n'est pas exclue
12.             elif not included_keys and key not in self.__class__.excluded_keys and key not in excluded_keys:
13.                 self.set_value(key, value, new_attributes)
14.         # on rend le dictionnaire des attributs
15.         return new_attributes
```

Commentaires

- ligne 1 : la fonction **`[asdict]`** rend le dictionnaire des propriétés de l'objet ;
- ligne 1 : **`[included_keys]`** : la liste des clés à inclure dans le dictionnaire ;
- ligne 1 : **`[excluded_keys]`** : la liste des clés à exclure du dictionnaire ;
- ligne 3 : la propriété **`[self.__dict__]`** rend le dictionnaire des propriétés de l'objet. Les noms des propriétés sont les clés et leurs valeurs les valeurs du dictionnaire. Un objet peut contenir des références d'autres objets. Les noms des propriétés sont alors préfixées par le nom de la classe à laquelle elles appartiennent. C'est quelque chose qu'on ne veut pas. On veut les propriétés sans leur préfixe ;
- ligne 3 : il faut comprendre ici que si la fonction **`[asdict]`** s'exécute à l'intérieur d'une classe dérivée de **`[BaseEntity]`**, la propriété **`[self.__dict__]`** rend le dictionnaire des propriétés de **l'objet dérivé** ;
- ligne 5 : le dictionnaire que l'on va construire ;
- ligne 7 : on parcourt les valeurs de **`[self.__dict__]`** sous la forme (clé, valeur) ;
- ligne 9 : si la clé courante appartient à la liste des clés à inclure, alors elle est ajoutée au dictionnaire **`[new_attributes]`** par la fonction **`[set_value]`** que nous allons prochainement décrire ;
- ligne 12 : si le paramètre **`[included_keys]`** n'est pas présent, alors le paramètre **`[excluded_keys]`** est exploité. Si la propriété ne fait pas partie des propriétés à exclure, alors elle est ajoutée au dictionnaire **`[new_attributes]`** ;
- ligne 12 : il y a plusieurs façons d'exclure une propriété du dictionnaire :
  - elle a été définie au niveau de l'attribut de classe **`[excluded_keys]`** ;
  - elle a été définie dans la liste **`[excluded_keys]`** passée à la fonction **`[asdict]`** ;
  - le paramètre **`[included_keys]`** est présent et ne comprend pas la propriété ;
- ligne 15 : on rend le dictionnaire **`[new_attributes]`**

La fonction **`[set_value]`** des lignes 10 et 13 est la suivante :

```
1.      @staticmethod
2.      def set_value(key: str, value, new_attributes: dict):
3.          # les clés peuvent être de la forme __Class__key
4.          match = re.match("^.*?__(.*?)$", key)
5.          if match:
6.              # on note la nouvelle clé
7.              newkey = match.groups()[0]
8.          else:
9.              # la clé reste inchangée
10.             newkey = key
11.         # on insère la nouvelle clé dans le dictionnaire [new_attributes]
12.         # en transformant si besoin est la valeur associée en l'un des types
13.         # dict, list, type simple
14.         new_attributes[newkey] = BaseEntity.check_value(value)
```

Commentaires

- ligne 4 : on regarde si la clé est de la forme __Class_key. C'est la forme qu'elle a si elle appartient à un objet inclus dans l'objet principal. Dans ce cas, on ne veut garder que la chaîne **[key]** ;
- ligne 7 : on ne garde que la chaîne qui suit les deux derniers caractères soulignés de la chaîne ;
- ligne 8-10 : si la clé n'est pas de la forme __Class_key alors on la garde telle quelle ;
- lignes 11-14 : la valeur associée à la clé **[newkey]** est calculée par la méthode statique **[BaseEntity.check_value]** ;

La méthode statique **[BaseEntity.check_value]** est la suivante :

```
1.      @staticmethod
2.      def check_value(value):
3.          # la valeur peut être de type BaseEntity, list, dict ou un type simple
4.          # value est-elle une instance de BaseEntity ?
5.          if isinstance(value, BaseEntity):
6.              value2 = value.asdict()
7.          # value est-elle de type list
8.          elif isinstance(value, list):
9.              value2 = BaseEntity.list2list(value)
10.         # value est-elle de type dict ?
11.         elif isinstance(value, dict):
12.             value2 = BaseEntity.dict2dict(value)
13.         # value est un type simple
14.         else:
15.             value2 = value
16.         # on rend le résultat
17.         return value2
```

- ligne 1 : la méthode **[check_value]** est statique (méthode de classe et non d'instance). Elle reçoit en paramètre la valeur à associer à une clé du dictionnaire :
  - ligne 17 : si cette valeur est un type simple, elle reste inchangée ;
  - lignes 5-6 : si cette valeur est de type BaseEntity, la valeur est remplacée par son dictionnaire. On a alors un appel récursif ;
  - lignes 8-9 : si cette valeur est une liste, alors elle est remplacée par la valeur **[BaseEntity.list2list]** ;
  - lignes 11-12 : si cette valeur est un dictionnaire, alors elle est remplacée par la valeur **[BaseEntity.dict2dict]** ;

La méthode statique **[BaseEntity.list2list]** est la suivante :

```
1.      @staticmethod
2.      def list2list(liste: list) -> list:
3.          # on inspecte les éléments de la liste
4.          newlist = []
5.          for value in liste:
6.              newlist.append(BaseEntity.check_value(value))
7.          # on rend la nouvelle liste
8.          return newlist
```

- ligne 2 : la méthode reçoit une liste et rend une liste ;
- lignes 5-6 : on remplace chaque valeur de la liste reçue en paramètre par la valeur rendue par la méthode statique **[BaseEntity.check_value]**. On a donc là un appel récursif. La méthode statique **[BaseEntity.check_value]** est appelée jusqu'à ce que son paramètre **[value]** soit un type simple (pas un type BaseEntity ou list ou dict) ;

La méthode statique **[BaseEntity.dict2dict]** est la suivante :

```
1.      @staticmethod
2.      def dict2dict(dictionary: dict) -> dict:
3.          # on inspecte les éléments du dictionnaire
4.          newdict = {}
5.          for key, value in dictionary.items():
6.              newdict[key] = BaseEntity.check_value(value)
7.          # on rend le nouveau dictionnaire
8.          return newdict
```

- ligne 2 : la méthode reçoit un dictionnaire et rend un dictionnaire ;
- lignes 5-6 : on remplace chaque valeur du dictionnaire reçu en paramètre par la valeur rendue par la méthode statique **[BaseEntity.check_value]**. On a donc là un appel récursif. La méthode statique **[BaseEntity.check_value]** est appelée jusqu'à ce que son paramètre **[value]** soit un type simple (pas un type BaseEntity ou list ou dict) ;

13.2.2.2 Exemples

Le script **[asdict_01]** montre diverses utilisations de la méthode **[asdict]** :

```
1.  # on configure l'application
2.  import config
3.  config = config.configure()
4.
5.  # le syspath est configuré - on peut faire les imports
6.  from Enseignant import Enseignant
7.  from BaseEntity import BaseEntity
8.
9.  # un enseignant
10. enseignant1 = Enseignant().fromdict({"id": 1, "nom": "lourou", "prénom": "paul", "âge": 56})
11. dict1 = enseignant1.asdict()
12. print(type(dict1))
13. print(enseignant1.__dict__)
14. print(dict1)
15. print(enseignant1.asdict(excluded_keys=["_Personne__âge"]))
16. Enseignant.excluded_keys = ["_Personne__prénom"]
17. print(enseignant1)
18. # un autre enseignant
19. enseignant2 = Enseignant().fromdict({"id": 2, "nom": "abélard", "prénom": "béatrice", "âge": 57})
20. print(enseignant2.asdict())
21. print(enseignant2.asdict(included_keys=["_Personne__nom"]))
22. # une liste d'entités dans une entité
23. Enseignant.excluded_keys = []
24. entity1 = BaseEntity()
25. enseignants = [enseignant1, enseignant2]
26. setattr(entity1, "enseignants", enseignants)
27. print(entity1.asdict())
28. # un dictionnaire d'entités dans une entité
29. matières = {"maths": enseignant1, "français": enseignant2}
30. setattr(entity1, "matières", matières)
31. print(entity1.asdict())
```

Les résultats de l'exécution sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/02/asdict_01.py
2.  <class 'dict'>
3.  {'_BaseEntity__id': 1, '_Personne__nom': 'lourou', '_Personne__prénom': 'paul', '_Personne__âge': 56}
4.  {'id': 1, 'nom': 'lourou', 'prénom': 'paul', 'âge': 56}
5.  {'id': 1, 'nom': 'lourou', 'prénom': 'paul'}
6.  {"id": 1, "nom": "lourou", "âge": 56}
7.  {'id': 2, 'nom': 'abélard', 'âge': 57}
8.  {'nom': 'abélard'}
9.  {'enseignants': [{'id': 1, 'nom': 'lourou', 'prénom': 'paul', 'âge': 56}, {'id': 2, 'nom': 'abélard',
    'prénom': 'béatrice', 'âge': 57}]}
10. {'enseignants': [{'id': 1, 'nom': 'lourou', 'prénom': 'paul', 'âge': 56}, {'id': 2, 'nom': 'abélard',
    'prénom': 'béatrice', 'âge': 57}], 'matières': {'maths': {'id': 1, 'nom': 'lourou', 'prénom': 'paul',
    'âge': 56}, 'français': {'id': 2, 'nom': 'abélard', 'prénom': 'béatrice', 'âge': 57}}}
11.
12. Process finished with exit code 0
```

- la ligne 4 montre l'intérêt de la méthode **[asdict]** vis-à-vis de l'utilisation de la propriété **[__dict__]**. Les propriétés sont débarrassées du préfixe de leur classe. Cela se prête mieux à un affichage ;
- il y a plusieurs façons d'utiliser la méthode **[asdict]** :
  - on veut toutes les propriétés : on utilise la méthode **[asdict]** sans paramètres ;
  - on ne veut que certaines propriétés :
    - il y a plus de propriétés à inclure qu'à exclure : on utilisera le seul paramètre **[excluded_keys]** ;
    - il y a moins de propriétés à inclure qu'à exclure : on utilisera le seul paramètre **[included_keys]** ;

13.2.3 La méthode [BaseEntity.asjson]

Cette méthode permet d'obtenir la chaîne jSON d'un objet **[BaseEntity]** ou dérivé. Elle affiche la chaîne jSON du dictionnaire rendu par la méthode **[asdict]**. Son code est le suivant :

```
1.  def asjson(self, included_keys: list = None, excluded_keys: list = []) -> str:
2.          # la chaîne json
3.          return json.dumps(self.asdict(included_keys=included_keys, excluded_keys=excluded_keys),
    ensure_ascii=False)
```

- ligne 1 : les paramètres de la méthode **[asjson]** sont ceux de la méthode **[asdict]** ;

Voici un exemple (asjson_01) utilisant cette méthode :

```
1.  # on configure l'application
2.  import config
3.  config = config.configure()
4.
5.  # le syspath est configuré - on peut faire les imports
6.  from Enseignant import Enseignant
7.  from BaseEntity import BaseEntity
8.
9.  # un enseignant
10. enseignant1 = Enseignant().fromdict({"id": 1, "nom": "lourou", "prénom": "paul", "âge": 56})
11. print(type(enseignant1.asjson()))
12. print(enseignant1.asjson(excluded_keys=["_Personne__âge"]))
13. Enseignant.excluded_keys = ["_Personne__prénom"]
14. print(enseignant1.asjson())
15. # un autre enseignant
16. enseignant2 = Enseignant().fromdict({"id": 2, "nom": "abélard", "prénom": "béatrice", "âge": 57})
17. print(enseignant2.asjson())
18. print(enseignant2.asjson(included_keys=["_Personne__nom"]))
19. # une liste d'entités dans une entité
20. Enseignant.excluded_keys = []
21. entity1 = BaseEntity()
22. enseignants = [enseignant1, enseignant2]
23. setattr(entity1, "enseignants", enseignants)
24. print(entity1.asjson())
25. # un dictionnaire d'entités dans une entité
26. matières = {"maths": enseignant1, "français": enseignant2}
27. setattr(entity1, "matières", matières)
28. print(entity1.asjson())
```

Les résultats sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/02/asjson_01.py
2.  <class 'str'>
3.  {"id": 1, "nom": "lourou", "prénom": "paul"}
4.  {"id": 1, "nom": "lourou", "âge": 56}
5.  {"id": 2, "nom": "abélard", "âge": 57}
6.  {"nom": "abélard"}
7.  {"enseignants": [{"id": 1, "nom": "lourou", "prénom": "paul", "âge": 56}, {"id": 2, "nom": "abélard",
    "prénom": "béatrice", "âge": 57}]}
8.  {"enseignants": [{"id": 1, "nom": "lourou", "prénom": "paul", "âge": 56}, {"id": 2, "nom": "abélard",
    "prénom": "béatrice", "âge": 57}], "matières": {"maths": {"id": 1, "nom": "lourou", "prénom": "paul",
    "âge": 56}, "français": {"id": 2, "nom": "abélard", "prénom": "béatrice", "âge": 57}}}
9.
10. Process finished with exit code 0
```

La méthode **[BaseEntity.__str__]** utilise la méthode **[asjson]** pour afficher l'identité de l'objet **[BaseEntity]** ou dérivé :

```
1.  # toString
2.      def __str__(self) -> str:
3.          return self.asjson()
```

13.2.4 La méthode [BaseEntity.fromjson]

La méthode **[BaseEntity.fromjson]** permet d'initialiser un objet de type **[BaseEntity]** ou dérivé à partir d'un dictionnaire jSON. Son code est le suivant :

```
1.  def fromjson(self, json_state: str, silent: bool = False):
2.          # on met à jour l'état de l'objet à partir de la chaîne jSON
3.          return self.fromdict(json.loads(json_state), silent=silent)
```

- ligne 1 : la méthode admet deux paramètres :
  - **[json_state]** : le dictionnaire jSON qui va servir à initialiser l'objet **[BaseEntity]** ;
  - **[silent]** : pour indiquer si la présence dans le dictionnaire jSON d'une clé qui ne peut être acceptée comme propriété de l'objet **[BaseEntity]** provoque une exception (silent=False) ou est simplement ignorée (silent=True) ;
- ligne 3 : on commence par construire le dictionnaire Python image du dictionnaire jSON, puis on utilise la méthode **[fromdict]** pour initialiser l'objet **[BaseEntity]** à partir de ce dictionnaire Python ;

Voici un exemple (fromjson_01) :

```
1.  # on configure l'application
```

```
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from Enseignant import Enseignant
8.  import json
9.
10. # un enseignant
11. json1 = json.dumps({"id": 1, "nom": "lourou", "prénom": "paul", "âge": 56})
12. enseignant1 = Enseignant().fromjson(json1)
13. enseignant1.show()
```

- ligne 11 : on crée la chaîne jSON d'un dictionnaire ;
- ligne 12 : un objet **[Enseignant]** est initialisé avec cette chaîne ;
- ligne 13 : l'enseignant est affiché ;

Les résultats sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/02/fromjson_01.py
2.  Enseignant[1, paul, lourou, 56]
3.
4.  Process finished with exit code 0
```

13.2.5 Le script [main]

Le script **[main]** récapitule les différentes méthodes rencontrées :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from BaseEntity import BaseEntity
8.  from MyException import MyException
9.
10.
11. # une classe
12. class ChildEntity(BaseEntity):
13.     # attributs exclus de l'état de la classe
14.     excluded_keys = []
15.
16.     @staticmethod
17.     def get_allowed_keys():
18.         return ["att1", "att2", "att3", "att4"]
19.
20.     @property
21.     def att1(self) -> int:
22.         return self.__att1
23.
24.     @att1.setter
25.     def att1(self, value: int):
26.         if 10 >= value >= 1:
27.             self.__att1 = value
28.         else:
29.             raise MyException(1, f"L'attribut [att1] attend une valeur dans l'intervalle [1,10] ({value})")
30.
31.
32. # configuration ChildEntity
33. ChildEntity.excluded_keys = []
34. # instance ChildEntity
35. child = ChildEntity().fromdict({"att1": 1, "att2": 2})
36. # attention aux noms des propriétés
37. # ce sont ces noms qui sont utilisés dans [excluded_keys] et [included_keys]
38. print(child.__dict__)
39. # propriétés non préfixées par leur classe
40. print(child)
41.
42. # instance ChildEntity
43. try:
44.     child = ChildEntity().fromdict({"att1": 1, "att5": 5})
45.     print(child)
46. except MyException as erreur:
```

```
47.     print(erreur)
48.
49. # instance ChildEntity
50. child = ChildEntity().fromdict({"att1": 1, "att2": 2, "att3": 3, "att4": 4})
51. print(child)
52.
53. # exclusions de certaines clés de l'état des instances
54. ChildEntity.excluded_keys = ['att3']
55. print(child)
56.
57. # on exclut une clé explicitement de l'affichage
58. # elle se rajoute à celles exclues globalement au niveau de la classe
59. print(child.asdict(excluded_keys=["_ChildEntity__att1"]))
60. print(child.asjson(excluded_keys=["att2"]))
61.
62. # intérêt de la classe vis à vis du dictionnaire
63. # elle peut vérifier la validité de son contenu
64. try:
65.     child = ChildEntity().fromdict({"att1": 20})
66. except MyException as erreur:
67.     print(erreur)
68.
69. # instance ChildEntity
70. child1 = ChildEntity().fromdict({"att1": 1, "att2": 2, "att3": 3, "att4": 4})
71. # instanche ChildEntity contenant une autre instance ChildEntity
72. child2 = ChildEntity().fromdict({"att1": 10, "att2": 20, "att3": 30, "att4": child1})
73. print(child2)
74.
75. # included_keys a priorité sur excluded_keys qui sont alors ignorées
76. ChildEntity.excluded_keys = ['_ChildEntity__att1', 'att2']
77. print(child.asdict(included_keys=["_ChildEntity__att1", "att3"], excluded_keys=["att3", "att4"]))
```

Les résultats de l'exécution sont les suivants :

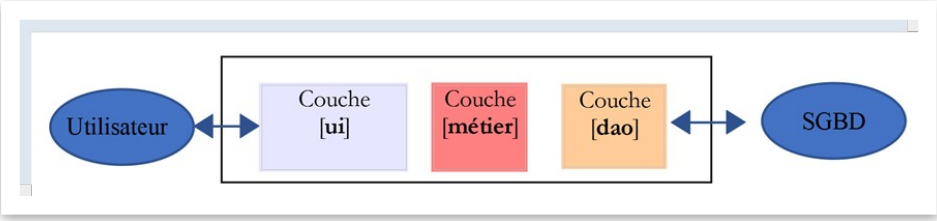
```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes/02/main.py
2.  {'_ChildEntity__att1': 1, 'att2': 2}
3.  {"att1": 1, "att2": 2}
4.  MyException[2, la clé [att5] n'est pas autorisée]
5.  {"att1": 1, "att2": 2, "att3": 3, "att4": 4}
6.  {"att1": 1, "att2": 2, "att4": 4}
7.  {'att2': 2, 'att4': 4}
8.  {"att1": 1, "att4": 4}
9.  MyException[1, L'attribut [att1] attend une valeur dans l'intervalle [1,10] (20)]
10. {"att1": 10, "att2": 20, "att4": {"att1": 1, "att2": 2, "att4": 4}}
11. {'att1': 1, 'att3': 3}
12.
13. Process finished with exit code 0
```

On prêtera attention à la ligne 2 des résultats : c'est la propriété **[ChildEntity.__dict__]** (ligne 38 du code) qui nous permet de connaître les noms des propriétés à mettre dans les listes **[included_keys]** et **[excluded_keys]**. On notera, toujours ligne 2 des résultats, que selon que la propriété est définie à l'intérieur de la classe par un getter / setter ou qu'elle a été créée comme on créerait la clé d'un dictionnaire, elle est ou pas préfixée par le nom de la classe **[ChildEntity]**.

14 Architecture en couches et programmation par interfaces

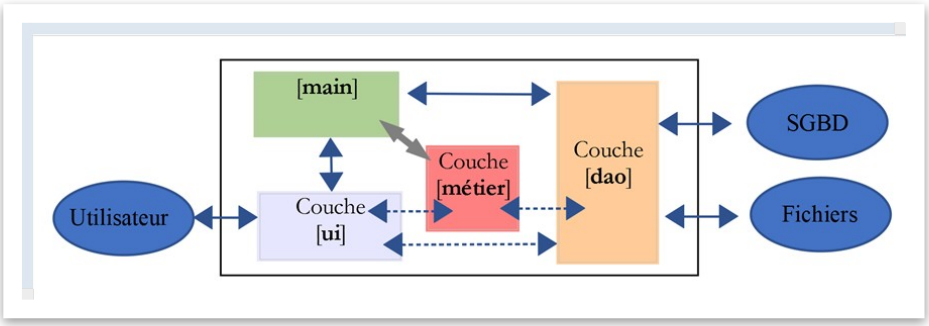
14.1 Introduction

Nous nous proposons d'écrire une application permettant l'affichage des notes des élèves d'un collège. Cette application peut avoir une architecture multicouche :



- la couche **[ui]** (**U**ser **I**nterface) est la couche en contact avec l'utilisateur de l'application ;
- la couche **[métier]** implémente les règles de gestion de l'application, tels que le calcul d'un salaire ou d'une facture. Cette couche utilise des données provenant de l'utilisateur via la couche **[présentation]** et du SGBD via la couche **[dao]** ;
- la couche **[dao]** (**D**ata **A**ccess **O**bjects) gère l'accès aux données du SGBD (Système de Gestion de Bases de Données).

C'est l'architecture qui avait été utilisée dans le |cours sur Python 2|. On peut également introduire une variante :



Les différences vis à vis de la structure en couches précédente sont les suivantes :

- un script principal appelé **[main]** ci-dessus organise l'instanciation des couches ;
- les couches **[ui, métier, dao]** ne communiquent plus forcément entre-elles. Si elles le doivent, le script **[main]** leur fournit les références des couches dont elles ont besoin ;

Le code est ici organisé en centres de compétences avec un chef d'orchestre :

- le chef d'orchestre est le script principal **[main]** ;
- les couches **[ui]**, **[dao]** et **[métier]** sont les centres de compétences ;

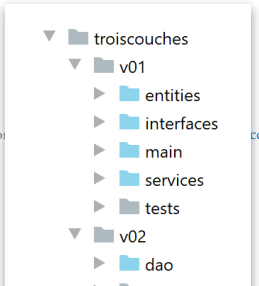
On pourrait appeler cette organisation, une organisation **orchestrale**.

14.2 Exemple 1

Nous allons illustrer l'architecture en couches avec une application console simple :

- il n'y aura pas de base de données ;
- la couche **[dao]** gèrera des entités *Elève*, *Classe*, *Matière*, *Note* permettant de gérer les notes des élèves ;
- la couche **[métier]** permettra de calculer des indicateurs sur les notes d'un élève précis ;
- la couche **[ui]** sera une application console qui affichera les résultats des élèves ;

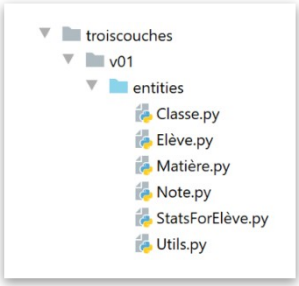
Le projet PyCharm de l'application est le suivant :



Note : les dossiers en bleu font partie des **[Sources Root]** du projet PyCharm.

14.2.1 Les entités de l'application

On appellera entités, des classes dont le seul rôle est d'encapsuler des données. On pourrait pour ce faire utiliser des dictionnaires. L'intérêt de la classe est de permettre de tester la validité des données stockées dans l'objet et de fournir une méthode fournissant l'identité de l'objet sous la forme d'une chaîne de caractères.



14.2.1.1 L'entité [Classe]

L'entité **[Classe]** (Classe.py) représente une classe du collège :

```
1.  # imports
2.  from BaseEntity import BaseEntity
3.  from MyException import MyException
4.  from Utils import Utils
5.
6.
7.  class Classe(BaseEntity):
8.      # attributs exclus de l'état de la classe
9.      excluded_keys = []
10.
11.     # propriétés de la classe
12.     @staticmethod
13.     def get_allowed_keys() -> list:
14.         # id : identifiant de la classe
15.         # nom : nom de la classe
16.         return BaseEntity.get_allowed_keys() + ["nom"]
17.
18.     # getter
19.     @property
20.     def nom(self: object) -> str:
21.         return self.__nom
22.
23.     #  setters
24.     @nom.setter
25.     def nom(self: object, nom: str):
26.         # nom doit être une chaîne de caractères non vide
27.         if Utils.is_string_ok(nom):
28.             self.__nom = nom
29.         else:
30.             raise MyException(11, f"Le nom de la classe {self.id} doit être une chaîne de caractères non
    vide")
```

Notes

- ligne 7 : l'entité **[Classe]** dérive de l'entité **[BaseEntity]** étudiée au paragraphe |La classe BaseEntity| ;
- lignes 11-16 : une classe est définie par un n° *id* et un *nom* (ligne 16). La propriété **[id]** est fournie par la classe **[BaseEntity]** et le nom par la classe **[Classe]** ;
- lignes 18-30 : getter / setter de l'attribut **[nom]** ;

14.2.1.2 L'entité [Matière]

La classe **[Matière]** (matière.py) est la suivante :

```
1.  # imports
2.  from BaseEntity import BaseEntity
3.  from MyException import MyException
4.  from Utils import Utils
5.
```

```
6.
7.  class Matière(BaseEntity):
8.      # attributs exclus de l'état de la classe
9.      excluded_keys = []
10.
11.     # propriétés de la classe
12.     @staticmethod
13.     def get_allowed_keys() -> list:
14.         # id : identifiant de la matière
15.         # nom : nom de la matière
16.         # coefficient : coefficient de la matière
17.         return BaseEntity.get_allowed_keys() + ["nom", "coefficient"]
18.
19.     # getter
20.     @property
21.     def nom(self: object) -> str:
22.         return self.__nom
23.
24.     @property
25.     def coefficient(self: object) -> float:
26.         return self.__coefficient
27.
28.     #  setters
29.     @nom.setter
30.     def nom(self: object, nom: str):
31.         # nom doit être une chaîne de caractères non vide
32.         if Utils.is_string_ok(nom):
33.             self.__nom = nom
34.         else:
35.             raise MyException(21, f"Le nom de la matière {self.id} doit être une chaîne de caractères non
    vide")
36.
37.     @coefficient.setter
38.     def coefficient(self, coefficient: float):
39.         # le coefficient doit être un réel >=0
40.         erreur = False
41.         if isinstance(coefficient, (int, float)):
42.             if coefficient >= 0:
43.                 self.__coefficient = coefficient
44.             else:
45.                 erreur = True
46.         else:
47.             erreur = True
48.         # erreur ?
49.         if erreur:
50.             raise MyException(22, f"Le coefficient de la matière {self.nom} doit être un réel >=0")
```

Notes

- ligne 7 : la classe **[Classe]** dérive de la classe **[BaseEntity]** ;
- lignes 11-17 : une matière est définie par son n° **[id]**, son nom **[nom]**, son coefficient **[coefficient]** ;
- lignes 19-50 : getters / setters des attributs de la classe ;

14.2.1.3 L'entité [Elève]

La classe **[Elève]** (élève.py) est la suivante :

```
1.  # imports
2.  from BaseEntity import BaseEntity
3.  from Classe import Classe
4.  from MyException import MyException
5.
6.  from Utils import Utils
7.
8.
9.  class Elève(BaseEntity):
10.     # attributs exclus de l'état de la classe
11.     excluded_keys = []
12.
13.     # propriétés de la classe
14.     @staticmethod
15.     def get_allowed_keys() -> list:
16.         # id : identifiant de l'élève
17.         # nom : nom de l'élève
18.         # prénom : prénom de l'élève
19.         # classe : classe de l'élève
```

```
20.         return BaseEntity.get_allowed_keys() + ["nom", "prénom", "classe"]
21.
22.     # getters
23.     @property
24.     def nom(self: object) -> str:
25.         return self.__nom
26.
27.     @property
28.     def prénom(self: object) -> str:
29.         return self.__prénom
30.
31.     @property
32.     def classe(self: object) -> Classe:
33.         return self.__classe
34.
35.     #  setters
36.     @nom.setter
37.     def nom(self: object, nom: str) -> str:
38.         # nom doit être une chaîne de caractères non vide
39.         if Utils.is_string_ok(nom):
40.             self.__nom = nom
41.         else:
42.             raise MyException(41, f"Le nom de l'élève {self.id} doit être une chaîne de caractères non
    vide")
43.
44.     @prénom.setter
45.     def prénom(self: object, prénom: str) -> str:
46.         # prénom doit être une chaîne de caractères non vide
47.         if Utils.is_string_ok(prénom):
48.             self.__prénom = prénom
49.         else:
50.             raise MyException(42, f"Le prénom de l'élève {self.id} doit être une chaîne de caractères non
    vide")
51.
52.     @classe.setter
53.     def classe(self: object, value):
54.         try:
55.             # on attend un type Classe
56.             if isinstance(value, Classe):
57.                 self.__classe = value
58.             # ou un type dict
59.             elif isinstance(value,dict):
60.                 self.__classe=Classe().fromdict(value)
61.             # ou un type json
62.             elif isinstance(value,str):
63.                 self.__classe = Classe().fromjson(value)
64.         except BaseException as erreur:
65.             raise MyException(43, f"L'attribut [{value}] de l'élève {self.id} doit être de type Classe ou
    dict ou json. Erreur : {erreur}")
```

Notes

- ligne 9 : la classe **[Elève]** dérive de la classe **[BaseEntity]** ;
- lignes 13-20 : un élève est caractérisé par son n° **[id]**, son nom **[nom]**, son prénom **[prénom]**, sa classe**[classe]**. Ce dernier paramètre est une référence sur un objet **[Classe]** ;
- lignes 22-65 : getters / setters des attributs de la classe ;

14.2.1.4 L'entité [Note]

La classe **[Note]** (note.py) est la suivante :

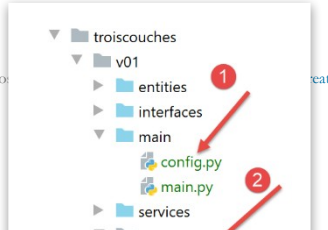
```
1.  # imports
2.  from BaseEntity import BaseEntity
3.  from Elève import Elève
4.  from Matière import Matière
5.  from MyException import MyException
6.
7.
8.  class Note(BaseEntity):
9.      # attributs exclus de l'état de la classe
10.     excluded_keys = []
11.
12.     # propriétés de la classe
13.     @staticmethod
14.     def get_allowed_keys() -> list:
15.         # id : identifiant de la note
```

```
16.         # valeur : la note elle-même
17.         # élève : élève (de type Elève) concerné par la note
18.         # matière : matière (de type Matière) concernée par la note
19.         # l'objet Note est donc la note d'un élève dans une matière
20.         return BaseEntity.get_allowed_keys() + ["valeur", "élève", "matière"]
21.
22.     # getters
23.     @property
24.     def valeur(self: object) -> float:
25.         return self.__valeur
26.
27.     @property
28.     def élève(self: object) -> Elève:
29.         return self.__élève
30.
31.     @property
32.     def matière(self: object) -> Matière:
33.         return self.__matière
34.
35.     # getters
36.     @valeur.setter
37.     def valeur(self: object, valeur: float):
38.         # la note doit être un réel entre 0 et 20
39.         if isinstance(valeur, (int, float)) and 0 <= valeur <= 20:
40.             self.__valeur = valeur
41.         else:
42.             raise MyException(31,
43.                               f"L'attribut {valeur} de la note {self.id} doit être un nombre dans
    l'intervalle [0,20]")
44.
45.     @élève.setter
46.     def élève(self: object, value):
47.         try:
48.             # on attend un type Elève
49.             if isinstance(value, Elève):
50.                 self.__élève = value
51.             # ou un type dict
52.             elif isinstance(value, dict):
53.                 self.__élève = Elève().fromdict(value)
54.             # ou un type json
55.             elif isinstance(value, str):
56.                 self.__élève = Elève().fromjson(value)
57.         except BaseException as erreur:
58.             raise MyException(32,
59.                               f"L'attribut [{value}] de la note {self.id} doit être de type Elève ou dict
    ou json. Erreur : {erreur}")
60.
61.     @matière.setter
62.     def matière(self: object, value):
63.         try:
64.             # on attend un type Matière
65.             if isinstance(value, Matière):
66.                 self.__matière = value
67.             # ou un type dict
68.             elif isinstance(value, dict):
69.                 self.__matière = Matière().fromdict(value)
70.             # ou un type json
71.             elif isinstance(value, str):
72.                 self.__matière = Matière().fromjson(value)
73.         except BaseException as erreur:
74.             raise MyException(33,
75.                               f"L'attribut [{value}] de la note {self.id} doit être de type Matière ou dict
    ou json. Erreur : {erreur}")
```

Notes

- ligne 8 : la classe **[Note]** dérive de la classe **[BaseEntity]** ;
- lignes 12-20 : un objet **[Note]** est caractérisé par son n° **[id]**, la valeur de la note **[valeur]**, une référence **[élève]** sur **l'élève** qui a cette note, une référence sur la **matière [matière]** objet de la note ;
- lignes 22-75 : getters / setters des attributs de la classe ;

14.2.2 Configuration de l'application



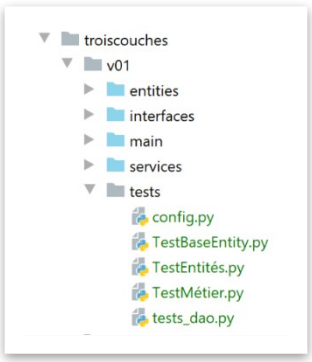
Le fichier **[config.py]** configure l'environnement du script principal **[main]** (1) ainsi que celui des tests (2). Tous ces scripts ont une instruction **[import config]** en début de code. On rappelle que le dossier contenant le script objet de la commande **[python script]** fait automatiquement partie du Python Path.Si donc **[config]** est dans le même dossier que les scripts ayant l'instruction **[import config]**, il sera trouvé. Les fichiers **[1]** et **[2]** sont ici identiques. Ce pourrait ne pas être le cas.

Le fichier **[config.sys]** est le suivant :

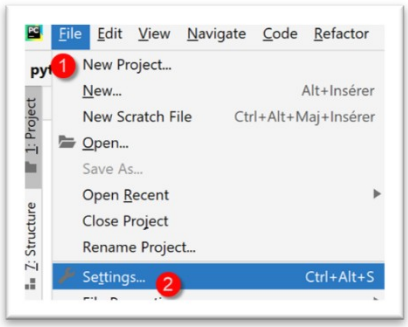
```
1.  def configure():
2.      import os
3.
4.      # chemin absolu du dossier de ce script
5.      script_dir = os.path.dirname(os.path.abspath(__file__))
6.      # root_dir
7.      root_dir="C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes"
8.      # dépendances absolues
9.      absolute_dependencies=[
10.         # les dossiers locaux contenant des classes et interfaces
11.         f"{root_dir}/02/entities",
12.         f"{script_dir}/../entities",
13.         f"{script_dir}/../interfaces",
14.         f"{script_dir}/../services",
15.     ]
16.
17.     # mise à jour du syspath
18.     from myutils import set_syspath
19.     set_syspath(absolute_dependencies)
20.
21.     # on rend la config
22.     return {}
```

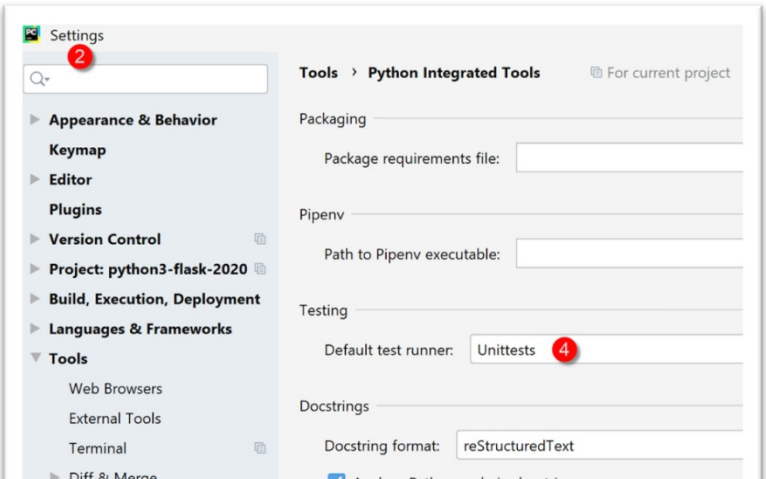
- lignes 11-14 : les dossiers qui doivent faire partie du Python Path (sys.path) ;
- le dossier **[f"{root_dir}/02/entities"]** donne accès aux classes **[BaseEntity]** et **[MyException]** ;
- le dossier **[f"{script_dir}/../entities"]** donne accès aux classes **[Elève]**, **[Classe]**, **[Matière]**, **[Note]** ;
- le dossier **[f"{script_dir}/../interfaces",]** donne accès aux interfaces de l'application ;
- le dossier **[f"{script_dir}/../services"]** donne accès aux classes implémentant les interfaces ;

14.2.3 Tests des entités

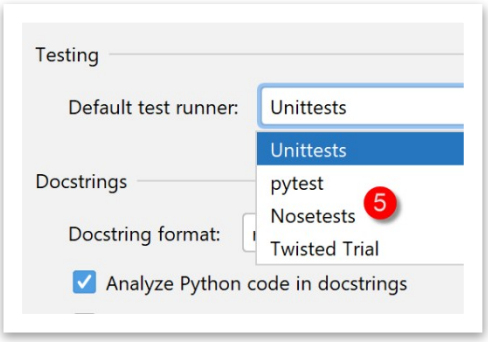


Nous allons ici écrire des tests exécutés par un outil appelé **[unittest]**. PyCharm vient avec plusieurs frameworks de test. Le choix de l'un d'eux se fait dans la configuration de PyCharm :





- en **[4]**, plusieurs frameworks de test sont disponibles :



14.2.3.1 La classe de tests [TestBaseEntity]

Le script de test **[TestBaseEntity]** sera le suivant :

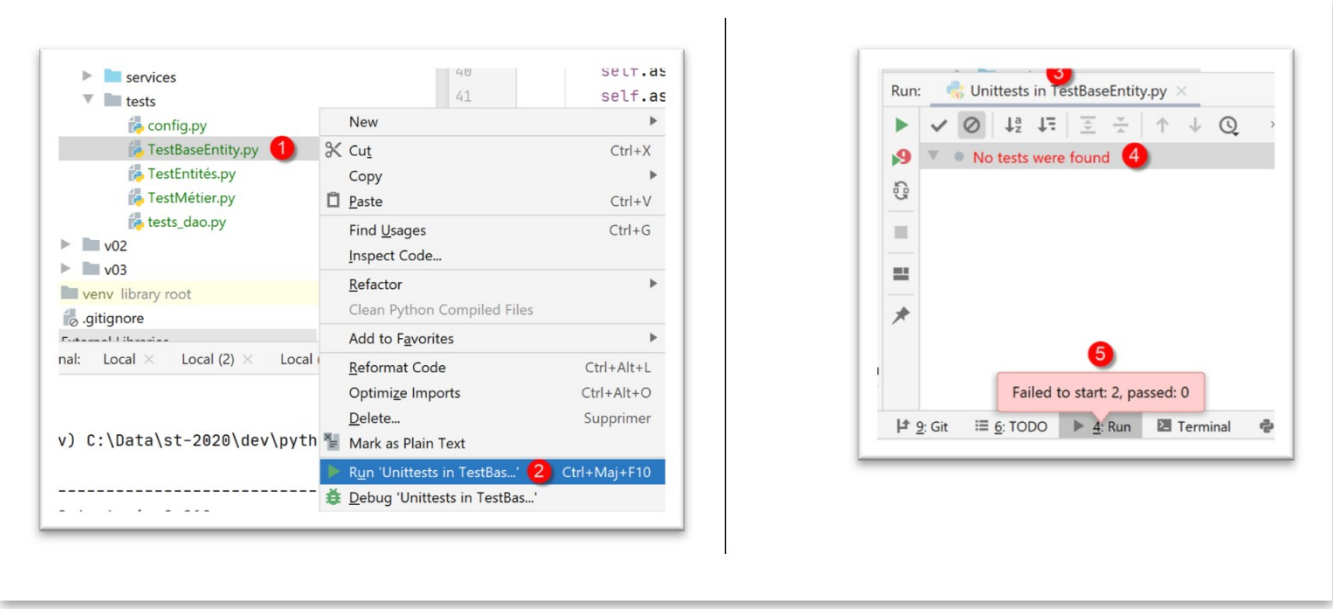
```
1.  import unittest
2.
3.  # on configure l'application
4.  import config
5.
6.  config = config.configure()
7.
8.
9.  class TestBaseEntity (unittest.TestCase):
10.
11.     def test_note1(self):
12.         # imports
13.         from Note import Note
14.         from Elève import Elève
15.         from Classe import Classe
16.         from Matière import Matière
17.         # construction d'une note à partir d'une chaîne jSON
18.         note = Note().fromjson(
19.             '{"id": 8, "valeur": 12, "élève": {"id": 42, "nom": "nom4", "prénom": "prénom4", "classe":
    {"id": 2, "nom": "classe2"}}, "matière": {"id": 2, "nom": "matière2", "coefficient": 2}}')
20.         # vérifications
21.         self.assertIsInstance(note, Note)
22.         self.assertIsInstance(note.élève, Elève)
23.         self.assertIsInstance(note.élève.classe, Classe)
24.         self.assertIsInstance(note.matière, Matière)
25.
26.
27.     def test_note2(self):
28.         # imports
29.         from Note import Note
30.         from Elève import Elève
31.         from Classe import Classe
32.         from Matière import Matière
33.         # construction d'une note à partir d'un dictionnaire
34.         note = Note().fromdict(
35.             {"id": 8, "valeur": 12, "élève": {"id": 42, "nom": "nom4", "prénom": "prénom4",
36.                                               "classe": {"id": 2, "nom": "classe2"}},
37.              "matière": {"id": 2, "nom": "matière2", "coefficient": 2}})
38.         # vérifications
39.         self.assertIsInstance(note, Note)
40.         self.assertIsInstance(note.élève, Elève)
41.         self.assertIsInstance(note.élève.classe, Classe)
42.         self.assertIsInstance(note.matière, Matière)
43.
44. if __name__ == '__main__':
45.     unittest.main()
```

Notes

- ligne 1 : on importe le module **[unittest]** qui va fournir les différentes méthodes de test ;
- lignes 3-6 : on configure l'application pour que les classes nécessaires aux tests soient trouvées ;
- ligne 9 : une classe de test **[unittest]** doit étendre la classe **[unittest.TestCase]** ;
- lignes 11, 27 : les fonctions de test doivent avoir un nom commençant par **[test]** sinon elles ne seront pas reconnues ;

- lignes 13-16 : on importe les classes dont on a besoin ;
- dans cette classe de test, on veut vérifier le comportement des méthodes **[BaseEntity.fromdict]** (ligne 34) et **[BaseEntity.fromjson]** (ligne 18). La classe **[Note]** a des propriétés qui sont des références à d'autres classes. On veut vérifier que les deux méthodes précédentes créent des objets **[Note]** valides ;
- ligne 18 : on crée un objet **[Note]** à partir d'un objet jSON ;
- ligne 21 : on vérifie que l'objet créé est bien de type **[Note]**. La méthode **[assertIsInstance]** est une méthode de la classe **[unittest.TestCase]**, classe parent de la classe **[TestBaseEntity]** ;
- ligne 22 : on vérifie que **[note.élève]** est bien de type **[Elève]** ;
- ligne 23 : on vérifie que **[note.élève.classe]** est bien de type **[Classe]** ;
- ligne 24 : on vérifie que **[note.matière]** est bien de type **[Matière]** ;
- lignes 33-42 : on fait de même avec la méthode **[BaseEntity.fromdict]** ;

Il y a plusieurs façon d'exécuter les tests :



- en **[1-2]**, on exécute **[TestBaseEntity]** avec le framework **[UnitTest]** ;
- en **[3-5]**, les tests échouent. **[UnitTests]** indique qu'il n'a trouvé aucun test à exécuter ;

L'échec des tests vient de l'organisation du code de **[TestBaseEntity]** :

```
1.  import unittest
2.
3.  # on configure l'application
4.  import config
5.
6.  config = config.configure()
7.
8.
9.  class TestBaseEntity(unittest.TestCase):
```

Ce qui gêne le framework **[UnitTest]**, c'est la présence de code exécutable, lignes 3-6, avant la définition de la classe de test, ligne 9.

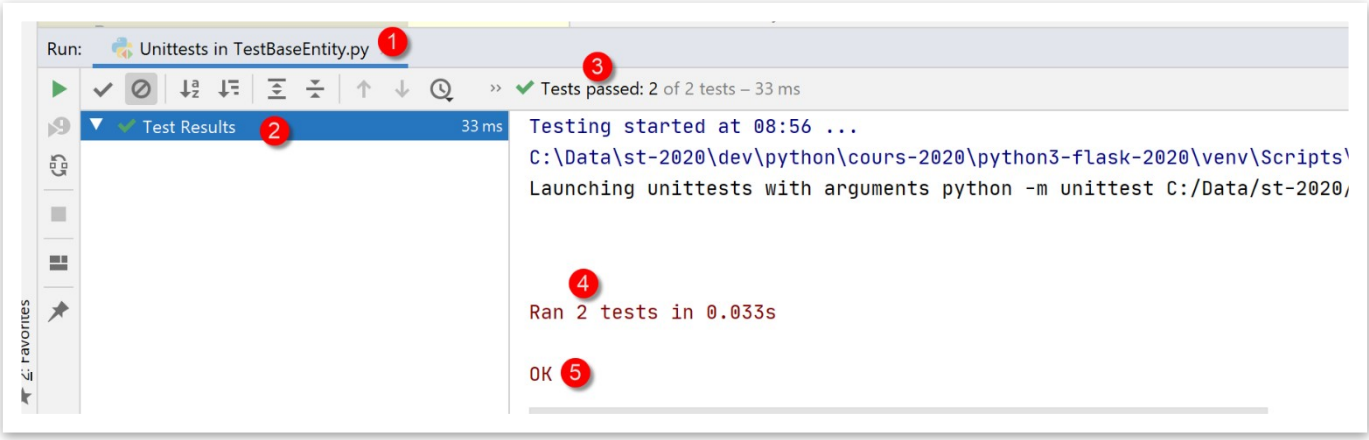
On réorganise alors le code de la façon suivante :

```
1.  import unittest
2.
3.
4.  class TestBaseEntity(unittest.TestCase):
5.
6.      def setUp(self):
7.          # on configure l'application
8.          import config
9.
10.         config.configure()
11.
12.     def test_note1(self):
13.         …
14.
15.     def test_note2(self):
```

```
16.         …
17.
18.
19. if __name__ == '__main__':
20.     unittest.main()
```

- lignes 6-10 : on définit une fonction **[setUp]**. Cette fonction a un rôle particulier : elle est exécutée avant chaque fonction de test (test_note1, test_note2) ;

Ceci fait, l'exécution de la classe **[TestBaseEntity]** donne les résultats suivants :



Cette fois-ci les deux méthodes de test ont été exécutées et les tests ont été réussis.

Voyons ce qui se passe lorsqu'un test échoue. Modifions le code de **[test_note1]** de la façon suivante :

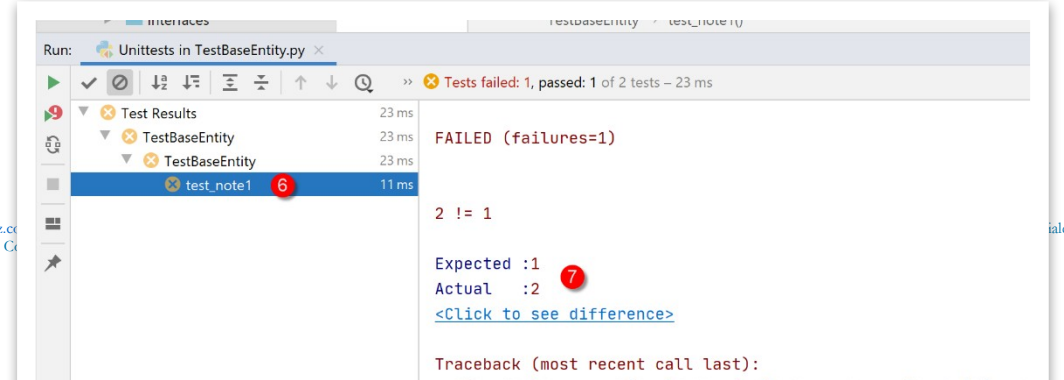
```
1.     def test_note1(self):
2.         # erreur volontaire - on vérifie que 1==2
3.         self.assertEqual(1,2)
4.         # imports
5.         from Note import Note
6.  …
```

- ligne 2 : on vérifie que 1==2 ;

Les résultats de l'exécution sont alors les suivants :

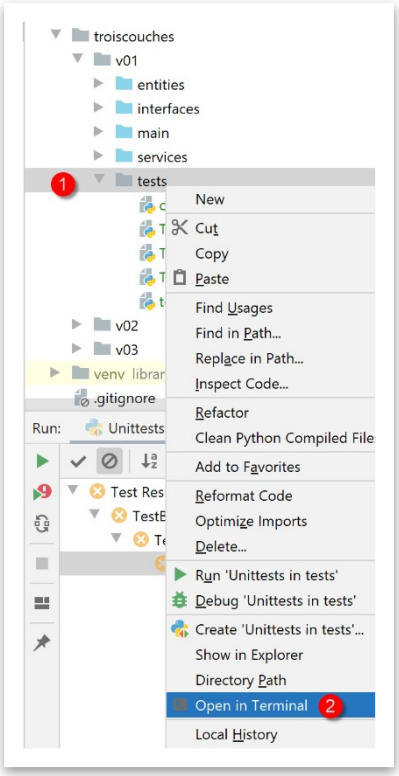


On peut connaître la cause de l'erreur en cliquant sur le test raté **[2]** :



- en **[7-8]**, la cause de l'erreur ;

Une autre façon d'exécuter une classe de tests est de l'exécuter dans un terminal :




```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\troiscouches\v01\tests>python -m unittest
    TestBaseEntity.py
2.  ..
3.  ----------------------------------------------------------------------
4.  Ran 2 tests in 0.026s
5.
6.  OK
```

La ligne 6 indique que les deux tests ont réussi (on a enlevé l'erreur 1==2) ;

Enfin une troisème façon d'exécuter la classe de tests **[TestBaseEntity]**, toujours dans un terminal, est la suivante. On termine la classe de tests avec les lignes 6-7 suivantes ;

```
1.  …
2.          self.assertIsInstance(note.élève.classe, Classe)
3.          self.assertIsInstance(note.matière, Matière)
4.
5.
6.  if __name__ == '__main__':
7.      unittest.main()
```

- ligne 6 : la variable **[__name__]** est le nom donné au script qui s'exécute. Lorsque le script est le script lancé par la commande **[python script.py]**, la variable **[__name__]** vaut **[__main__]** (2 caractères soulignés avant et après l'identifiant). Ainsi la ligne 7 n'est exécutée que lorsque le script **[TestBaseEntity]** est lancé par la commande **[python TestBaseEntity.py]**. L'instruction **[unittest.main()]** lance l'exécution du script par le framework **[UnitTest]**. Voici un exemple :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\troiscouches\v01\tests>python
    TestBaseEntity.py
2.  ..
3.  ----------------------------------------------------------------------
4.  Ran 2 tests in 0.013s
5.
6.  OK
```

14.2.3.2 La classe de tests `[TestEntités]`

La classe de tests **`[TestEntités]`** est la suivante :

```
import unittest


class TestEntités(unittest.TestCase):
    def setUp(self):
        # on configure l'application
        import config

        config.configure()

    def test_code1a(self):
        # imports
        from Elève import Elève
        from MyException import MyException
        # code d'erreur
        code = None
        try:
            # id invalide
            Elève().fromdict({"id": "x", "nom": "y", "prénom": "z", "classe": "t"})
        except MyException as ex:
            print(f"\ncode erreur={ex.code}, message={ex}")
            code = ex.code
        # vérification
        self.assertEqual(code, 1)

    def test_code41(self):
        # imports
        from Elève import Elève
        from MyException import MyException
        # code d'erreur
        code = None

        try:
            # nom invalide
            Elève().fromdict({"id": 1, "nom": "", "prénom": "z", "classe": "t"})
        except MyException as ex:
            print(f"\ncode erreur={ex.code}, message={ex}")
            code = ex.code
        # vérification
        self.assertEqual(code, 41)

    def test_code42(self):
        # imports
        from Elève import Elève
        from MyException import MyException
        # code d'erreur
        code = None
        try:
            # prénom invalide
            Elève().fromdict({"id": 1, "nom": "y", "prénom": "", "classe": "t"})
        except MyException as ex:
            print(f"\ncode erreur={ex.code}, message={ex}")
            code = ex.code
        # vérification
        self.assertEqual(code, 42)

    def test_code43(self):
        # imports
        from Elève import Elève
        from MyException import MyException
        # code d'erreur
        code = None
        try:
            # classe invalide
            Elève().fromdict({"id": 1, "nom": "y", "prénom": "z", "classe": "t"})
        except MyException as ex:
            print(f"\ncode erreur={ex.code}, message={ex}")
            code = ex.code
        # vérification
        self.assertEqual(code, 43)

    def test_code1b(self):
        # imports
```

```
74.         from Classe import Classe
75.         from MyException import MyException
76.         # code d'erreur
77.         code = None
78.         try:
79.             # identifiant invalide
80.             Classe().fromdict({"id": "x", "nom": "y"})
81.         except MyException as ex:
82.             print(f"\ncode erreur={ex.code}, message={ex}")
83.             code = ex.code
84.         # vérification
85.         self.assertEqual(code, 1)
86.
87.     def test_code11(self):
88.         # imports
89.         from Classe import Classe
90.         from MyException import MyException
91.
92.         # code d'erreur
93.         code = None
94.         try:
95.             # nom invalide
96.             Classe().fromdict({"id": 1, "nom": ""})
97.         except MyException as ex:
98.             code = ex.code
99.         # vérification
100.        self.assertEqual(code, 11)
101.
102.    def test_code1c(self):
103.        # imports
104.        from Matière import Matière
105.        from MyException import MyException
106.
107.        # code d'erreur
108.        code = None
109.        try:
110.            # identifiant invalide
111.            Matière().fromdict({"id": "x", "nom": "y", "coefficient": "t"})
112.        except MyException as ex:
113.            print(f"\ncode erreur={ex.code}, message={ex}")
114.            code = ex.code
115.        # vérification
116.        self.assertEqual(code, 1)
117.
118.    def test_code21(self):
119.        # imports
120.        from Matière import Matière
121.        from MyException import MyException
122.        # code d'erreur
123.        code = None
124.        try:
125.            # nom invalide
126.            Matière().fromdict({"id": "1", "nom": "", "coefficient": "t"})
127.        except MyException as ex:
128.            print(f"\ncode erreur={ex.code}, message={ex}")
129.            code = ex.code
130.        # vérification
131.        self.assertEqual(code, 21)
132.
133.    def test_code22(self):
134.        # imports
135.        from Matière import Matière
136.        from MyException import MyException
137.        # code d'erreur
138.        code = None
139.        try:
140.            # coefficient invalide
141.            Matière().fromdict({"id": 1, "nom": "y", "coefficient": "t"})
142.        except MyException as ex:
143.            print(f"\ncode erreur={ex.code}, message={ex}")
144.            code = ex.code
145.        # vérification
146.        self.assertEqual(code, 22)
147.
148.    def test_code1d(self):
149.        # imports
150.        from Note import Note
```

```
        from MyException import MyException
        # code d'erreur
        code = None
        try:
            # identifiant invalide
            Note().fromdict({"id": "x", "valeur": "x", "élève": "y", "matière": "z"})
        except MyException as ex:
            print(f"\ncode erreur={ex.code}, message={ex}")
            code = ex.code
        # vérification
        self.assertEqual(code, 1)

    def test_code31(self):
        # imports
        from Note import Note
        from MyException import MyException

        # code d'erreur
        code = None
        try:
            # valeur invalide
            Note().fromdict({"id": 1, "valeur": "x", "élève": "y", "matière": "z"})
        except MyException as ex:
            print(f"\ncode erreur={ex.code}, message={ex}")
            code = ex.code
        # vérification
        self.assertEqual(code, 31)

    def test_code32(self):
        # imports
        from Note import Note
        from MyException import MyException

        # code d'erreur
        code = None
        try:
            # élève invalide
            Note().fromdict({"id": 1, "valeur": 10, "élève": "y", "matière": "z"})
        except MyException as ex:
            print(f"\ncode erreur={ex.code}, message={ex}")
            code = ex.code
        # vérification
        self.assertEqual(code, 32)

    def test_code33(self):
        # imports
        from Elève import Elève
        from Note import Note
        from Classe import Classe
        from MyException import MyException

        # code d'erreur
        code = None
        try:
            # matière invalide
            classe = Classe().fromdict({"id": 1, "nom": "x"})
            élève = Elève().fromdict({"id": 1, "nom": "a", "prénom": "b", "classe": classe})
            Note().fromdict({"id": 1, "valeur": 10, "élève": élève, "matière": "z"})
        except MyException as ex:
            print(f"\ncode erreur={ex.code}, message={ex}")
            code = ex.code
        # vérification
        self.assertEqual(code, 33)

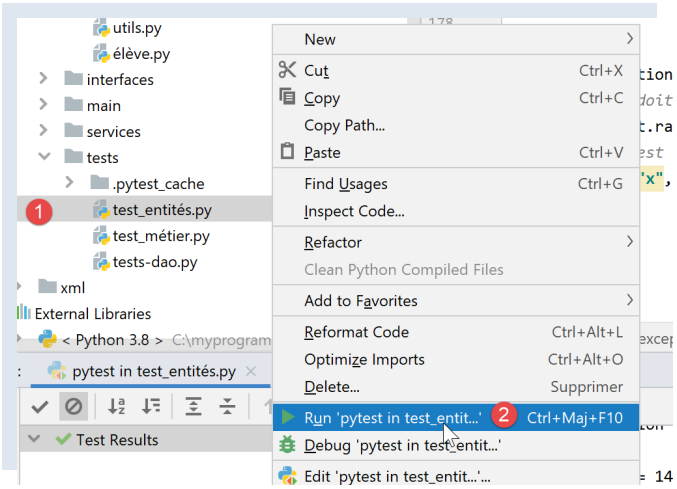
    def test_exception(self):
        # imports
        from Elève import Elève
        # le test doit lancer le type [MyException] pour réussir
        from MyException import MyException
        with self.assertRaises(MyException):
            # le test
            Elève().fromdict({"id": "x", "nom": "y", "prénom": "z", "classe": "t"})


if __name__ == '__main__':
    unittest.main()
```

- le script de test a pour but de tester les setters des classes : vérifier qu'on ne peut pas attribuer des valeurs incorrectes aux attributs des différentes entités ;
- lignes 11-24 : on teste qu'on ne peut pas passer un identifiant invalide à un élève. Parce qu'on passe la valeur 'x', ligne 16, comme identifiant de l'élève, on s'attend à avoir une exception. On devrait donc passer dans les lignes 20-22 ;
- ligne 21 : affichage du message d'erreur ;
- ligne 22 : on récupère le code de l'erreur (cf paragraphe |L'entité MyException|) ;
- ligne 24 : on vérifie (assert) que le code d'erreur est 1. Ici, on vérifie deux choses :
  - qu'il y a bien eu erreur ;
  - que le code d'erreur est 1 ;
- ce processus est répété avec les fonctions des lignes 24-213 ;
- lignes 215-222 : on teste qu'une action lève une exception d'un certain type ;
- ligne 220 : on indique que le test est réussi s'il lève une exception de type **[MyException]** ;

Résultats

On exécute le script de test :



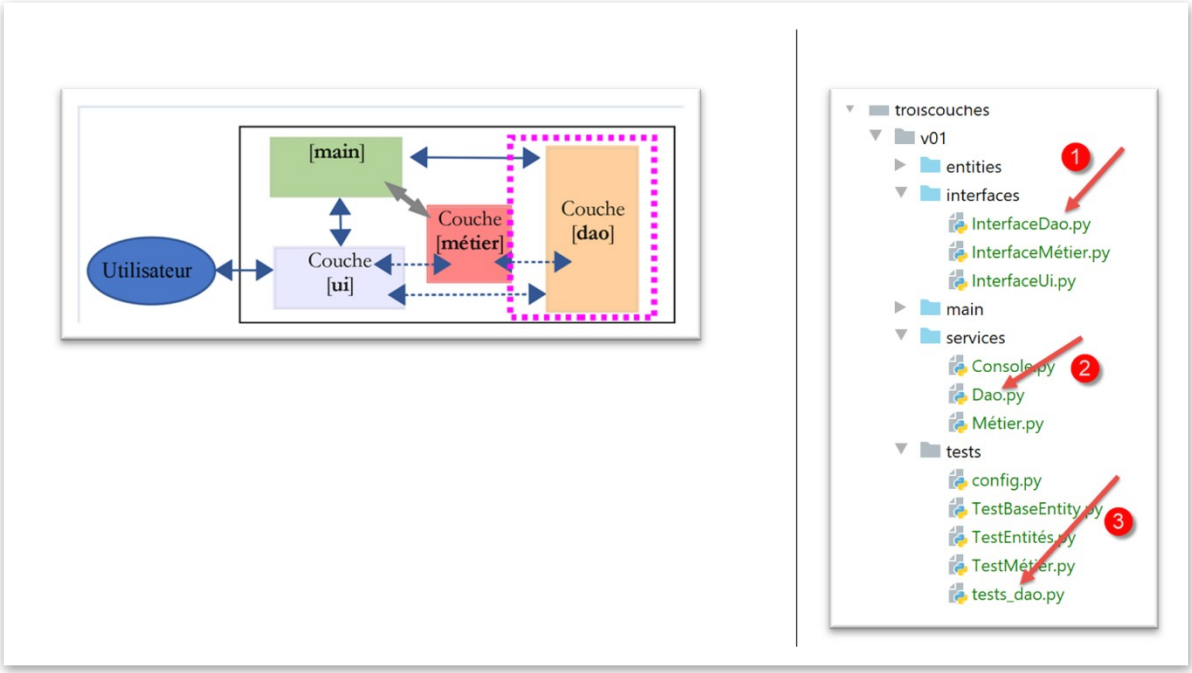
Les résultats obtenus sont les suivants :

```
1.  Testing started at 09:39 ...
2.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe "C:\Program
    Files\JetBrains\PyCharm Community Edition 2020.1.2\plugins\python-
    ce\helpers\pycharm\_jb_unittest_runner.py" --path C:/Data/st-2020/dev/python/cours-2020/python3-
    flask-2020/troiscouches/v01/tests/TestEntités.py
3.  Launching unittests with arguments python -m unittest C:/Data/st-2020/dev/python/cours-2020/python3-
    flask-2020/troiscouches/v01/tests/TestEntités.py in C:\Data\st-2020\dev\python\cours-2020\python3-
    flask-2020\troiscouches\v01\tests
4.
5.
6.  code erreur=1, message=MyException[1, L'identifiant d'une entité <class 'Elève.Elève'> doit être un entier
    >=0]
7.
8.  code erreur=1, message=MyException[1, L'identifiant d'une entité <class 'Classe.Classe'> doit être un
    entier >=0]
9.
10. code erreur=1, message=MyException[1, L'identifiant d'une entité <class 'Matière.Matière'> doit être un
    entier >=0]
11.
12. code erreur=1, message=MyException[1, L'identifiant d'une entité <class 'Note.Note'> doit être un entier
    >=0]
13.
14. code erreur=21, message=MyException[21, Le nom de la matière 1 doit être une chaîne de caractères non vide]
15.
16. code erreur=22, message=MyException[22, Le coefficient de la matière y doit être un réel >=0]
17.
18. code erreur=31, message=MyException[31, L'attribut x de la note 1 doit être un nombre dans l'intervalle
    [0,20]]
19.
20. code erreur=32, message=MyException[32, L'attribut [y] de la note 1 doit être de type Elève ou dict ou
    json. Erreur : Expecting value: line 1 column 1 (char 0)]
21.
```

```
22. code erreur=33, message=MyException[33, L'attribut [z] de la note 1 doit être de type Matière ou dict ou
    json. Erreur : Expecting value: line 1 column 1 (char 0)]
23.
24. code erreur=41, message=MyException[41, Le nom de l'élève 1 doit être une chaîne de caractères non vide]
25.
26. code erreur=42, message=MyException[42, Le prénom de l'élève 1 doit être une chaîne de caractères non vide]
27.
28. code erreur=43, message=MyException[43, L'attribut [t] de l'élève 1 doit être de type Classe ou dict ou
    json. Erreur : Expecting value: line 1 column 1 (char 0)]
29.
30.
31. Ran 14 tests in 0.040s
32.
33. OK
34.
35. Process finished with exit code 0
```

Ici, tous les tests ont été réussis

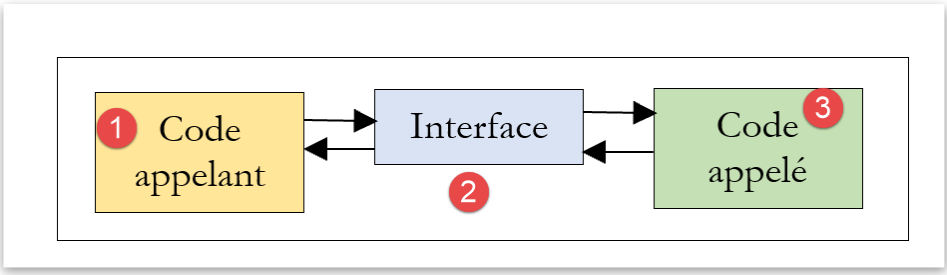
14.2.4 La couche [dao]



La couche **[dao]** implémente l'interface **[InterfaceDao] [1]**. Celle-ci est implémentée par la classe **[Dao]** (2). Le script **[tests_dao]** (3) teste les méthodes de la couche **[dao]**.

14.2.4.1 Interface [InterfaceDao]

Une interface est un contrat passé entre code appelant et code appelé. C'est le code appelé qui offre l'interface :



- le code appelant **[1]** ne connait pas l'implémentation du code appelé **[3]**. Il ne connaît que la façon de l'appeler. C'est l'interface **[2]** qui le lui indique. Celle-ci définit un certain nombre de méthodes / fonctions à utiliser pour interagir avec le code appelé. On appelle également API (Application Programming Interface) cette interface ;

La couche **[dao]** offrira l'interface suivante :

- **[get_classes]** rend la liste des classes du collège ;
- **[get_matières]** rend la liste des matières enseignées au collège ;
- **[get_élèves]** rend la liste des élèves du collège ;
- **[get_notes]** rend la liste des notes de tous les élèves ;
- **[get_notes_for_élève_by_id]** rend les notes d'un élève particulier ;
- **[get_élève_by_id]** rend un élève repéré par son n° ;

Le code appelant n'utilisera que ces méthodes. Il n'a pas à savoir comment elles sont implémentées. Les données peuvent alors provenir de différentes sources (en dur, d'une base de données, de fichiers texte…) sans que cela impacte le code appelant. On appelle cela la **programmation par interfaces**.

Python 3 a une notion qui s'approche de celle d'interface : la **classe abstraite**. Nous allons l'utiliser. Nous allons regrouper les interfaces de cet exemple dans le dossier **[interfaces]**.

Nous définissons une classe abstraite **[InterfaceDao]** (InterfaceDao.py) pour la couche **[dao]** :

```
1.  # imports
2.  from abc import ABC, abstractmethod
3.
4.  # interface Dao
5.  from Elève import Elève
6.
7.
8.  class InterfaceDao(ABC):
9.      # liste des classes
10.     @abstractmethod
11.     def get_classes(self: object) -> list:
12.         pass
13.
14.     # liste des élèves
15.     @abstractmethod
16.     def get_élèves(self: object) -> list:
17.         pass
18.
19.     # liste des matières
20.     @abstractmethod
21.     def get_matières(self: object) -> list:
22.         pass
23.
24.     # liste des notes
25.     @abstractmethod
26.     def get_notes(self: object) -> list:
27.         pass
28.
29.     # liste des notes d'un élève
30.     @abstractmethod
31.     def get_notes_for_élève_by_id(self: object, élève_id: int) -> list:
32.         pass
33.
34.     # chercher un élève par son id
35.     @abstractmethod
36.     def get_élève_by_id(self, élève_id: int) -> Elève:
37.         pass
38.
```

Notes :

- ligne 2 : ABC=Abstract Base Class. On importe du module **[abc]**, la classe ABC ainsi que le décorateur **[abstractmethod]** utilisé aux lignes 10, 15, 20, 25, 30 et 35 ;
- ligne 8 : la classe abstraite s'appelle **[InterfaceDao]** et dérive de la classe **[ABC]** ;
- les méthodes de la classe abstraite sont décorées avec le décorateur **[@abstractmethod]** qui fait de la méthode ainsi décorée une méthode abstraite : son code n'est pas défini. Néanmoins, on y met du code : l'instruction **[pass]** qui ne fait rien ;
- la classe abstraite **[InterfaceDao]** ne peut être instanciée. Seules peuvent l'être les classes **dérivées** de **[InterfaceDao]** ayant implémenté toutes les méthodes de **[InterfaceDao]**. Si donc on crée deux classes **[Dao1]** et **[Dao2]** dérivant de la classe **[InterfaceDao]**, elles implémenteront toutes deux les méthodes abstraites de **[InterfaceDao]**. On pourrait dire ainsi qu'elles implémentent l'interface **[InterfaceDao]** ;
- les langages implémentant à la fois les interfaces et les classes abstraites donnent à l'interface un rôle différent de celui de la classe abstraite. Une interface n'a pas d'attributs et ne peut être instanciée. Une classe peut implémenter une interface en définissant toutes les méthodes de celle-ci ;

14.2.4.2 Implémentation [Dao]

La classe **[Dao]** (dao.py) implémente l'interface **[InterfaceDao]** de la façon suivante :

```
1.  # import des entités et des interfaces
2.  from Classe import Classe
3.  from Elève import Elève
4.  from InterfaceDao import InterfaceDao
5.  from Matière import Matière
6.  from MyException import MyException
7.  from Note import Note
8.
9.
10. # couche [dao] implémente l'interface InterfaceDao
11. class Dao(InterfaceDao):
12.     # constructeur
13.     # on construit des listes en dur
14.     def __init__(self):
15.         # on instancie les classes
16.         classe1 = Classe().fromdict({"id": 1, "nom": "classe1"})
17.         classe2 = Classe().fromdict({"id": 2, "nom": "classe2"})
18.         self.classes = [classe1, classe2]
19.         # les matières
20.         matière1 = Matière().fromdict({"id": 1, "nom": "matière1", "coefficient": 1})
21.         matière2 = Matière().fromdict({"id": 2, "nom": "matière2", "coefficient": 2})
22.         self.matières = [matière1, matière2]
23.         # les élèves
24.         élève11 = Elève().fromdict({"id": 11, "nom": "nom1", "prénom": "prénom1", "classe": classe1})
25.         élève21 = Elève().fromdict({"id": 21, "nom": "nom2", "prénom": "prénom2", "classe": classe1})
26.         élève32 = Elève().fromdict({"id": 32, "nom": "nom3", "prénom": "prénom3", "classe": classe2})
27.         élève42 = Elève().fromdict({"id": 42, "nom": "nom4", "prénom": "prénom4", "classe": classe2})
28.         self.élèves = [élève11, élève21, élève32, élève42]
29.         # les notes des élèves dans les différentes matières
30.         note1 = Note().fromdict({"id": 1, "valeur": 10, "élève": élève11, "matière": matière1})
31.         note2 = Note().fromdict({"id": 2, "valeur": 12, "élève": élève21, "matière": matière1})
32.         note3 = Note().fromdict({"id": 3, "valeur": 14, "élève": élève32, "matière": matière1})
33.         note4 = Note().fromdict({"id": 4, "valeur": 16, "élève": élève42, "matière": matière1})
34.         note5 = Note().fromdict({"id": 5, "valeur": 6, "élève": élève11, "matière": matière2})
35.         note6 = Note().fromdict({"id": 6, "valeur": 8, "élève": élève21, "matière": matière2})
36.         note7 = Note().fromdict({"id": 7, "valeur": 10, "élève": élève32, "matière": matière2})
37.         note8 = Note().fromdict({"id": 8, "valeur": 12, "élève": élève42, "matière": matière2})
38.         self.notes = [note1, note2, note3, note4, note5, note6, note7, note8]
39.
40.     # -----------
41.     # interface IDao
42.     # -----------
43.
44.
```

Notes :

- lignes 1-7 : on importe les entités et l'interface **[InterfaceDao]** ;
- ligne 11 : la classe **[Dao]** dérive de la classe abstraite **[InterfaceDao]**. Nous dirons qu'elle implémente l'interface **[InterfaceDao]** ;
- ligne 14 : le constructeur n'a pas de paramètres. Il construit en dur quatre listes :
  - lignes 15-18 : la liste des classes ;
  - lignes 19-22 : la liste des matières ;
  - lignes 23-28 : la liste des élèves ;
  - lignes 29-38 : la liste des notes ;
- lignes 40-44 : implémentation des méthodes de l'interface **[Interface Dao]**. Ici, nous ne les définissons pas pour voir le message d'erreur émis par Python ;

Un programme de test pourrait être le suivant **[tests-dao.py]** :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # instanciation couche [dao]
7.  from Dao import Dao
8.
9.  daoImpl = Dao()
10.
```

```
11. # liste des classes
12. for classe in daoImpl.get_classes():
13.     print(classe)
14.
15. # liste des matières
16. for matière in daoImpl.get_matières():
17.     print(matière)
18.
19. # liste des classes
20. for élève in daoImpl.get_élèves():
21.     print(élève)
22.
23. # liste des notes
24. for note in daoImpl.get_notes():
25.     print(note)
```

Note : le script **[tests-dao.py]** n'est pas un test **[unittest]** car il ne contient pas de méthodes avec un nom commençant par **[test_]**.

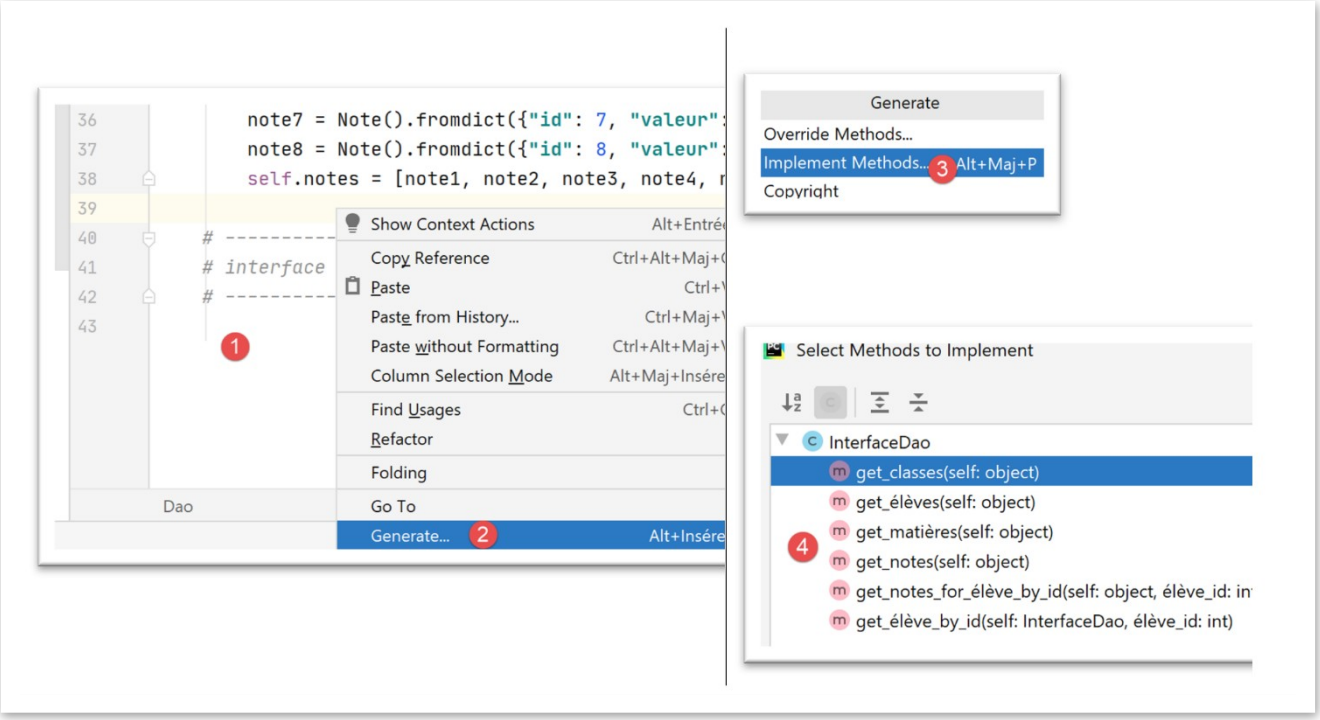
Les commentaires se suffisent à eux-mêmes. Les lignes 11-25 utilisent l'interface de la couche **[dao]**. Il n'y a pas là d'hypothèses sur l'implémentation réelle de la couche. Ligne 9, on instancie la couche **[dao]**.

Les résultats de l'exécution de ce script sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/troiscouches/v01/tests/tests_dao.py
2.  Traceback (most recent call last):
3.    File "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/troiscouches/v01/tests/tests_dao.py", line
    9, in <module>
4.      daoImpl = Dao()
5.  TypeError: Can't instantiate abstract class Dao with abstract methods get_classes, get_matières, get_notes,
    get_notes_for_élève_by_id, get_élève_by_id, get_élèves
6.
7.  Process finished with exit code 1
```

On voit qu'une erreur se produit dès l'instanciation de la classe **[Dao]** (ligne 3 ci-dessus). L'interpréteur Python 3 nous dit qu'il ne peut instancier la classe, ceci parce que nous n'avons pas défini les méthodes abstraites **[get_classes, get_matières, get_notes, get_notes_for_élève_by_id, get_élève_by_id, get_élèves]**.

Pycharm a également la notion de classe abstraite et nous propose de définir les méhodes de celle-ci :



- en **[1]**, cliquer droit sur le code ;

- en **[2-3]**, choisir **[Generate / Implement Methods]** pour implémenter les méthodes manquantes de la classe **[Dao]** ;
- en **[4]**, choisir les méthodes à implémenter, ici toutes ;

Ceci fait, la classe **[Dao]** est complétée par PyCharm de la façon suivante :

```
1.      # -----------
2.      # interface IDao
3.      # -----------
4.
5.      def get_classes(self: object) -> list:
6.          pass
7.
8.      def get_élèves(self: object) -> list:
9.          pass
10.
11.     def get_matières(self: object) -> list:
12.         pass
13.
14.     def get_notes(self: object) -> list:
15.         pass
16.
17.     def get_notes_for_élève_by_id(self: object, élève_id: int) -> list:
18.         pass
19.
20.     def get_élève_by_id(self, élève_id: int) -> Elève:
21.         pass
```

Nous complétons la classe **[Dao]** de la façon suivante :

```
1.      # -----------
2.      # interface IDao
3.      # -----------
4.
5.      # liste des classes
6.      def get_classes(self) -> list:
7.          return self.classes
8.
9.      # liste des matières
10.     def get_matières(self) -> list:
11.         return self.matières
12.
13.     # liste des élèves
14.     def get_élèves(self) -> list:
15.         return self.élèves
16.
17.     # liste des notes
18.     def get_notes(self) -> list:
19.         return self.notes
20.
21.     def get_notes_for_élève_by_id(self, élève_id: int) -> dict:
22.         # on recherche l'élève
23.         élève = self.get_élève_by_id(élève_id)
24.         # on récupère ses notes
25.         notes = list(filter(lambda n: n.élève.id == élève_id, self.get_notes()))
26.         # on rend le résultat
27.         return {"élève": élève, "notes": notes}
28.
29.     def get_élève_by_id(self, élève_id: int) -> Elève:
30.         # on filtre les élèves
31.         élèves = list(filter(lambda e: e.id == élève_id, self.get_élèves()))
32.         # trouvé ?
33.         if not élèves:
34.             raise MyException(10, f"L'élève d'identifiant {élève_id} n'existe pas")
35.         # résultat
36.         return élèves[0]
```

- les lignes 5-19 ne posent pas de difficultés ;
- lignes 29-36 : la méthode qui rend l'élève dont on passe le numéro. Si l'élève n'existe pas, une exception est levée ;
- ligne 31 : la fonction **[filter]** permet de filtrer une liste :
  - le 1er paramètre est le critère de filtrage ;
  - le second paramètre est la liste à filtrer, ici la liste des élèves ;
- ligne 31 : le critère de filtrage de la liste est implémenté à l'aide d'une fonction **[f(e :Elève)->bool]**. Celle-ci est appliquée à chacun des éléments de la liste à filtrer. Si l'élément satisfait le critère de filtrage, il est retenu dans la liste filtrée, sinon il en est exclu. On peut ici soit :

    - donner le nom de la fonction f et implémenter celle-ci ailleurs. L'appel à la fonction **[filter]** devient alors **[filter(f,self.get_élèves()]** ;
    - donner la définition de la fonction f. L'appel à la fonction **[filter]** devient alors **[filter(f(e :Elève){…},self.get_élèves()]** où **[e]** représente un élément de la liste filtrée, ç-à-d un élève. C'est ce qui a été fait ici. La définition de la fonction f serait ici **[f(e :Elève){return e.id==élève_id)]** : un élève n'est retenu que si non n° **[id]** est égal à celui cherché. Une telle fonction peut être remplacée par une fonction dite lambda : **[lambda e: e.id == élève_id]** :
        - e : représente le paramètre de la fonction f, ici un élève. On peut utiliser le nom que l'on veut ;
        - e.id==élève_id est le critère de filtrage : un élève **[e]** n'est retenu que si son n° **[id]** correspond à celui qui est cherché ;
- ligne 31 : la fonction **[filter]** rend la liste filtrée sous un type qui n'est pas le type **[list]** mais qui supporte d'être transformé en type **[list]**. C'est ce que nous faisons ici avec l'expression **[list(liste filtrée)]** ;
- lignes 33-34 : si la liste filtrée est vide c'est que l'élève cherché n'existe pas. On lève alors une exception ;
- ligne 36 : si on arrive ici, c'est qu'il n'y a pas eu d'exception. On sait alors qu'on a récupéré une liste à 1 élément (il n'y a pas deux élèves ayant le même n° **[id]**). On rend donc le 1er élément de la liste ;
- lignes 21-27 : la méthode **[get_notes_for_élève_by_id]** doit rendre les notes de l'élève dont on lui passe le n° **[id]** ;
- lignes 22-23 : on commence par rechercher l'élève de n° **[élève_id]** à l'aide de la méthode **[get_élève_by_id]** que l'on vient de commenter. Il peut se produire une exception si l'élève cherché n'existe pas. Comme il n'y a pas de try / catch autour de l'instruction de la ligne 23, l'exception remontera au code appelant. C'est ce qui est désiré ;
- lignes 24-25 : une fois l'élève récupéré, on récupère toutes ses notes. On le fait de nouveau avec un filtre :
    - le filtre est **[filter(critère, self_getnotes()]**. La liste à filtrer est donc la liste de toutes les notes de tous les élèves du collège ;
    - le critère de filtrage est exprimé à l'aide d'une fonction **[lambda]** : lambda n: n.élève.id == élève_id. Le paramètre n est un élément de la liste à filtrer, donc une note. Le type **[Note]** a une propriété **[élève]** qui représente l'élève propriétaire de la note. Il faut alors que **[n.élève.id]** qui représente le n° de cet élève soit égal au numéro de l'élève cherché ;

Puis nous exécutons le script **[tests-dao.py]**.

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # instanciation couche [dao]
7.  from Dao import Dao
8.
9.  daoImpl = Dao()
10.
11. # liste des classes
12. for classe in daoImpl.get_classes():
13.     print(classe)
14.
15. # liste des matières
16. for matière in daoImpl.get_matières():
17.     print(matière)
18.
19. # liste des classes
20. for élève in daoImpl.get_élèves():
21.     print(élève)
22.
23. # liste des notes
24. for note in daoImpl.get_notes():
25.     print(note)
26.
27. # un élève particulier
28. print(daoImpl.get_élève_by_id(11))
29.
30. # la liste de ses notes
31. dict1 = daoImpl.get_notes_for_élève_by_id(11)
32. print(f"élève n° 11 = {dict1['élève']}")
33. for note in dict1["notes"]:
34.     print(f"note de l'élève n° 11 = {note}")
```

Nous obtenons alors les résultats suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/troiscouches/v01/tests/tests_dao.py
2.  {"id": 1, "nom": "classe1"}
3.  {"id": 2, "nom": "classe2"}
4.  {"id": 1, "nom": "matière1", "coefficient": 1}
5.  {"id": 2, "nom": "matière2", "coefficient": 2}
```

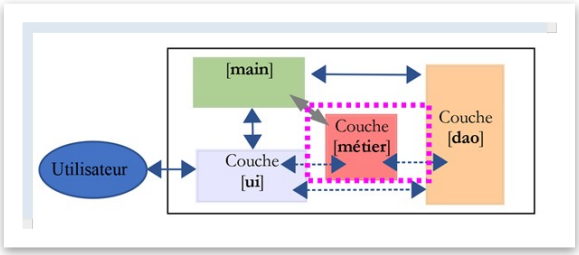
```
6.  {"id": 11, "nom": "nom1", "prénom": "prénom1", "classe": {"id": 1, "nom": "classe1"}}
7.  {"id": 21, "nom": "nom2", "prénom": "prénom2", "classe": {"id": 1, "nom": "classe1"}}
8.  {"id": 32, "nom": "nom3", "prénom": "prénom3", "classe": {"id": 2, "nom": "classe2"}}
9.  {"id": 42, "nom": "nom4", "prénom": "prénom4", "classe": {"id": 2, "nom": "classe2"}}
10. {"id": 1, "valeur": 10, "élève": {"id": 11, "nom": "nom1", "prénom": "prénom1", "classe": {"id": 1, "nom":
    "classe1"}}, "matière": {"id": 1, "nom": "matière1", "coefficient": 1}}
11. {"id": 2, "valeur": 12, "élève": {"id": 21, "nom": "nom2", "prénom": "prénom2", "classe": {"id": 1, "nom":
    "classe1"}}, "matière": {"id": 1, "nom": "matière1", "coefficient": 1}}
12. {"id": 3, "valeur": 14, "élève": {"id": 32, "nom": "nom3", "prénom": "prénom3", "classe": {"id": 2, "nom":
    "classe2"}}, "matière": {"id": 1, "nom": "matière1", "coefficient": 1}}
13. {"id": 4, "valeur": 16, "élève": {"id": 42, "nom": "nom4", "prénom": "prénom4", "classe": {"id": 2, "nom":
    "classe2"}}, "matière": {"id": 1, "nom": "matière1", "coefficient": 1}}
14. {"id": 5, "valeur": 6, "élève": {"id": 11, "nom": "nom1", "prénom": "prénom1", "classe": {"id": 1, "nom":
    "classe1"}}, "matière": {"id": 2, "nom": "matière2", "coefficient": 2}}
15. {"id": 6, "valeur": 8, "élève": {"id": 21, "nom": "nom2", "prénom": "prénom2", "classe": {"id": 1, "nom":
    "classe1"}}, "matière": {"id": 2, "nom": "matière2", "coefficient": 2}}
16. {"id": 7, "valeur": 10, "élève": {"id": 32, "nom": "nom3", "prénom": "prénom3", "classe": {"id": 2, "nom":
    "classe2"}}, "matière": {"id": 2, "nom": "matière2", "coefficient": 2}}
17. {"id": 8, "valeur": 12, "élève": {"id": 42, "nom": "nom4", "prénom": "prénom4", "classe": {"id": 2, "nom":
    "classe2"}}, "matière": {"id": 2, "nom": "matière2", "coefficient": 2}}
18. {"id": 11, "nom": "nom1", "prénom": "prénom1", "classe": {"id": 1, "nom": "classe1"}}
19. élève n° 11 = {"id": 11, "nom": "nom1", "prénom": "prénom1", "classe": {"id": 1, "nom": "classe1"}}
20. note de l'élève n° 11 = {"id": 1, "valeur": 10, "élève": {"id": 11, "nom": "nom1", "prénom": "prénom1",
    "classe": {"id": 1, "nom": "classe1"}}, "matière": {"id": 1, "nom": "matière1", "coefficient": 1}}
21. note de l'élève n° 11 = {"id": 5, "valeur": 6, "élève": {"id": 11, "nom": "nom1", "prénom": "prénom1",
    "classe": {"id": 1, "nom": "classe1"}}, "matière": {"id": 2, "nom": "matière2", "coefficient": 2}}
22.
23. Process finished with exit code 0
```

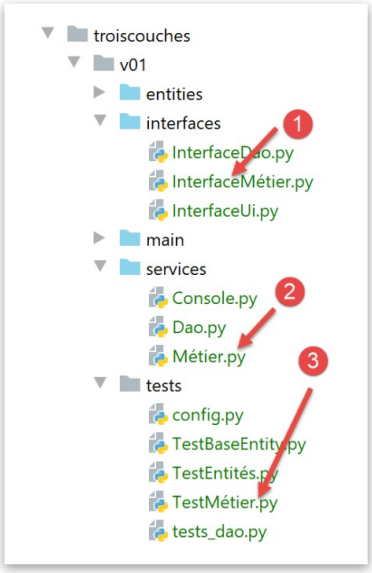
On pourra remarquer que lorsqu'on affiche une note (pour les autres objets, c'est similaire), on a également :

- l'élève propriétaire de la note ;
- la matière référencée par la note ;

C'est la fonction **[BaseEntity.asdict]** qui produit ce résultat (cf. paragraphe lien).

14.2.5 La couche [métier]





- **[InterfaceMétier]** est l'interface de la couche **[métier]** ;
- **[Métier]** est la classe d'implémentation de la couche **[métier]** ;
- **[Testmétier]** est une classe **[UnitTest]** de test de la classe **[Métier]** ;

14.2.5.1 Interface [InterfaceMétier]

La couche **[métier]** implémentera l'interface **[InterfaceMétier]** suivante (InterfaceMétier.py) :

```
1.  # imports
```

```
2.  from abc import ABC, abstractmethod
3.
4.  from StatsForElève import StatsForElève
5.
6.
7.  # interface Métier
8.  class InterfaceMétier(ABC):
9.      # calcul de statistiques pour un élève
10.     @abstractmethod
11.     def get_stats_for_élève(self, idElève: int) -> StatsForElève:
12.         pass
```

- **[get_stats_for_élève]** rend les notes de l'élève n° *idElève* ainsi que des informations sur celles-ci : *moyenne pondérée*, *note la plus basse*, *note la plus haute*. Ces informations sont encapsulées dans un objet de type **[StatsForElève]** ;

14.2.5.2 L'entité [StatsForElève]

Le type **[StatsForElève]** (StatsForElève.py) qui encapsule les statistiques (notes, min, max, moyenne pondérée) d'un élève est le suivant :

```
1.  # imports
2.  from BaseEntity import BaseEntity
3.
4.
5.  # statistiques d'un élève particulier
6.
7.
8.  class StatsForElève(BaseEntity):
9.      # attributs exclus de l'état de la classe
10.     excluded_keys = []
11.
12.     # propriétés de la classe
13.     @staticmethod
14.     def get_allowed_keys() -> list:
15.         # id : identifiant de la note
16.         # élève : l'élève concerné
17.         # notes : ses notes
18.         # moyennePondérée : sa moyenne pondérée par les coefficients des matières
19.         # min : sa note minimale
20.         # max : sa note maximale
21.
22.         return BaseEntity.get_allowed_keys() + ["élève", "notes", "moyenne_pondérée", "min", "max"]
23.
24.     # toString
25.     def __str__(self) -> str:
26.         # cas de l'élève sans notes
27.         if len(self.notes) == 0:
28.             return f"Elève={self.élève}, notes=[]"
29.         # cas de l'élève avec notes
30.         str = ""
31.         for note in self.notes:
32.             str += f"{note.valeur} "
33.         return f"Elève={self.élève}, notes=[{str.strip()}], max={self.max}, min={self.min}, " \
34.                f"moyenne pondérée={self.moyenne_pondérée:4.2f}"
```

Notes :

- ligne 8 : la classe **[StatsForElève]** dérive de la classe **[BaseEntity]** ;
- lignes 13-22 : les propriétés de la classe ;
  - un identifiant **[id]** provenant de **[BaseEntity]** ;
  - l'élève **[élève]** dont on encapsile les statistiques ;
  - ses notes **[notes]** ;
  - sa moyenne pondérée **[moyenne_pondérée]** ;
  - sa note minimale **[min]** ;
  - sa note maximale **[max]** ;
- on ne définit pas de getters / setters pour ces attributs. On part du principe que c'est la couche **[métier]** qui crée des objets de ce type et que celle-ci ne crée pas d'objets invalides ;
- lignes 23-33 : la fonction **[__str__]** rend une chaîne de caractères reprenant les propriétés de l'objet ;

14.2.5.3 L'implémentation [Métier]

L'implémentation **[Métier]** (Metier.py) de l'interface **[InterfaceMétier]** sera la suivante :

```
1.  # imports
2.  from InterfaceDao import InterfaceDao
3.  from InterfaceMétier import InterfaceMétier
4.  from StatsForElève import StatsForElève
5.
6.
7.  class Métier(InterfaceMétier):
8.
9.      # constructeur
10.     def __init__(self, dao: InterfaceDao):
11.         # on mémorise le paramètre
12.         self.__dao = dao
13.
14.     # -----------
15.     # interface
16.     # -----------
17.
18.     # les indicateurs sur les notes d'un élève particulier
19.     def get_stats_for_élève(self, id_élève: int) -> StatsForElève:
20.         # Stats pour l'élève de n° idEleve
21.         # id_élève : n° de l'élève
22.
23.         # on récupère ses notes avec la couche [dao]
24.         notes_élève = self.__dao.get_notes_for_élève_by_id(id_élève)
25.         élève = notes_élève["élève"]
26.         notes = notes_élève["notes"]
27.
28.         # on s'arrête s'il n'y a pas de notes
29.         if len(notes) == 0:
30.             # on rend le résultat
31.             return StatsForElève().fromdict({"élève": élève, "notes": []})
32.
33.         # exploitation des notes de l'élève
34.         somme_pondérée = 0
35.         somme_coeff = 0
36.         max = -1
37.         min = 21
38.         for note in notes:
39.             # valeur de la note
40.             valeur = note.valeur
41.             # coefficient de la matière
42.             coeff = note.matière.coefficient
43.             # somme des coefficients
44.             somme_coeff += coeff
45.             # somme pondérée
46.             somme_pondérée += valeur * coeff
47.             # recherche du min
48.             if valeur < min:
49.                 min = valeur
50.             # recherche du max
51.             if valeur > max:
52.                 max = valeur
53.         # calcul des indicateurs manquants
54.         moyenne_pondérée = float(somme_pondérée) / somme_coeff
55.
56.         # on rend le résultat sous la forme d'un type [StatsForElève]
57.         return StatsForElève(). \
58.             fromdict({"élève": élève, "notes": notes,
59.                       "moyenne_pondérée": moyenne_pondérée,
60.                       "min": min, "max": max})
```

Notes

- ligne 7 : la classe **[Métier]** dérive de la classe **[InterfaceMétier]**. On a pris l'habitude de dire qu'elle implémente l'interface **[InterfaceMétier]** ;
- lignes 9-12 : le constructeur reçoit pour seul paramètre une référence sur la couche **[dao]**. Ligne 10, on notera qu'on a donné le type **[InterfaceDao]** au paramètre **[dao]**. On n'attend pas une implémentation précise mais simplement une implémentation restectant l'interface **[InterfaceDao]**. Ici, ça n'a pas d'importance puisque Python ne va pas tenir compte de ce type mais c'est une bonne habitude de travailler avec des interfaces plutôt qu'avec des implémentations précises. Le code est alors plus facilement modifiable ;
- lignes 19-60 : implémentation de la méthode **[get_stats_for_élève]** ;
- ligne 19 : la méthode reçoit un unique paramètre, le n° **[idElève]** de l'élève dont on veut les statistiques ;
- ligne 24 : on demande à la couche **[dao]**, les notes de l'élève. Cette demande débouche sur une exception si l'élève n'existe pas. Celle-ci n'est pas gérée (absence de try / catch) et remonte donc au code appelant ;

- ligne 25 : on arrive ici s'il n'y a pas eu exception. **[notes_élève]** est alors un dictionnaire à deux clés **[élève, note]** :
  - ligne 25 : on récupère les informations sur l'élève (son nom, sa classe, …) ;
  - ligne 26 : on récupère ses notes ;
- lignes 28-31 : on regarde si l'élève a des notes. S'il n'en a pas, il n'y a pas de statistiques à calculer ;
- ligne 31 : on rend un objet **[StatsForElève]** construit à partir d'un dictionnaire avec la méthode **[BaseEntity.fromdict]** ;
- lignes 33-54 : on exploite les notes de l'élève pour calculer les statistiques demandées. Les commentaires du code devraient suffire à sa compréhension ;
- lignes 56-60 : on rend un objet **[StatsForElève]** construit à partir d'un dictionnaire avec la méthode **[BaseEntity.fromdict]** ;

14.2.5.4 Test de la couche [métier]

Un script **[UnitTest]** de la couche **[métier]** pourrait être le suivant (TestMétier.py) :

```
1.  # imports
2.  import unittest
3.
4.
5.  class Testmétier(unittest.TestCase):
6.      def setUp(self):
7.          # on configure l'application
8.          import config
9.          config.configure()
10.
11.     def test_statsForEleve11(self):
12.         # imports
13.         from Dao import Dao
14.         from Métier import Métier
15.         # on teste les indicateurs de l'élève 11
16.         dao = Dao()
17.         stats_for_élève = Métier(dao).get_stats_for_élève(11)
18.         # affichage
19.         print(f"\nstats={stats_for_élève}")
20.         # vérifications
21.         self.assertEqual(stats_for_élève.min, 6)
22.         self.assertEqual(stats_for_élève.max, 10)
23.         self.assertAlmostEqual(stats_for_élève.moyenne_pondérée, 7.333, delta=1e-3)
24.
25.
26. if __name__ == '__main__':
27.     unittest.main()
```

Notes

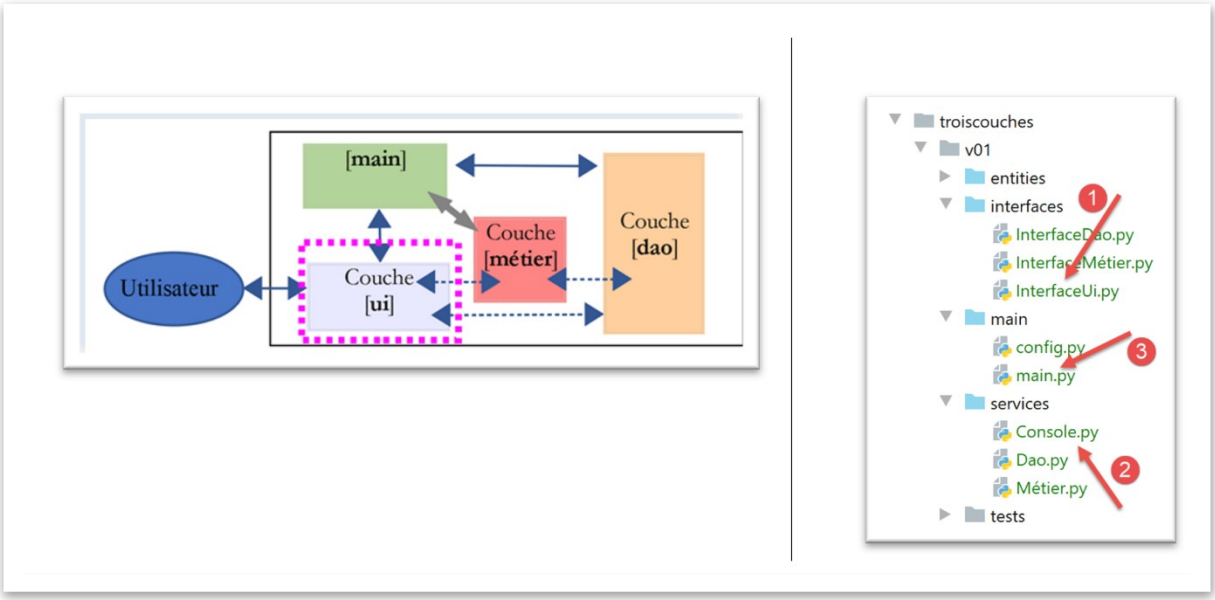
- lignes 6-9 : la fonction **[setUp]** est ici utilisée pour configurer le Python Path du test ;
- ligne 16 : on instancie la couche **[dao]** ;
- ligne 17 : on instancie la couche **[métier]** et on utilise sa méthode **[get_stats_for_élève]** pour calculer les statistiques de l'élève n° 11 ;
- ligne 19 : on affiche le résultat **[StatsForElève]** obtenu. Comme **[StatsForElève]** dérive de **[BaseEntity]**, c'est la chaîne jSON de **[StatsForElève]** qui est ici affichée ;
- ligne 21 : on vérifie la note minimale de l'élève ;
- ligne 22 : on vérifie sa note maximale ;
- ligne 23 : on teste que la moyenne pondérée vaut 7,333 à 10^{-3} près. En général, il n'est pas possible de comparer les nombres réels de façon exacte car de façon interne, ils n'ont le plus souvent qu'une représentation approchée ;

Les résultats du test sont les suivants :

```
1.  Testing started at 18:17 ...
2.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe "C:\Program
    Files\JetBrains\PyCharm Community Edition 2020.1.2\plugins\python-
    ce\helpers\pycharm\_jb_unittest_runner.py" --path C:/Data/st-2020/dev/python/cours-2020/python3-
    flask-2020/troiscouches/v01/tests/TestMétier.py
3.  Launching unittests with arguments python -m unittest C:/Data/st-2020/dev/python/cours-2020/python3-
    flask-2020/troiscouches/v01/tests/TestMétier.py in C:\Data\st-2020\dev\python\cours-2020\python3-
    flask-2020\troiscouches\v01\tests
4.
5.
6.
7.  Ran 1 test in 0.015s
8.
9.  OK
10.
```

```
11. stats=Elève={"id": 11, "nom": "nom1", "prénom": "prénom1", "classe": {"id": 1, "nom": "classe1"}},
    notes=[10 6], max=10, min=6, moyenne pondérée=7.33
12.
13. Process finished with exit code 0
```

14.2.6 La couche [ui]



- en **[1]**, l'interface de la couche **[ui]** ;
- en **[2]**, l'implémentation de cette interface ;
- en **[3]**, le script principal de l'application ;

14.2.6.1 Interface [InterfaceUi]

L'interface de la couche **[UI]** sera la suivante :

```
1.  # imports
2.  from abc import ABC, abstractmethod
3.
4.
5.  # interface UI
6.  class InterfaceUi(ABC):
7.      # exécution de la couche UI
8.      @abstractmethod
9.      def run(self: object):
10.         pass
```

Notes

- lignes 9-10 : la couche **[UI]** n'aura qu'une méthode, **[run]** ;

14.2.6.2 L'implémentation [Console]

La couche **[console]** est implémentée par le script **[Console.py]** suivant :

```
1.  # imports des couches
2.
3.  from InterfaceDao import InterfaceDao
4.  from InterfaceMétier import InterfaceMétier
5.  from InterfaceUi import InterfaceUi
6.
7.  # autres dépendances
8.  from MyException import MyException
9.
10.
11. class Console(InterfaceUi):
12.     # constructeur
13.     def __init__(self: object, métier: InterfaceMétier):
14.         # métier : la couche [métier]
15.
```

```
16.         # on mémorise les attributs
17.         self.métier = métier
18.
19.
20.         # -----------
21.         # interface
22.         # -----------
23.
24.     def run(self):
25.         # dialogue utilisateur
26.         fini = False
27.         while not fini:
28.             # question / réponse
29.             réponse = input("Numéro de l'élève (>=1 et * pour arrêter) : ").strip()
30.             # fini ?
31.             if réponse == "*":
32.                 break
33.             # a-t-on une saisie correcte ?
34.             ok = False
35.             try:
36.                 id_élève = int(réponse, 10)
37.                 ok = id_élève >= 1
38.             except ValueError as erreur:
39.                 pass
40.             # donnée correcte ?
41.             if not ok:
42.                 print("Saisie incorrecte. Recommencez...")
43.                 continue
44.             # calcul des statistiques pour l'élève choisi
45.             try:
46.                 print(self.métier.get_stats_for_élève(id_élève))
47.             except MyException as erreur:
48.                 print(f"L'erreur suivante s'est produite : {erreur}")
```

- lignes 3-5 : import de toutes les interfaces ;
- ligne 11 : la classe **[Console]** implémente l'interface **[InterfaceUi]** ;
- lignes 12-17 : le constructeur de la classe **[Console]** reçoit en paramètre une référence sur la couche **[métier]**. On notera qu'on a donné le type **[InterfaceMétier]** à ce paramètre pour rappeler qu'on travaille avec des interfaces plutôt qu'avec des implémentations précises ;
- ligne 24 : implémentation de la méthode **[run]** de l'interface ;
- ligne 27 : une boucle qui s'arrête lorsque la condition de la ligne 31 est vérifiée ;
- ligne 29 : saisie d'une donnée tapée au clavier. La fonction **[input]** reçoit un paramètre facultatif : le message à écrire sur l'écran pour demander la saisie. Celle-ci est toujours récupérée comme une chaîne de caractères. La fonction **[strip]** débarrasse celle-ci de ses « blancs » qui la suivent ou la précèdent ;
- lignes 34-39 : on vérifie que la saisie, un n° d'élève, est valide. Il faut que ce soit un entier >=1. On rappelle que la saisie a été faite en tant que chaîne de caractères ;
- ligne 36 : on essaie de transformer la saisie en nombre entier en base 10. La fonction **[int]** lance une exception si ce n'est pas possible ;
- ligne 37 : on arrive là seulement s'il n'y a pas eu d'exception. On vérifie que le nombre entier récupéré est bien >=1 ;
- lignes 38-39 : on gère l'exception. S'il y a eu exception, la variable **[ok]** de la ligne 34 est restée à **[False]** ;
- lignes 41-43 : si la saisie a été incorrecte, on affiche un message d'erreur et on reboucle (ligne 43) ;
- lignes 45-48 : on calcule les statistiques de l'élève dont on a saisi le n° ;
- ligne 46 : on utilise la méthode **[get_stats_for_élève]** de la couche **[métier]**. Celle-ci lance une exception si l'élève n'existe pas. Celle-ci est gérée aux lignes 47-48. On sait que les couches **[dao]** et **[métier]** lancent l'exception **[MyException]** ;

14.3 Le script principal [main]

Le script principal **[main]** est le suivant (main.py) :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  from Console import Console
8.  from Dao import Dao
9.  from Métier import Métier
10.
11. # ----------- couche [console]
12. try:
13.     # instanciation couche [dao]
```

```
14.     dao = Dao()
15.     # instanciation couche [métier]
16.     métier = Métier(dao)
17.     # instanciation couche [ui]
18.     console = Console(métier)
19.     # exécution couche [console]
20.     console.run()
21. except BaseException as ex:
22.     # on affiche l'erreur
23.     print(f"L'erreur suivante s'est produite : {ex}")
24. finally:
25.     pass
```

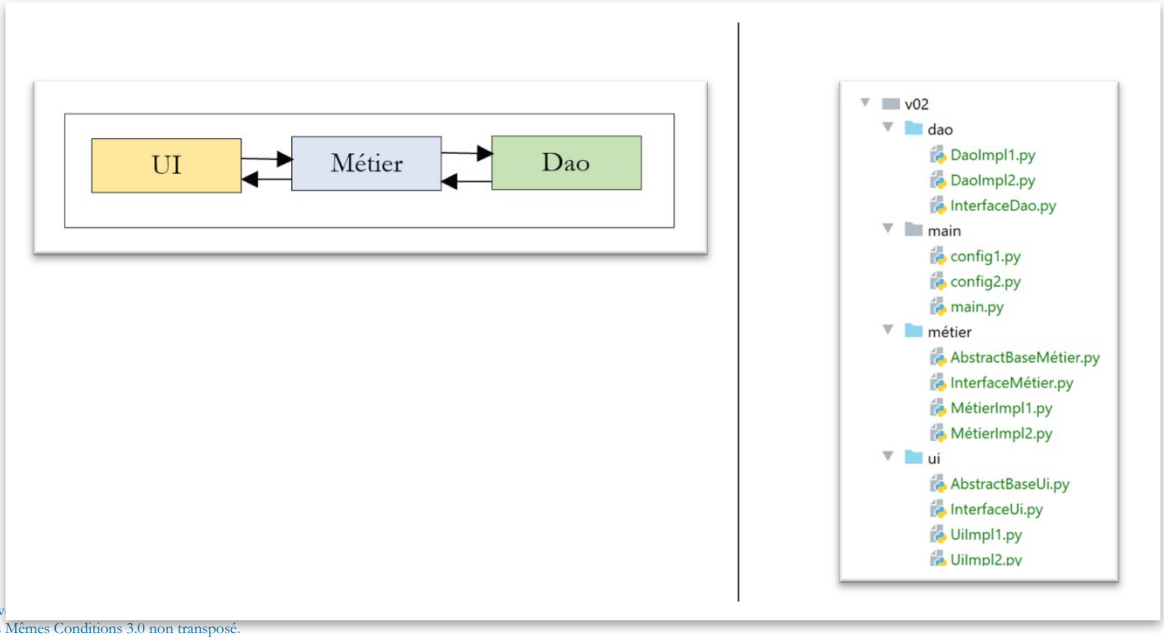
- lignes 1-4 : on configure le Python Path de l'application ;
- lignes 6-9 : on importe les classes et interfaces dont on a besoin ;
- ligne 14 : instanciation de la couche **[dao]** ;
- ligne 16 : instanciation de la couche **[métier]** ;
- ligne 18 : instanciation de la couche **[ui]** ;
- ligne 20 : on lance le dialogue avec l'utilisateur ;
- lignes 13-20 : normalement aucune exception ne sort de ces lignes. Celles qui remontent des couches **[dao]** et **[métier]** sont arrêtées par la couche **[Console]**. La gestion des exceptions est un art difficile lorsqu'on ne connaît pas parfaitement les couches utilisées (ici ce n'est pas le cas). Dans le cas de doutes, on peut ajouter du code pour arrêter tout type d'exception pouvant être lancé par le code exécuté. C'est ce qui est fait ici, lignes 21-23. On arrête toute exception dérivant de **[BaseException]**, ç-à-d toutes les exceptions ;
- lignes 24-25 : la clause **[finally]** ne fait rien ici. Elle n'est là que pour pouvoir mettre en commentaires les lignes 21-23. En effet, en mode débogage on n'a pas intérêt à arrêter les exceptions. Dans ce cas, c'est l'interpréteur Python qui les arrête et il donne alors le n° de la ligne où l'exception s'est produite. Une information indispensable. Lorsque les lignes 21-23 sont mises en commentaires, la présence des lignes 24-25 permet d'avoir un try/catch syntaxiquement correct. En leur absence, Python déclare une erreur ;

Voici un exemple d'exécution :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/troiscouches/v01/main/main.py
2.  Numéro de l'élève (>=1 et * pour arrêter) : 11
3.  Elève={"id": 11, "nom": "nom1", "prénom": "prénom1", "classe": {"id": 1, "nom": "classe1"}}, notes=[10 6],
    max=10, min=6, moyenne pondérée=7.33
4.  Numéro de l'élève (>=1 et * pour arrêter) : 1
5.  L'erreur suivante s'est produite : MyException[10, L'élève d'identifiant 1 n'existe pas]
6.  Numéro de l'élève (>=1 et * pour arrêter) : *
7.
8.  Process finished with exit code 0
```

14.4 Exemple 2

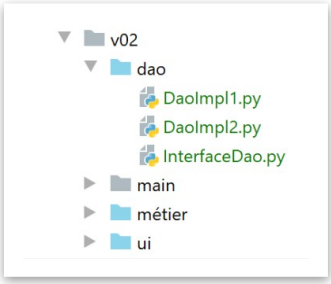
Ce nouvel exemple d'architectures en couches vise à montrer l'intérêt de la programmation par interfaces. Celle-ci facilite la maintenance et le test des applications. Nous utiliserons de nouveau une architecture à trois couches :



Partage dans les Mêmes Conditions 3.0 non transposé.

Chaque couche sera implémentée de deux façons différentes. Nous voulons montrer que l'on peut changer facilement l'implémentation d'une couche avec un impact minimal sur les autres.

14.4.1 La couche `[dao]`



L'interface **`[InterfaceDao]`** est la suivante:

```
1.  # imports
2.  from abc import ABC, abstractmethod
3.
4.
5.  # interface Dao
6.  class InterfaceDao(ABC):
7.      # une seule méthode
8.      @abstractmethod
9.      def do_something_in_dao_layer(self, x: int, y: int) -> int:
10.         pass
```

- lignes 8-10: la méthode **`[do_something_in_dao_layer]`** est l'unique méthode de l'interface ;

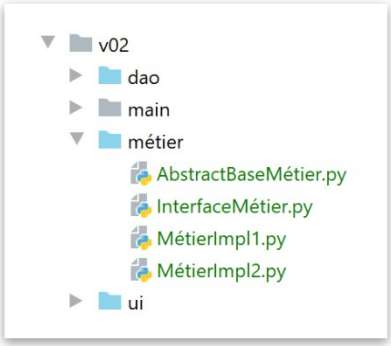
La classe **`[DaoImpl1]`** implémente l'interface **`[InterfaceDao]`** de la façon suivante :

```
1.  from InterfaceDao import InterfaceDao
2.
3.
4.  class DaoImpl1(InterfaceDao):
5.      # implémentation InterfaceDao
6.      def do_something_in_dao_layer(self: InterfaceDao, x: int, y: int) -> int:
7.          return x + y
```

La classe **`[DaoImpl2]`** implémente l'interface **`[InterfaceDao]`** de la façon suivante :

```
1.  from InterfaceDao import InterfaceDao
2.
3.
4.  class DaoImpl2(InterfaceDao):
5.      # implémentation InterfaceDao
6.      def do_something_in_dao_layer(self: InterfaceDao, x: int, y: int) -> int:
7.          return x - y
```

14.4.2 La couche `[métier]`



L'interface **[InterfaceMétier]** est la suivante:

```
1.  # imports
2.  from abc import ABC, abstractmethod
3.
4.
5.  # interface métier
6.  class InterfaceMétier(ABC):
7.      # une seule méthode
8.      @abstractmethod
9.      def do_something_in_métier_layer(self, x: int, y: int) -> int:
10.         pass
```

- lignes 8-10: la méthode **[do_something_in_métier_layer]** est l'unique méthode de l'interface ;

La classe **[AbstractBaseMétier]** implémente l'interface **[InterfaceMétier]** de la façon suivante :

```
1.  # imports
2.  from abc import ABC, abstractmethod
3.
4.  from InterfaceDao import InterfaceDao
5.  from InterfaceMétier import InterfaceMétier
6.
7.
8.  class AbstractBaseMétier(InterfaceMétier, ABC):
9.      # propriétés
10.     # __dao est une référence sur la couche [dao]
11.     @property
12.     def dao(self) -> InterfaceDao:
13.         return self.__dao
14.
15.     @dao.setter
16.     def dao(self, dao: InterfaceDao):
17.         self.__dao = dao
18.
19.     # implémentation de l'interface [InterfaceMétier]
20.     @abstractmethod
21.     def do_something_in_métier_layer(self, x: int, y: int) -> int:
22.         pass
```

- ligne 8 : la classe **[AbstractBaseMétier]** dérive deux classes :
  - **[InterfaceMétier]** : la classe **[AbstractBaseMétier]** implémente cette interface aux lignes 19-22. En fait on voit qu'elle n'a pas implémenté la méthode **[do_something_in_métier_layer]** qu'elle a déclarée abstraite (ligne 20). Ce sera aux classes dérivées d'implémenter la méthode;
  - **[ABC]** pour avoir accès aux annotations **[@abstractmethod]** ;
  - l'ordre a un sens : si on l'inverse ici, Python déclare une erreur à l'exécution ;

C'est la première fois qu'on utilise l'héritage multiple (hériter de plusieurs classes). La classe **[AbstractBaseMétier]** hérite, à la fois, des propriétés des classes **[InterfaceMétier]** et **[ABC]**.

- lignes 9-17 : on définit la propriété **[dao]** qui sera une référence sur la couche **[dao]** ;

Une interface est destinée à être implémentée. Lorsque des implémentations différentes partagent des propriétés il est intéressant de mettre celles-ci dans une classe parent afin d'éviter de les dupliquer. C'est le cas ici de la propriété **[dao]**. La classe parent est en général toujours abstraite parce qu'elle ne sait pas implémenter toutes les méthodes de l'interface.

La classe **[MétierImpl1]** implémente l'interface **[InterfaceMétier]** de la façon suivante :

```
1.  from AbstractBaseMétier import AbstractBaseMétier
2.
3.
4.  class MétierImpl1(AbstractBaseMétier):
5.      # implémentation de l'interface [InterfaceMétier]
6.      def do_something_in_métier_layer(self:AbstractBaseMétier, x: int, y: int) -> int:
7.          x += 1
8.          y += 1
9.          return self.dao.do_something_in_dao_layer(x, y)
```

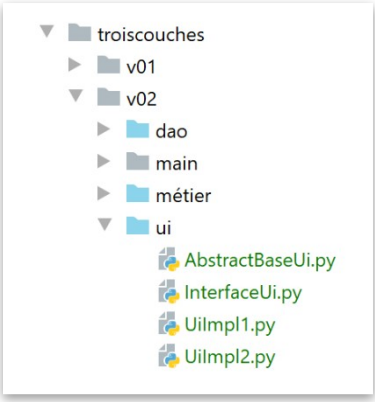
- ligne 4 : la classe **[MétierImpl1]** dérive de la classe **[AbstractbaseMétier]**. Elle hérite donc de la propriété **[dao]** de cette classe ;

- lignes 6-9 : implémentation de l'interface **[InterfaceMétier]** que n'a pas implémentée la classe parent **[AbstractbaseMétier]** ;
- ligne 9 : on utilise la couche **[dao]** ;

La classe **[MétierImpl2]** implémente l'interface **[InterfaceMétier]** de façon analogue :

```
1.  from AbstractBaseMétier import AbstractBaseMétier
2.
3.
4.  class MétierImpl2(AbstractBaseMétier):
5.      # implémentation de l'interface [InterfaceMétier]
6.      def do_something_in_métier_layer(self:AbstractBaseMétier, x: int, y: int) -> int:
7.          x -= 1
8.          y -= 1
9.          return self.dao.do_something_in_dao_layer(x, y)
```

14.4.3 La couche [ui]



L'interface **[InterfaceUi]** est la suivante :

```
1.  # imports
2.  from abc import ABC, abstractmethod
3.
4.
5.  # interface Ui
6.  class InterfaceUi(ABC):
7.      # une seule méthode
8.      @abstractmethod
9.      def do_something_in_ui_layer(self, x: int, y: int) -> int:
10.         pass
```

- lignes 8-10 : l'unique méthode de l'interface ;

La classe **[AbstractBaseUi]** implémente l'interface **[InterfaceUi]** de la façon suivante :

```
1.  # imports
2.  from abc import ABC, abstractmethod
3.
4.  from InterfaceMétier import InterfaceMétier
5.  from InterfaceUi import InterfaceUi
6.
7.
8.  class AbstractBaseUi(InterfaceUi, ABC):
9.      # propriétés
10.     # métier est une référence sur la couche [métier]
11.     @property
12.     def métier(self) -> InterfaceMétier:
13.         return self.__métier
14.
15.     @métier.setter
16.     def métier(self, métier: InterfaceMétier):
17.         self.__métier = métier
18.
19.     # implémentation de l'interface [InterfaceUI]
20.     @abstractmethod
```

```
21.     def do_something_in_ui_layer(self: InterfaceUi, x: int, y: int) -> int:
22.         pass
```

- la classe **[AbstractBaseUi]** est une classe abstraite (ligne 20). Elle devra être dérivée pour implémenter l'interface **[InterfaceUi]** ;
- lignes 9-17 : la classe **[AbstractBaseUi]** possède une référence sur la couche **[métier]** ;

La classe d'implémentation **[UiImpl1]** est la suivante :

```
1.  from AbstractBaseUi import AbstractBaseUi
2.
3.
4.  class UiImpl1(AbstractBaseUi):
5.      # implémentation de l'interface [InterfaceUi]
6.      def do_something_in_ui_layer(self: AbstractBaseUi, x: int, y: int) -> int:
7.          x += 1
8.          y += 1
9.          return self.métier.do_something_in_métier_layer(x, y)
```

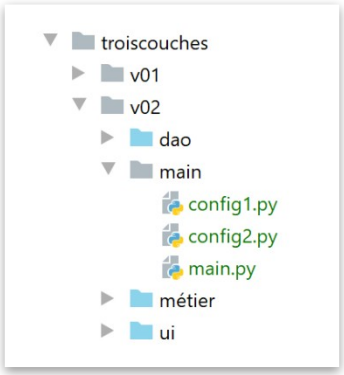
- ligne 4 : la classe **[UiImpl1]** dérive de la classe **[AbstractBaseUi]** et hérite donc de sa propriété **[métier]**. Celle-ci est utilisée ligne 9 ;

La classe d'implémentation **[UiImpl2]** est analogue :

```
1.  from AbstractBaseUi import AbstractBaseUi
2.
3.
4.  class UiImpl2(AbstractBaseUi):
5.      # implémentation de l'interface [InterfaceUi]
6.      def do_something_in_ui_layer(self: AbstractBaseUi, x: int, y: int) -> int:
7.          x -= 1
8.          y -= 1
9.          return self.métier.do_something_in_métier_layer(x, y)
```

- ligne 4 : la classe **[UiImpl2]** dérive de la classe **[AbstractBaseUi]** et hérite donc de sa propriété **[métier]**. Celle-ci est utilisée ligne 9 ;

14.4.4 Les fichiers de configuration



- les fichiers **[config1, config2]** configurent l'application de deux façons différentes ;
- le fichier **[main]** est le script principal de l'application ;

Le fichier **[config1]** est le suivant :

```
1.  def configure():
2.      # étape 1 ------
3.      # chemin absolu du dossier de ce script
4.      import os
5.      script_dir = os.path.dirname(os.path.abspath(__file__))
6.      # dépendances
7.      absolute_dependencies = [
8.          # dossiers locaux du Python Path
9.          f"{script_dir}/../dao",
10.         f"{script_dir}/../ui",
```

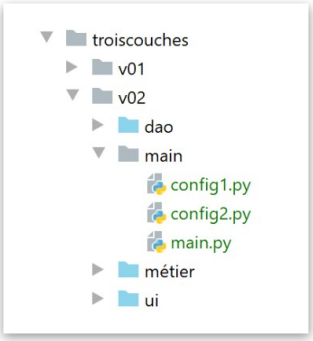
```
11.         f"{script_dir}/../métier",
12.     ]
13.
14.     # on configure le syspath
15.     from myutils import set_syspath
16.     set_syspath(absolute_dependencies)
17.
18.     # étape 2 ------
19.     # configuration des couches de l'application
20.     from DaoImpl1 import DaoImpl1
21.     from MétierImpl1 import MétierImpl1
22.     from UiImpl1 import UiImpl1
23.     # instanciation des couches
24.     # dao
25.     dao = DaoImpl1()
26.     # métier
27.     métier = MétierImpl1()
28.     métier.dao = dao
29.     # ui
30.     ui = UiImpl1()
31.     ui.métier = métier
32.
33.     # on met les instances de couche dans la config
34.     # seule la couche ui est ici nécessaire
35.     config = {"ui": ui}
36.
37.     # on rend la config
38.     return config
```

- lignes 2-16 : configuration du Python Path de l'application ;
- lignes 18-31 : instanciation des couches **[dao, métier, ui]**. Pour implémenter leurs interfaces, on choisit à chaque fois la 1ère implémentation construite ;
- lignes 33-35 : on met les références de couches dans la configuration. Ici, le script principal n'a besoin que de la couche **[ui]** ;

Le fichier **[config2]** est analogue et implémente chaque interface avec la 2ième implémentation disponible :

```
1.  def configure():
2.      # étape 1 ---
3.      # chemin absolu du dossier de ce script
4.      import os
5.      script_dir = os.path.dirname(os.path.abspath(__file__))
6.      # dépendances
7.      absolute_dependencies = [
8.          # dossiers locaux du Python Path
9.          f"{script_dir}/../dao",
10.         f"{script_dir}/../ui",
11.         f"{script_dir}/../métier",
12.     ]
13.
14.     # on configure le syspath
15.     from myutils import set_syspath
16.
17.     set_syspath(absolute_dependencies)
18.
19.     # étape 2 ------
20.     # configuration des couches de l'application
21.     from DaoImpl2 import DaoImpl2
22.     from MétierImpl2 import MétierImpl2
23.     from UiImpl2 import UiImpl2
24.     # instanciation des couches
25.     # dao
26.     dao = DaoImpl2()
27.     # métier
28.     métier = MétierImpl2()
29.     métier.dao = dao
30.     # ui
31.     ui = UiImpl2()
32.     ui.métier = métier
33.
34.     # on met les instances de couche dans la config
35.     # seule la couche ui est ici nécessaire
36.     config = {"ui": ui}
37.
38.     # on rend la config
39.     return config
```

14.4.5 Le script principal [main]



Le script principal est le suivant :

```
1.  # imports
2.  import importlib
3.  import sys
4.
5.  # main ---------
6.
7.  # il faut deux arguments
8.  nb_args = len(sys.argv)
9.  if nb_args != 2 or (sys.argv[1] != "config1" and sys.argv[1] != "config2"):
10.     print(f"Syntaxe : {sys.argv[0]} config1 ou config2")
11.     sys.exit()
12.
13. # configuration de l'application
14. module = importlib.import_module(sys.argv[1])
15. config = module.configure()
16.
17. # exécution de la couche [ui]
18. print(config["ui"].do_something_in_ui_layer(10, 20))
```

Ce script reçoit un paramètre :

- **[config1]** pour utiliser la configuration n° 1 ;
- **[config2]** pour utiliser la configuration n° 2 ;

Python enregistre les paramètres dans une liste **[sys.argv]** :

- sys.argv**[0]** est le nom du script, ici **[main]**. Ce paramètre est toujours présent ;
- sys.argv**[1]** est le 1er paramètre passé au script, sys.argv**[2]** le 2ième, …

- ligne 8 : on récupère le nombre de paramètres ;
- lignes 9-11 : on vérifie qu'il y a bien un paramètre et que sa valeur est soit **[config1]**, soit **[config2]**. Si ce n'est pas le cas, un message d'erreur est affiché (ligne 10) et on quitte le programme (ligne 11) ;

Une fois connue la configuration souhaitée, il nous faut exécuter cette configuration. Par exemple, si c'est la configuration 1 qui a été choisie, il nous faut exécuter le code :

```
1.  import config1
2.  config1.configure()
```

Le problème ici est que la configuration à utiliser est dans une variable, la variable **[sys.argv[1]**. Pour importer un module dont le nom est dans une variable, il nous faut utiliser le package**[importlib]** (ligne 2).

- ligne 14 : on importe le module dont le nom est dans **[sys.argv[1]** ;
- ligne 15 : ceci fait, on exécute la fonction **[configure]** de ce module. On récupère un dictionnaire **[config]** qui est la configuration de l'application ;
- ligne 18 : on sait qu'une référence de la couche **[ui]** est dans config**['ui']**. On l'utilise pour appeler la méthode **[do_something_in_ui_layer]**. On sait que cette méthode va appeler une méthode de la couche **[métier]** qui elle-même va appeler une méthode de la couche **[dao]** ;

Par exemple, la fonction **[do_something_in_ui_layer]** est la suivante :

```
1.  class UiImpl1(AbstractBaseUi):
2.      # implémentation de l'interface [InterfaceUi]
3.      def do_something_in_ui_layer(self: AbstractBaseUi, x: int, y: int) -> int:
4.          x += 1
5.          y += 1
6.          return self.métier.do_something_in_métier_layer(x, y)
```

- ligne 6 ci-dessus utilise la propriété **[métier]** de la classe **[UiImpl1]**, ligne 1. Or dans la configuration **[config1]** il a été écrit :

```
1.  # métier
2.      métier = MétierImpl1()
3.      métier.dao = dao
4.      # ui
5.      ui = UiImpl1()
6.      ui.métier = métier
```

- ligne 6 : la propriété **[métier]** de **[UIImpl1]** est une référence à la classe **[MétierImpl1]** (ligne 2). Ainsi c'est la méthode **[do_something_in_ui_layer]** de la classe **[MétierImpl1]** qui va être exécutée ;

Dans la classe **[MétierUiImpl1]**, il est écrit :

```
1.  class MétierImpl1(AbstractBaseMétier):
2.      # implémentation de l'interface [InterfaceMétier]
3.      def do_something_in_métier_layer(self: AbstractBaseMétier, x: int, y: int) -> int:
4.          x += 1
5.          y += 1
6.          return self.dao.do_something_in_dao_layer(x, y)
```

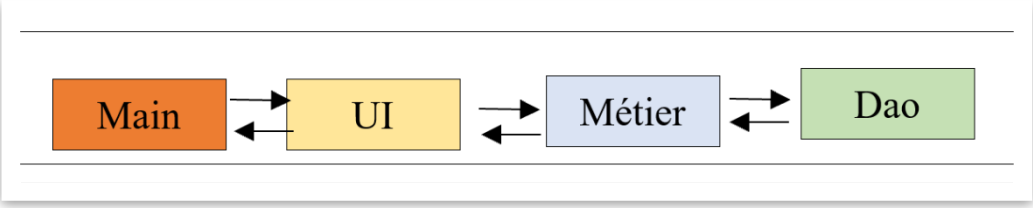
- ligne 6, la méthode appelée par la couche **[ui]** va à son tour appelée une méthode de la propriété **[dao]** de la classe **[MétierImpl1]** ;

Or dans la configuration **[config1]**, il a été écrit :

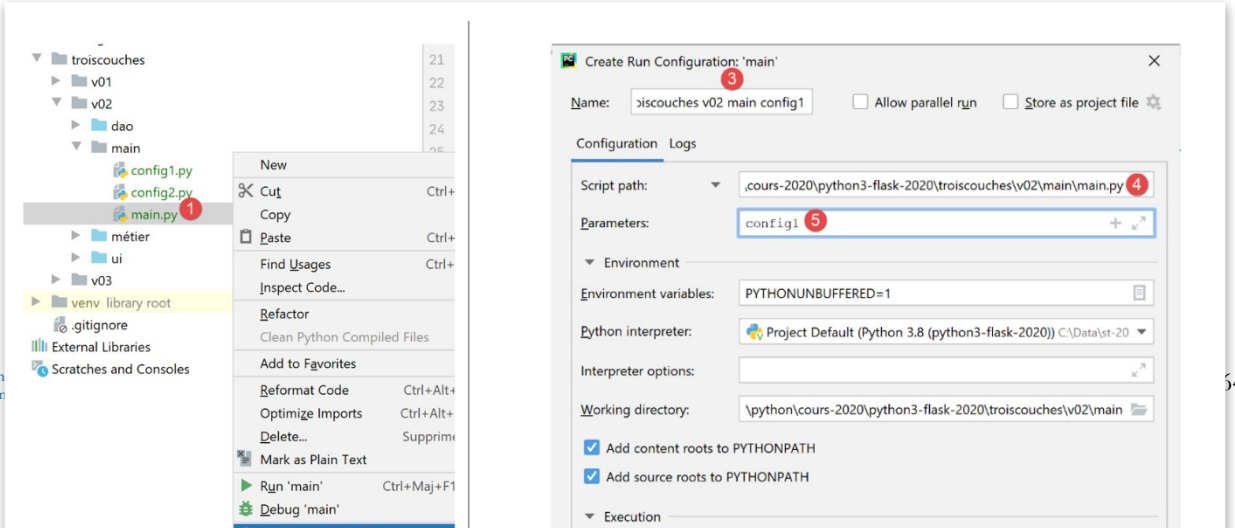
```
1.  # dao
2.      dao = DaoImpl1()
3.      # métier
4.      métier = MétierImpl1()
5.      métier.dao = dao
```

- ligne 5 : la propriété **[MétierImpl1.dao]** est de type **[DaoImpl1]** (ligne 2) ;

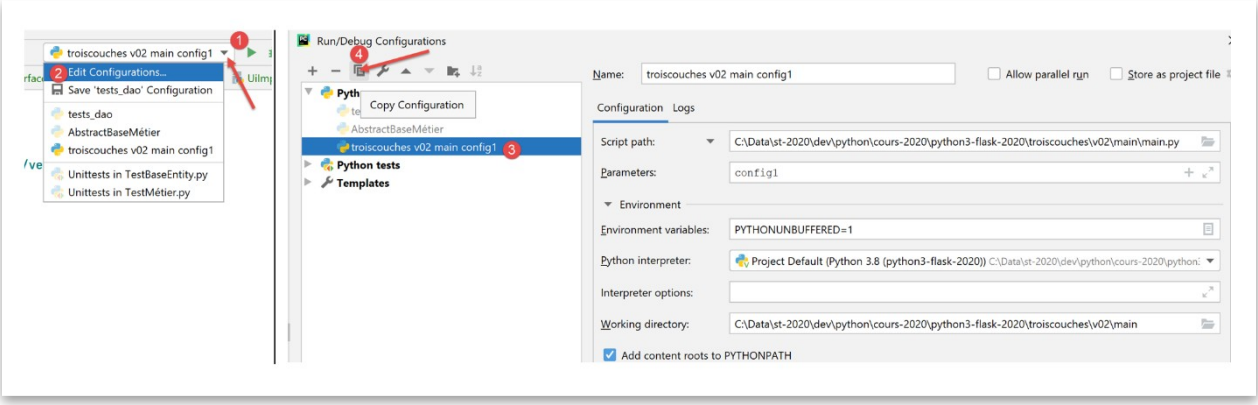
Ce qu'on veut montrer ici, c'est que le script **[main]** n'a pas à se préoccuper des couches **[métier]** et **[dao]**. Il n'a à se préoccuper que de la couche **[ui]**, les liens entre cette couche et les autres ayant été faits par configuration.



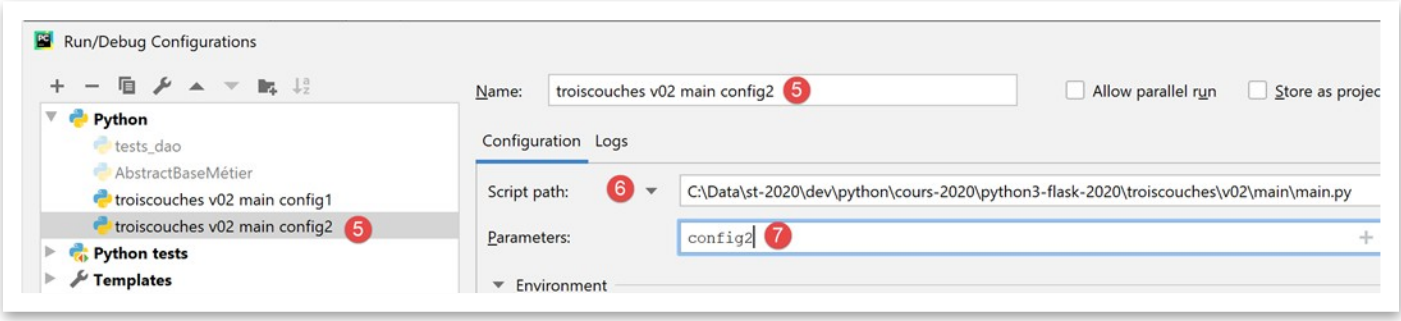
Pour passer le paramètre **[config1]** ou **[config2]** au script **[main]**, on procèdera de la façon suivante :



- en **[1-2]**, on crée ce qu'on appelle une configuration d'exécution ;
- en **[3]**, on donne un nom à cette configuration pour pouvoir la retrouver ;
- en **[4]**, on sélectionne le script à exécuter. Si on a suivi la procédure **[1-2]**, le bon script a déjà été sélectionné ;
- en **[5]**, on met ici les paramètres à transmettre au script. On passe ici la chaîne **[config1]** pour demander au script d'utiliser la configuration n° 1 ;
- en **[6]**, on valide la configuration d'exécution ;

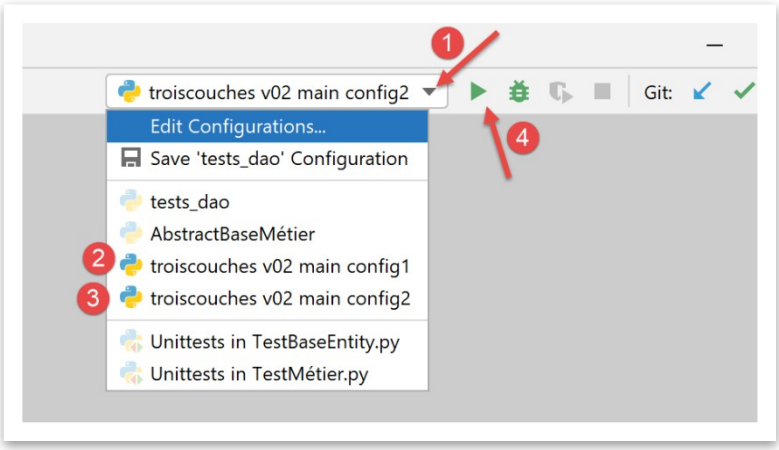


- en **[1-2]**, on demande à voir les contextes d'exécution existants ;
- en **[3]**, on sélectionne le contexte d'exécution existant et on le duplique **[4]** ;



- en **[5]**, le nom donné à la nouvelle configuration. Ce sera celle qui exécute le script **[main]** **[6]** en lui passant le paramètre **[config2]** **[7]** ;

Les configurations d'exécution sont disponibles en haut à droite de la fenêtre PyCharm :



Il suffit de sélectionner **[2]** ou **[3]** puis d'appuyer sur **[4]** pour exécuter le script **[main]** avec l'un ou l'autre des paramètres **[config1]** ou **[config2]**.

Avec **[config1]**, l'exécution de **[main]** donne les résultats suivants :

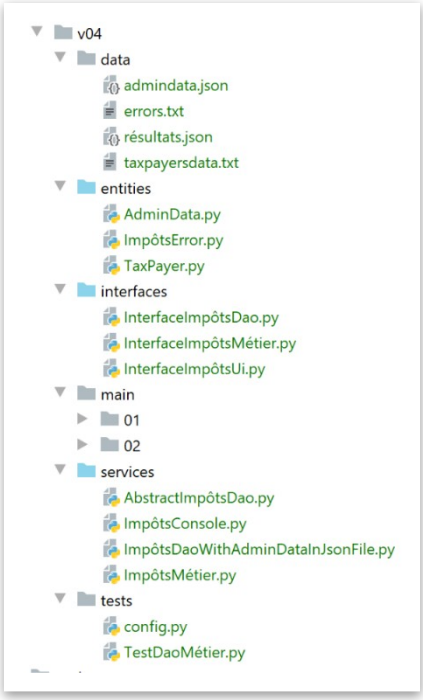
```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/troiscouches/v02/main/main.py config1
2.  34
3.
4.  Process finished with exit code 0
```

Avec **[config2]**, l'exécution de **[main]** donne les résultats suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/troiscouches/v02/main/main.py config2
2.  -10
3.
4.  Process finished with exit code 0
```

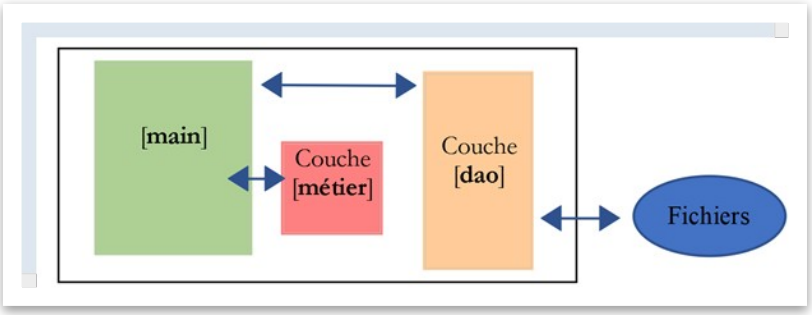
Le lecteur est invité à vérifier ces résultats.

15 Exercice d'application - version 4



Nous reprenons ici l'exercice décrit au paragraphe |Version 3| et nous le traitons maintenant avec des classes et interfaces. Nous écrirons deux applications :

L'application 1 sera la suivante :

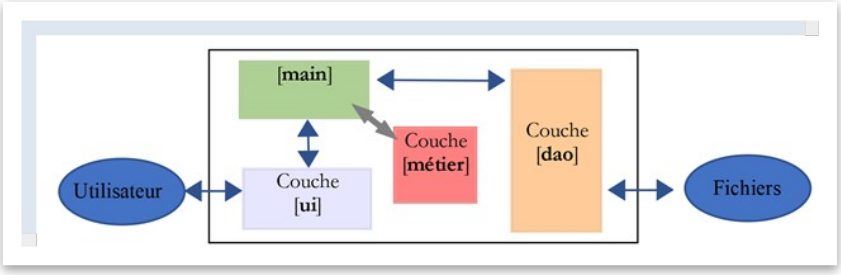


Un script principal **[main]** instanciera une couche **[dao]** et une couche **[métier]** :

- la couche **[dao]** aura pour rôle de gérer des données stockées dans des fichiers texte et ultérieurement dans une base de données ;
- la couche **[métier]** aura pour rôle de faire le calcul de l'impôt ;

Dans cette application, il n'y aura pas d'actions de la part d'un utilisateur : les données des contribuables seront trouvées dans un fichier texte dont on donnera le nom au module **[main].**

Dans l'application 2, c'est l'utilisateur qui tapera au clavier les données des contribuables. L'architecture évoluera alors de la façon suivante :

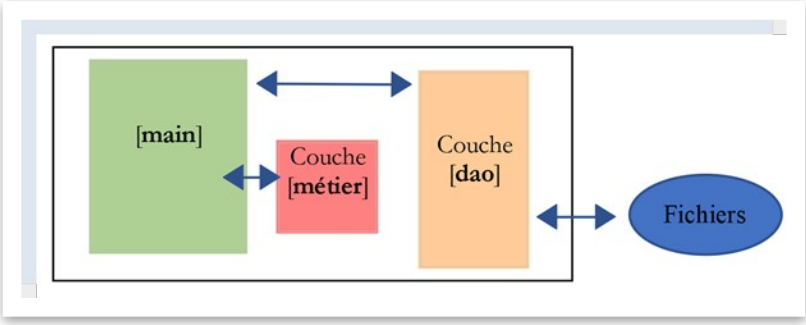


- la couche **[dao]** (**D**ata **A**ccess **O**bject) s'occupe de l'accès aux données externes
- la couche **[métier]** s'occupe des problèmes métier, ici le calcul de l'impôt. Elle ne s'occupe pas des données. Celles-ci peuvent avoir deux provenances :
  - la couche **[dao]** pour les données persistantes ;
  - la couche **[ui]** pour les données fournies par l'utilisateur.
- la couche **[ui]** (**U**ser **I**nterface) s'occupe des interactions avec l'utilisateur ;
- **[main]** est le chef d'orchestre ;

Dans la suite, les couches **[dao]**, **[métier]** et **[ui]** seront chacune implémentée à l'aide d'une classe. Les couches **[métier]** et **[dao]** seront les mêmes pour les deux applications. C'est la raison pour laquelle on les a réunies dans une même version de l'exercice d'application.

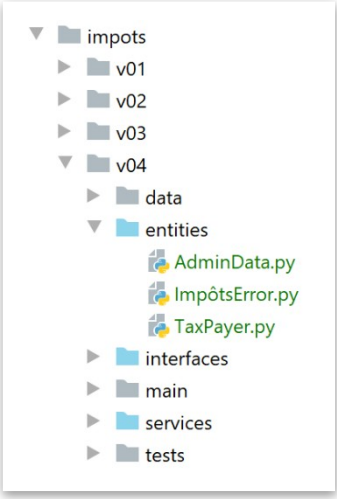
15.1 Version 4 – application 1

La version 4 calcule l'impôt d'une liste de contribuables placée dans un fichier texte. Elle a l'architecture suivante :



15.1.1 Les entités

Les entités sont des classes de données. Leur rôle est d'encapsuler des données et d'offrir des getters / setters qui permettent de



vérifier la validité de celles-ci. Les entités sont échangées par les couches. Une même entité peut partir de la couche **[ui]** pour aller jusqu'à la couche **[dao]** et vice-versa.

15.1.1.1 La classe [ImpôtsError]

Nous utiliserons une classe d'exception propriétaire :

```
1.  # ------------------------------
2.  # classe d'exception
3.  from MyException import MyException
4.
5.
6.  class ImpôtsError(MyException):
7.      pass
```

Dès que les couches **[métier]** et **[dao]** rencontreront un problème, elles lanceront cette exception. Elle dérive de la classe **[MyException]**. Elle s'utilise donc de la façon suivante : **[raise ImpôtsError(code_erreur, msg_erreur)]**.

15.1.1.2 La classe [AdminData]

La classe **[AdminData]** encapsule les constantes intervenant dans le calcul de l'impôt :

```
1.  from BaseEntity import BaseEntity
2.
3.
4.  # données de l'administration fiscale
5.  class AdminData(BaseEntity):
6.      # clés exclues de l'état de la classe
7.      excluded_keys = []
8.
9.      # clés aurorisées
10.     @staticmethod
11.     def get_allowed_keys() -> list:
12.         return [
13.             "limites",
14.             "coeffr",
15.             "coeffn",
16.             "plafond_qf_demi_part",
17.             "plafond_revenus_celibataire_pour_reduction",
18.             "plafond_revenus_couple_pour_reduction",
19.             "valeur_reduc_demi_part",
20.             "plafond_decote_celibataire",
21.             "plafond_decote_couple",
22.             "plafond_impot_couple_pour_decote",
23.             "plafond_impot_celibataire_pour_decote",
24.             "abattement_dixpourcent_max",
25.             "abattement_dixpourcent_min"
26.         ]
```

- ligne 5 : la classe **[AdminData]** étend la classe **[BaseEntity]** décrite au paragraphe |BaseEntity|. On se rappelle que les classes étendant la classe **[BaseEntity]** doivent définir :
  - un attribut de classe **[excluded_keys]** (ligne 7) qui liste les propriétés de l'objet exclues lorsque l'objet est transformé en dictionnaire ;
  - une méthode statique **[get_allowed_keys]** (lignes 10-26) qui rend la liste des propriétés acceptées lorsque l'objet est initialisé avec un dictionnaire ;

On n'a pas utilisé de setters pour vérifier la validité des données utilisées pour initialiser un objet **[AdminData]**. En effet, cet objet est unique et défini par configuration et donc pas susceptible d'être erroné.

15.1.1.3 La classe [TaxPayer]

La classe **[TaxPayer]** modèlisera un contribuable :

```
1.  # imports
2.  from BaseEntity import BaseEntity
3.  from ImpôtsError import ImpôtsError
4.
5.
6.  # un contribuable
7.  class TaxPayer(BaseEntity):
8.      # modélise un contribuable
9.      # id : identifiant
10.     # marié : oui / non
11.     # enfants : son nombre d'enfants
12.     # salaire : son salaire annuel
13.     # impôt : montant de l'impôt à payer
14.     # surcôte : surcôte d'impôt à payer
15.     # décôte : décôte de l'impôt à payer
16.     # réduction : réduction sur l'impôt à payer
17.     # taux : taux d'imposition du contribuable
18.
19.     # clés exclues de l'état de la classe
20.     excluded_keys = []
21.
22.     # clés aurorisées
23.     @staticmethod
24.     def get_allowed_keys() -> list:
25.         return ['id', 'marié', 'enfants', 'salaire', 'impôt', 'surcôte', 'décôte', 'réduction', 'taux']
26.
```

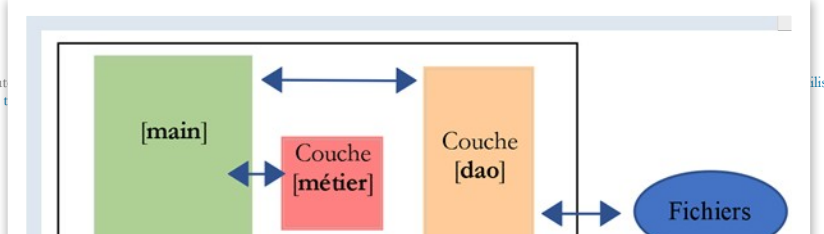
```
27.     # properties
28.     @property
29.     def marié(self) -> str:
30.         return self.__marié
31.
32.     @property
33.     def enfants(self) -> int:
34.         return self.__enfants
35.
36.     @property
37.     def salaire(self) -> int:
38.         return self.__salaire
39.
40.     @property
41.     def impôt(self) -> int:
42.         return self.__impôt
43.
44.     @property
45.     def surcôte(self) -> int:
46.         return self.__surcôte
47.
48.     @property
49.     def décôte(self) -> int:
50.         return self.__décôte
51.
52.     @property
53.     def réduction(self) -> int:
54.         return self.__réduction
55.
56.     @property
57.     def taux(self) -> float:
58.         return self.__taux
59.
60.     # setters
61.     @marié.setter
62.     def marié(self, marié: str):
63.         ok = isinstance(marié, str)
64.         if ok:
65.             marié = marié.strip().lower()
66.             ok = marié == "oui" or marié == "non"
67.         if ok:
68.             self.__marié = marié
69.         else:
70.             raise ImpôtsError(31, f"l'attribut marié [{marié}] doit avoir l'une des valeurs oui / non")
71.
72.     @enfants.setter
73.     def enfants(self, enfants):
74.         # enfants doit être un entier >=0
75.         try:
76.             enfants = int(enfants)
77.             erreur = enfants < 0
78.         except:
79.             erreur = True
80.         if not erreur:
81.             self.__enfants = enfants
82.         else:
83.             raise ImpôtsError(32, f"L'attribut enfants [{enfants}] doit être un entier >=0")
84.
85.     @salaire.setter
86.     def salaire(self, salaire):
87.         # salaire doit être un entier >=0
88.         try:
89.             salaire = int(salaire)
90.             erreur = salaire < 0
91.         except:
92.             erreur = True
93.         if not erreur:
94.             self.__salaire = salaire
95.         else:
96.             raise ImpôtsError(33, f"L'attribut salaire [{salaire}] doit être un entier >=0")
97.
98.     @impôt.setter
99.     def impôt(self, impôt):
100.        # impôt doit être un entier >=0
101.        try:
102.            impôt = int(impôt)
103.            erreur = impôt < 0
```

```
104.         except:
105.             erreur = True
106.         if not erreur:
107.             self.__impôt = impôt
108.         else:
109.             raise ImpôtsError(34, f"L'attribut impôt [{impôt}] doit être un nombre >=0")
110.
111.     @décôte.setter
112.     def décôte(self, décôte):
113.         # décôte doit être un entier >=0
114.         try:
115.             décôte = int(décôte)
116.             erreur = décôte < 0
117.         except:
118.             erreur = True
119.         if not erreur:
120.             self.__décôte = décôte
121.         else:
122.             raise ImpôtsError(35, f"L'attribut décôte [{décôte}] doit être un nombre >=0")
123.
124.     @surcôte.setter
125.     def surcôte(self, surcôte):
126.         # surcôte doit être un entier >=0
127.         try:
128.             surcôte = int(surcôte)
129.             erreur = surcôte < 0
130.         except:
131.             erreur = True
132.         if not erreur:
133.             self.__surcôte = surcôte
134.         else:
135.             raise ImpôtsError(36, f"L'attribut surcôte [{surcôte}] doit être un nombre >=0")
136.
137.     @réduction.setter
138.     def réduction(self, réduction):
139.         # surcôte doit être un entier >=0
140.         try:
141.             réduction = int(réduction)
142.             erreur = réduction < 0
143.         except:
144.             erreur = True
145.         if not erreur:
146.             self.__réduction = réduction
147.         else:
148.             raise ImpôtsError(37, f"L'attribut réduction [{réduction}] doit être un nombre >=0")
149.
150.     @taux.setter
151.     def taux(self, taux):
152.         # taux doit être un réel >=0
153.         try:
154.             taux = float(taux)
155.             erreur = taux < 0
156.         except:
157.             erreur = True
158.         if not erreur:
159.             self.__taux = taux
160.         else:
161.             raise ImpôtsError(38, f"L'attribut taux [{taux}] doit être un nombre >=0")
162.
```

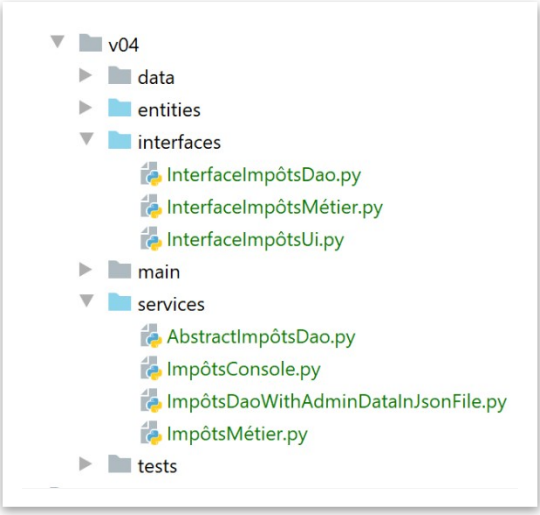
Notes :

- la classe **`[TaxPayer]`** encapsule un contribuable ;
- ligne 7 : la classe **`[TaxPayer]`** dérive de la classe **`[BaseEntity]`**. Elle a donc un identifiant **`[id]`** ;
- ligne 20 : aucune propriété n'est exclue de l'état d'un objet **`[AdminData]`** ;
- lignes 22-25 : les propriétés de la classe. Celles-ci sont explicitées aux lignes 9-17 ;
- lignes 27-58 : getters des attributs de la classe ;
- lignes 60-161 : les setters des attributs de la classe. On rappelle que l'intérêt d'une classe encapsulant des données vis-à-vis d'un simple dictionnaire est que la classe peut vérifier la validité de ses propriétés grâce à ses setters ;

15.1.2 La couche `[dao]`



Nous allons réunir les implémentations des couches dans un dossier **[services]**. Ces classes implémenteront des interfaces définies dans le dossier **[interfaces]**.



15.1.2.1 L'interface [InterfaceImpôtsDao]

La couche **[dao]** implémentera l'interface **[InterfaceImpôtsDao]** suivante (fichier *InterfaceImpôtsDao.py*) :

```
1.  # imports
2.  from abc import ABC, abstractmethod
3.
4.
5.  # interface IImpôtsDao
6.  from AdminData import AdminData
7.
8.
9.  class InterfaceImpôtsDao(ABC):
10.     # liste des tranches de l'impôt
11.     @abstractmethod
12.     def get_admindata(self) -> AdminData:
13.         pass
14.
15.     # liste des données contribuables
16.     @abstractmethod
17.     def get_taxpayers_data(self) -> dict:
18.         pass
19.
20.     # écriture des résultats du calcul de l'impôt
21.     @abstractmethod
22.     def write_taxpayers_results(self, taxpayers_results: list):
23.         pass
```

L'interface définit trois méthodes :

- **[get_admindata]** : est la méthode qui obtient le tableau des tranches d'impôt. On notera qu'on ne donne aucun renseignement sur la façon d'obtenir ces données. Dans la suite, elles seront trouvées d'abord dans un fichier texte puis dans une base de données. Ce seront aux classes qui implémentent l'interface de s'adapter au mode de stockage des données. On aura donc une classe pour récupérer les tranches d'impôt dans un fichier texte et une autre pour les récupérer dans une base de données. Elles implémenteront toutes deux la méthode **[get_admindata]** ;
- **[get_taxpayers_data]** : est la méthode qui obtient les données des contribuables. Là encore nous ne disons pas où elles seront trouvées. Nous ne traiterons que le cas où elles sont dans un fichier texte ;
- **[write_taxpayers_results]** : est la méthode qui va persister les résultats du calcul de l'impôt. Nous ne disons pas où. Nous ne traiterons que le cas où les résultats sont persistés dans un fichier texte. Le paramètre **[taxpayers_results]** sera la liste des résultats à persister ;

15.1.2.2 La classe [AbstractImpôtsDao]

La couche **[dao]** va être implémentée par deux classes :

- l'une ira chercher les données (contribuables, résultats, tranches d'impôt) dans des fichiers texte ;

- l'autre ira chercher les données (contribuables, résultats) dans des fichiers texte et les tranches de l'impôt dans une base de données ;

Les deux classes ne vont différer que par la gestion des tranches de l'impôt. Les données contribuables et les résultats des calculs de l'impôt seront, elles, gérées de la même façon. Pour cette raison, nous allons les gérer dans une classe parent **[AbstractImpôtsDao]**. La particularité de la gestion des tranches d'impôt sera, elle, gérée dans deux classes filles :

- la classe **[ImpôtsDaoWithAdminDataInJsonFile]** ira chercher les tranches de l'impôt dans un fichier texte au format jSON ;
- la classe **[ImpôtsDaoWithAdminDataInDatabase]** ira chercher les tranches de l'impôt dans une base de données ;

La classe parent **[AbstractImpôtsDao]** sera la suivante :

```
1.  # imports
2.  import codecs
3.  import json
4.  from abc import abstractmethod
5.
6.  from AdminData import AdminData
7.  from ImpôtsError import ImpôtsError
8.  from InterfaceImpôtsDao import InterfaceImpôtsDao
9.  from TaxPayer import TaxPayer
10.
11.
12. # classe de base pour la couche [dao]
13. class AbstractImpôtsDao(InterfaceImpôtsDao):
14.     # les contribuables et leur impôt seront dans des fichiers texte
15.     # constructeur
16.     def __init__(self, config: dict):
17.         # config[taxpayersFilename] : le nom du fichier texte des contribuables
18.         # config[resultsFilename] : le nom du fichier jSON des résultats
19.         # config[errorsFilename] : le nom du fichier des erreurs
20.
21.         # on mémorise les paramètres
22.         self.taxpayers_filename = config.get("taxpayersFilename")
23.         self.taxpayers_results_filename = config.get("resultsFilename")
24.         self.errors_filename = config.get("errorsFilename")
25.
26.     # ------------------
27.     # interface IImpôtsDao
28.     # ------------------
29.
30.     # liste des données contribuables
31.     def get_taxpayers_data(self) -> dict:
32.         …
33.
34.     # écriture de l'impôt des contribuables
35.     def write_taxpayers_results(self, taxpayers: list):
36.         …
37.
38.     # lecture des tranches de l'impôt
39.     @abstractmethod
40.     def get_admindata(self) -> AdminData:
41.         pass
```

- ligne 13 : la classe **[AbstractImpôtsDao]** implémente l'interface **[InterfaceImpôtsDao]**. Aussi trouve-t-on les trois méthodes de cette interface :
  - **[get_taxpayers_data]** : ligne 31 ;
  - **[write_taxpayers_results]** : ligne 35 ;
  - **[get_admindata]** : ligne 40. Cette méthode ne sera pas implémentée par la classe **[AbstractImpôtsDao]** aussi est-elle déclarée abstraite (ligne 39) ;

- ligne 16 : le constructeur reçoit un dictionnaire **[config]** contenant les informations suivantes :
  - **[taxpayersFilename]** : le nom du fichier texte qui contient les données des contribuables ;
  - **[resultsFilename]** : le nom du fichier texte dans lequel seront mis les résultats ;
  - **[errorsFilename]** : le nom du fichier texte listant les erreurs rencontrées lors de l'exploitation du fichier **[taxpayersFilename]** ;

La méthode **[get_taxpayers_data]** est la suivante :

```
1.      # liste des données contribuables
2.      def get_taxpayers_data(self) -> dict:
3.          # initialisations
```

```
4.          taxpayers_data = []
5.          datafile = None
6.          erreurs = []
7.          try:
8.              # ouverture du fichier des données
9.              datafile = open(self.taxpayers_filename, "r")
10.             # on exploite la ligne courante du fichier
11.             ligne = datafile.readline()
12.             # n° de ligne
13.             numligne = 0
14.             while ligne != '':
15.                 # une ligne de +
16.                 numligne += 1
17.                 # on enlève les blancs
18.                 ligne = ligne.strip()
19.                 # on ignore les lignes vides et les commentaires
20.                 if ligne != "" and ligne[0] != "#":
21.                     try:
22.                         # on récupère les 4 champs id,marié,enfants,salaire qui forment la ligne contribuable
23.                         (id, marié, enfants, salaire) = ligne.split(",")
24.                         # on crée un nouveau TaxPayer
25.                         taxpayers_data.append(
26.                             TaxPayer().fromdict({'id': id, 'marié': marié, 'enfants': enfants, 'salaire': salaire}))
27.                     except BaseException as erreur:
28.                         # on note l'erreur
29.                         erreurs.append(f"Ligne {numligne}, {erreur}")
30.                 # on lit une nouvelle ligne contribuable
31.                 ligne = datafile.readline()
32.             # on enregistre les erreurs s'il y en a
33.             if erreurs:
34.                 text = f"Analyse du fichier {self.taxpayers_filename}\n\n" + "\n".join(erreurs)
35.                 with codecs.open(self.errors_filename, "w", "utf-8") as fd:
36.                     fd.write(text)
37.             # on rend le résultat
38.             return {"taxpayers": taxpayers_data, "erreurs": erreurs}
39.         except BaseException as erreur:
40.             # on lance une exception ImpôtsError
41.             raise ImpôtsError(11, f"{erreur}")
42.         finally:
43.             # on ferme le fichier
44.             if datafile:
45.                 datafile.close()
```

- ligne 4 : les données des contribuables (marié, enfants, salaire) seront placées dans une liste d'objets de type **[TaxPayer]** ;
- lignes 8-9 : on ouvre le fichier texte des contribuables en lecture. Son contenu a la forme suivante :

```
1.  # données valides : id, marié, enfants, salaire
2.  1,oui,2,55555
3.  2,oui,2,50000
4.  3,oui,3,50000
5.  4,non,2,100000
6.  5,non,3,100000
7.  6,oui,3,100000
8.  7,oui,5,100000
9.  8,non,0,100000
10. 9,oui,2,30000
11. 10,non,0,200000
12. 11,oui,3,200000
13. # on peut avoir des lignes vides
14.
15. # on crée des lignes erronées
16. # pas assez de valeurs
17. 11,12
18. # trop de valeurs
19. 12,oui,3,200000, x, y
20. # des valeurs erronées
21. x,x,x,x
```

Par rapport aux versions précédentes :

- chaque ligne du fichier **[taxpayersFilename]** commence par l'identifiant du contribuable, un simple numéro ;
- les commentaires et les lignes vides sont autorisés ;
- on va gérer les erreurs. Ainsi les lignes 17, 19 et 21 doivent être déclarées erronées. Les erreurs sont consignées dans un fichier à part ;

Continuons l'examen du code :

- ligne 4 : les données du fichier texte sont transférées dans la liste **[taxPayersData]** ;
- lignes 14-31 : le fichier des contribuables est lu ligne par ligne ;

- ligne 14 : la fin du fichier est atteinte lorsqu'on lit une ligne vide (rien – pas même la marque de fin de ligne \r\n) ;
- ligne 20 : on ignore les lignes vides et les commentaires. Une ligne est un commentaire si, une fois la ligne débarrassée de ses blancs devant et derrière le texte, le 1er caractère est le caractère # ;
- ligne 24 : une ligne correcte est composée de quatre champs séparés par une virgule. On récupère ceux-ci. L'affectation de données à un tuple de quatre éléments échoue s'il n'y a pas exactement quatre données affectées ;
- ligne 25 : si l'un des quatre champs récupérés **[id, marié, enfants, salaire]** est invalide alors la méthode **[BaseEntity.fromdict]** lancera une exception de type **[MyException]** ;
- lignes 25-26 : un objet **[TaxPayer]** est ajouté à la liste **[taxpayers_data]** des contribuables ;
- lignes 27-29 : les éventuelles erreurs sont cumulées dans une liste **[erreurs]**. Cette liste a été créée ligne 6 ;
- lignes 33-36 : la liste des erreurs rencontrées est enregistrée dans le fichier texte **[errorsFilename]**. Elles sont de deux types :
  - une ligne n'avait pas le nombre correct de champs attendus ;
  - les informations de la ligne étaient erronées et ont échoué à construire un objet **[TaxPayer]** ;
- lignes 39-41 : on intercepte toute erreur (BaseException) et on la remonte en l'encapsulant dans un type **[ImpôtsError]** ;
- lignes 42-45 : dans tous les cas, réussite ou échec, le fichier texte des contrbuables est fermé s'il a été ouvert ;

La méthode **[write_taxpayers_results]** doit produire un fichier jSON de la forme :

```
1.  [
2.    {
3.      "id": 1,
4.      "marié": "oui",
5.      "enfants": 2,
6.      "salaire": 55555,
7.      "impôt": 2814,
8.      "surcôte": 0,
9.      "taux": 0.14,
10.     "décôte": 0,
11.     "réduction": 0
12.   },
13.   {
14.     "id": 2,
15.     "marié": "oui",
16.     "enfants": 2,
17.     "salaire": 50000,
18.     "impôt": 1384,
19.     "surcôte": 0,
20.     "taux": 0.14,
21.     "décôte": 384,
22.     "réduction": 347
23.   },
24.   {
25.     "id": 3,
26.     "marié": "oui",
27.     "enfants": 3,
28.     "salaire": 50000,
29.     "impôt": 0,
30.     "surcôte": 0,
31.     "taux": 0.14,
32.     "décôte": 720,
33.     "réduction": 0
34.   },
35.   …
36. ]
```

La méthode **[write_taxpayers_results]** est la suivante :

```
1.      # écriture de l'impôt des contribuables
2.      def write_taxpayers_results(self, taxpayers: list):
3.          # écriture des résultats dans un fichier jSON
4.          # taxpayers : liste d'objets de type TaxPayer
5.          # (id, marié, enfants, salaire, impôt, surcôte, décôte, réduction, taux)
6.          # la liste [taxpayers] est enregistrée dans le fichier texte [self.taxpayers_results_filename]
7.          file = None
8.          try:
9.              # ouverture du fichier des résultats
10.             file = codecs.open(self.taxpayers_results_filename, "w", "utf8")
11.             # création de la liste à sérialiser en jSON
12.             mapping = map(lambda taxpayer: taxpayer.asdict(), taxpayers)
13.             # sérialisation jSON
14.             json.dump(list(mapping), file, ensure_ascii=False)
15.         except BaseException as erreur:
16.             # on relance l'erreur sous un autre type
17.             raise ImpôtsError(12, f"{erreur}")
```

```
18.         finally:
19.             # on ferme le fichier s'il a été ouvert
20.             if file:
21.                 file.close()
```

- ligne 2 : la méthode reçoit une liste de contribuables **[taxpayers]** qu'elle doit enregistrer dans le fichier texte **[self.taxpayers_results_filename]** au format jSON ;
- ligne 10 : création du fichier UTF-8 des résultats ;
- ligne 12 : nous introduisons ici la fonction **[map]** dont la syntaxe ici est **[map (fonction, liste1)]**. La fonction **[fonction]** est appliquée à chaque élément de **[liste1]** et produit un nouvel élément qui alimente une liste **[liste2]**. Finalement, pour chaque i :

  ```
  liste2[i]=fonction(liste1[i])
  ```

  Ici, **[liste1]** est la liste **[taxPayers]**, une liste d'objets de type **[TaxPayer]**. La fonction **[fonction]** est exprimée ici sous la forme d'une fonction dite **[lambda]** qui exprime la transformation faite sur un élément **[taxpayer]** de la liste **[taxpayers]** : chaque élément **[taxpayer]** est remplacé par son dictionnaire **[taxpayer.asdict()]**. Finalement, la liste **[liste2]** obtenue est la liste des dictionnaires des éléments de la liste **[taxpayers]** ;
- ligne 12 : le résultat rendu par la fonction **[map]** n'est pas la liste **[liste2]** mais un objet de type **[map]**. Pour avoir **[liste2]**, il faut utiliser l'expression **[list(mapping)]** (ligne 14) ;
- ligne 14 : la liste **[liste2]** est enregistrée au format jSON dans le fichier **[self.taxpayers_results_filename]** ;
- lignes 15-17 : tout type d'exception est intercepté et encapsulé dans une erreur de type **[ImpôtsError]** avant d'être relancé (ligne 17) ;
- lignes 19-21 : dans tous les cas, réussite ou échec, le fichier des résultats est fermé s'il a été ouvert ;

15.1.2.3 Classe [ImpôtsDaoWithAdminDataInJsonFile]

La classe **[ImpôtsDaoWithAdminDataInJsonFile]** va dériver de la classe **[AbstractImpôtsDao]** et implémenter la méthode **[getAdminData]** que sa classe parent n'a pas implémentée. Elle ira chercher les données de l'administration fiscale dans un fichier jSON :

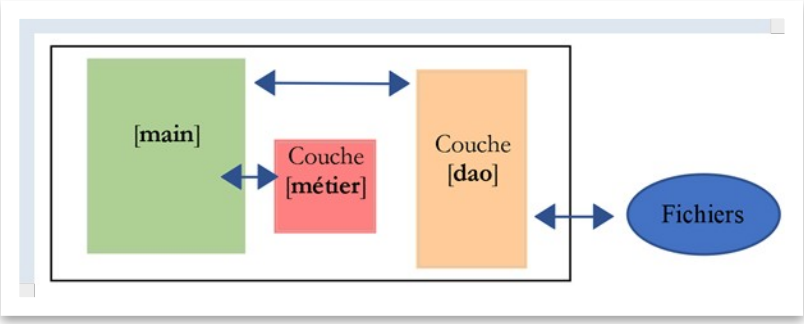
```
1.  {
2.      "limites": [9964, 27519, 73779, 156244, 0],
3.      "coeffr": [0, 0.14, 0.3, 0.41, 0.45],
4.      "coeffn": [0, 1394.96, 5798, 13913.69, 20163.45],
5.      "plafond_qf_demi_part": 1551,
6.      "plafond_revenus_celibataire_pour_reduction": 21037,
7.      "plafond_revenus_couple_pour_reduction": 42074,
8.      "valeur_reduc_demi_part": 3797,
9.      "plafond_decote_celibataire": 1196,
10.     "plafond_decote_couple": 1970,
11.     "plafond_impot_couple_pour_decote": 2627,
12.     "plafond_impot_celibataire_pour_decote": 1595,
13.     "abattement_dixpourcent_max": 12502,
14.     "abattement_dixpourcent_min": 437
15. }
```

La classe **[ImpôtsDaoWithAdminDataInJsonFile]** est la suivante :

```
1.  # imports
2.  import codecs
3.  import json
4.
5.  from AbstractImpôtsDao import AbstractImpôtsDao
6.  from AdminData import AdminData
7.  from ImpôtsError import ImpôtsError
8.
9.
10. # une implémentation de la couche [dao] où les données de l'administration fiscale sont dans un fichier jSON
11. class ImpôtsDaoWithAdminDataInJsonFile(AbstractImpôtsDao):
12.     # constructeur
13.     def __init__(self, config: dict):
14.         # config[admindataFilename] : le nom du fichier jSON contenant les données de l'administration fiscale
15.         # config[taxpayersFilename] : le nom du fichier texte des contribuables
16.         # config[resultsFilename] : le nom du fichier jSON des résultats
17.         # config[errorsFilename] : le nom du fichier des erreurs
18.
19.         # initialisation de la classe Parent
20.         AbstractImpôtsDao.__init__(self, config)
21.         # lecture des données de l'administration fiscale
22.         file = None
23.         try:
24.             # ouverture du fichier jSON des données fiscales en lecture
25.             file = codecs.open(config["admindataFilename"], "r", "utf8")
```

```
26.             # transfert du contenu du fichier jSON dans un objet [AdminData]
27.             self.admindata = AdminData().fromdict(json.load(file))
28.         except BaseException as erreur:
29.             # on relance l'erreur sous la forme d'un type [ImpôtsError]
30.             raise ImpôtsError(21, f"{erreur}")
31.         finally:
32.             # fermeture du fichier s'il a été ouvert
33.             if file:
34.                 file.close()
35.
36.     # -------------
37.     # interface
38.     # -------------
39.
40.     # récupération des données de l'administration fiscale
41.     # la méthode rend un objet [AdminData]
42.     def get_admindata(self) -> AdminData:
43.         return self.admindata
```

- ligne 11 : la classe **[ImpôtsDaoWithAdminDataInJsonFile]** hérite de la classe **[AbstractImpôtsDao]**. A ce titre elle implémente l'interface **[InterfaceImpôtsDao]** ;
- ligne 13 : le constructeur reçoit en paramètre un dictionnaire contenant les informations des lignes 14-17 ;
- ligne 20 : la classe parent est initialisée ;
- ligne 24 : ouverture du fichier jSON des données de l'administration fiscale ;
- ligne 25 : le fichier UTF-8 des données de l'administration fiscale est ouvert ;
- ligne 27 : le contenu du fichier est lu et placé dans l'objet **[self.admindata]** de type **[AdminData]**. Il faut que les clés du fichier jSON correspondent aux propriétés acceptées pour un objet **[AdminData]** sinon la méthode **[fromdict]** lancera une exception ;
- lignes 28-30 : gestion des exceptions. Les exceptions qui peuvent se produire sont encapsulées dans un type **[ImpôtsError]** avant d'être relancées ;
- lignes 32-34 : le fichier est fermé s'il a été ouvert ;
- lignes 42-43 : implémentation de la méthode **[get_admindata]** de l'interface **[InterfaceImpôtsDao]** ;



15.1.3 La couche [métier]

15.1.3.1 L'interface [InterfaceImpôtsMétier]

L'interface de la couche **[métier]** sera la suivante :

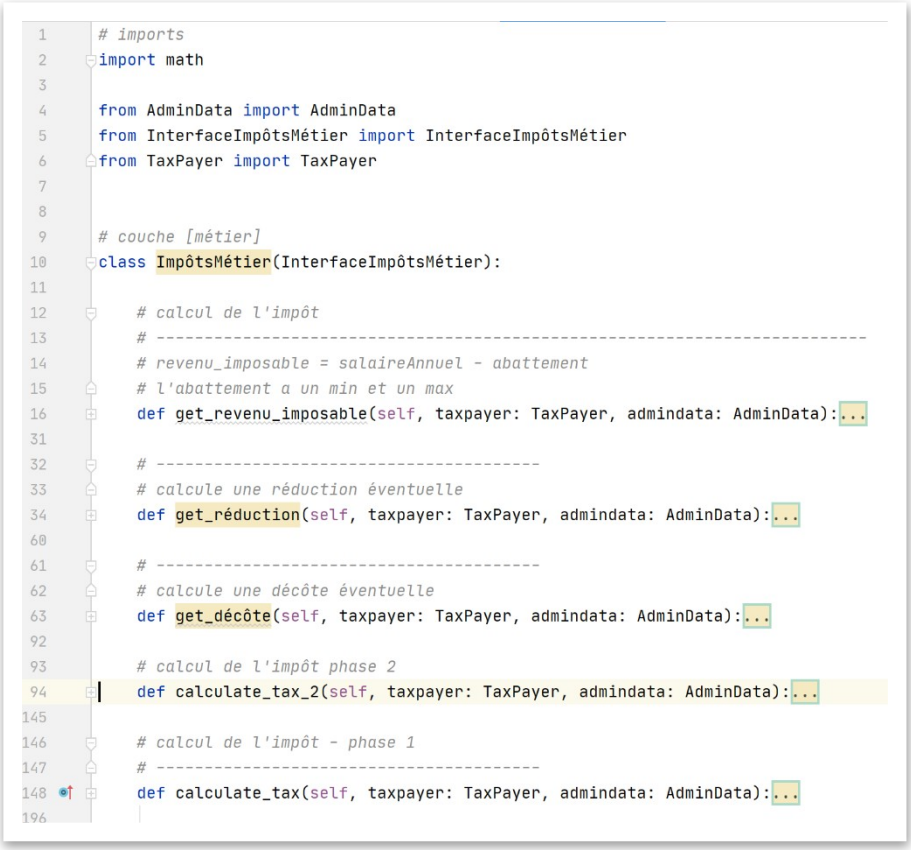
```
1.  # imports
2.  from abc import ABC, abstractmethod
3.
4.  from AdminData import AdminData
5.  from TaxPayer import TaxPayer
6.
7.
8.  # interface IImpôtsMétier
9.  class InterfaceImpôtsMétier(ABC):
10.     # calcul de l'impôt pour 1 contribuable
11.     @abstractmethod
12.     def calculate_tax(self, taxpayer: TaxPayer, admindata: AdminData):
13.         pass
```

- l'interface **[InterfaceImpôtsMétier]** définit une unique méthode :
  - ligne 12 : la méthode **[calculate_tax]** permet de calculer l'impôt d'un unique contribuable **[taxpayer]**. **[admindata]** est l'objet **[AdminData]** encapsulant les données de l'administration fiscale ;

- ligne 12 : la méthode **[calculate_tax]** ne rend pas de résultat. Les données obtenues (impôt, surcôte, décôte, réduction, taux) sont incluses dans le paramètre **[taxpayer]** : avant l'appel ces attributs sont vides, après l'appel ils ont été initialisés ;

15.1.3.2 La classe [ImpôtsMétier]

La classe **[ImpôtsMétier]** implémente l'interface **[InterfaceImpôtsMétier]** de la façon suivante :



```
1   # imports
2   import math
3
4   from AdminData import AdminData
5   from InterfaceImpôtsMétier import InterfaceImpôtsMétier
6   from TaxPayer import TaxPayer
7
8
9   # couche [métier]
10  class ImpôtsMétier(InterfaceImpôtsMétier):
11
12      # calcul de l'impôt
13      # ----------------------------------------------------------------------
14      # revenu_imposable = salaireAnnuel - abattement
15      # l'abattement a un min et un max
16      def get_revenu_imposable(self, taxpayer: TaxPayer, admindata: AdminData):...
31
32      # ---------------------------------------
33      # calcule une réduction éventuelle
34      def get_réduction(self, taxpayer: TaxPayer, admindata: AdminData):...
60
61      # ---------------------------------------
62      # calcule une décôte éventuelle
63      def get_décôte(self, taxpayer: TaxPayer, admindata: AdminData):...
92
93      # calcul de l'impôt phase 2
94      def calculate_tax_2(self, taxpayer: TaxPayer, admindata: AdminData):...
145
146     # calcul de l'impôt - phase 1
147     # ---------------------------------------
148     def calculate_tax(self, taxpayer: TaxPayer, admindata: AdminData):...
196
```

Les méthodes de la classe sont issues du module **[impôts_module_02]** du paragraphe |Le module [impots.v02.modules.impôts_module_02]|. On a seulement limité les paramètres des méthodes à deux :

- taxpayer(id, marié, enfants, salaire, impôt, décôte, surcôte, réduction, taux) : l'objet représentant un contribuable et son impôt ;
- admindata : l'objet encapsulant les données de l'administration fiscale ;

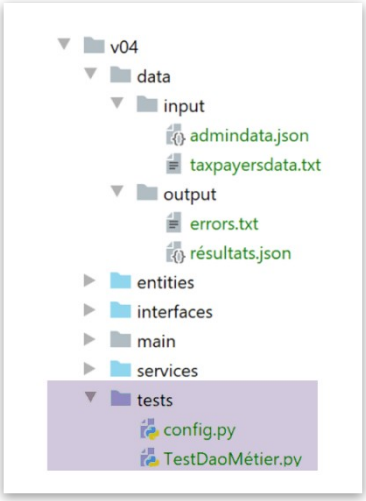
Nous montrons sur une méthode les changements ainsi apportés ;

```
1.      # calcul de l'impôt - phase 1
2.      # ---------------------------------------
3.      def calculate_tax(self, taxpayer: TaxPayer, admindata: AdminData):
4.          # taxpayer(id, marié, enfants, salaire, impôt, décôte, surcôte, réduction, taux)
5.          # admindata : données de l'administration fiscale
6.
7.          # calcul de l'impôt avec enfants
8.          self.calculate_tax_2(taxpayer, admindata)
9.          # les résultats sont dans taxpayer
10.         taux1 = taxpayer.taux
11.         surcôte1 = taxpayer.surcôte
12.         impot1 = taxpayer.impôt
13.         # calcul de l'impôt sans les enfants
14.         if taxpayer.enfants != 0:
15.             # calcul de l'impôt pour le même contribuable sans enfants
16.             taxpayer2 = TaxPayer().fromdict(
17.                 {'id': 0, 'marié': taxpayer.marié, 'enfants': 0, 'salaire': taxpayer.salaire})
18.             self.calculate_tax_2(taxpayer2, admindata)
19.             # les résultats sont dans taxpayer2
20.             taux2 = taxpayer2.taux
21.             surcôte2 = taxpayer2.surcôte
22.             impot2 = taxpayer2.impôt
```

```
23.             # application du plafonnement du quotient familial
24.             if taxpayer.enfants < 3:
25.                 # PLAFOND_QF_DEMI_PART euros pour les 2 premiers enfants
26.                 impot2 = impot2 - taxpayer.enfants * admindata.plafond_qf_demi_part
27.             else:
28.                 # PLAFOND_QF_DEMI_PART euros pour les 2 premiers enfants, le double pour les suivants
29.                 impot2 = impot2 - 2 * admindata.plafond_qf_demi_part - (taxpayer.enfants - 2) \
30.                         * 2 * admindata.plafond_qf_demi_part
31.         else:
32.             # si le contribuable n'a pas d'enfants alors impot2=impot1
33.             impot2 = impot1
34.
35.         # on prend l'impôt le plus fort avec le taux et la surcôte qui vont avec
36.         (impot, surcôte, taux) = (impot1, surcôte1, taux1) if impot1 >= impot2 else (
37.             impot2, impot2 - impot1 + surcôte2, taux2)
38.
39.         # résultats partiels
40.         taxpayer.impôt = impot
41.         taxpayer.surcôte = surcôte
42.         taxpayer.taux = taux
43.         # calcul d'une éventuelle décôte
44.         self.get_décôte(taxpayer, admindata)
45.         taxpayer.impôt -= taxpayer.décôte
46.         # calcul d'une éventuelle réduction d'impôts
47.         self.get_réduction(taxpayer, admindata)
48.         taxpayer.impôt -= taxpayer.réduction
49.         # résultat
50.         taxpayer.impôt = math.floor(taxpayer.impôt)
```

- ligne 3: la méthode **[calculate_tax]** est l'unique méthode de l'interface **[InterfaceImpôtsMétier]**. Elle admet deux paramètres :
  - **[tapPayer]** : le contribuable dont on calcule l'impôt ;
  - **[admindata]** : l'objet encapsulant les données de l'administration fiscale ;
  - les résultats du calcul sont encapsulés dans le paramètre **[taxpayer]** (lignes 40-50). Le contenu de cet objet n'est donc pas le même avant et après l'appel à la méthode ;

15.1.4 Tests des couches [dao] et [métier]



- **[TestDaoMétier]** est la classe UnitTest de test des couches **[dao]** et **[métier]** ;
- **[config]** est le fichier de configuration des tests ;

La configuration **[config]** est la suivante :

```
1.  def configure():
2.      import os
3.
4.      # étape 1 ------
5.      # configuration du python Path
6.
7.      # dossier de ce fichier
8.      script_dir = os.path.dirname(os.path.abspath(__file__))
9.
```

```
10.     # root_dir
11.     root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes"
12.
13.     # dépendances absolues de l'application
14.     absolute_dependencies = [
15.         f"{script_dir}/../entities",
16.         f"{script_dir}/../interfaces",
17.         f"{script_dir}/../services",
18.         f"{root_dir}/02/entities",
19.     ]
20.
21.     # on fixe le syspath
22.     from myutils import set_syspath
23.     set_syspath(absolute_dependencies)
24.
25.     # étape 2 ------
26.     # configuration de l'application
27.     config = {
28.         # chemins absolus des fichiers de l'application
29.         "admindataFilename": f"{script_dir}/../data/input/admindata.json"
30.     }
31.
32.     # instanciation des couches de l'application
33.     from ImpôtsDaoWithAdminDataInJsonFile import ImpôtsDaoWithAdminDataInJsonFile
34.     from ImpôtsMétier import ImpôtsMétier
35.
36.     dao = ImpôtsDaoWithAdminDataInJsonFile(config)
37.     métier = ImpôtsMétier()
38.
39.     # on met les instances de couches dans la config
40.     config["dao"] = dao
41.     config["métier"] = métier
42.
43.     # on rend la config
44.     return config
```

- lignes 4-23 : on configure le Python Path des tests ;
- lignes 32-41 : on instancie les couches **[dao]** et **[métier]**. On met leurs références dans le dictionnaire **[config]** ;
- ligne 44 : on rend ce dictionnaire ;

La classe de test **[TestDaoMétier]** est la suivante :

```
1.  import unittest
2.
3.
4.  def get_config() -> dict:
5.      # configuration de l'application
6.      import config
7.      # on rend la configuration
8.      return config.configure()
9.
10.
11. class TestDaoMétier(unittest.TestCase):
12.
13.     # exécutée avant chaque méthode test_
14.     def setUp(self) -> None:
15.         # on récupère la configuration des tests
16.         config = get_config()
17.         # on mémorise quelques informations
18.         self.métier = config['métier']
19.         self.admindata = config['dao'].get_admindata()
20.
21.     def test_1(self) -> None:
22.         from TaxPayer import TaxPayer
23.
24.         # {'marié': 'oui', 'enfants': 2, 'salaire': 55555,
25.         # 'impôt': 2814, 'surcôte': 0, 'décôte': 0, 'réduction': 0, 'taux': 0.14}
26.         taxpayer = TaxPayer().fromdict({"marié": "oui", "enfants": 2, "salaire": 55555})
27.         self.métier.calculate_tax(taxpayer, self.admindata)
28.         # vérification
29.         self.assertAlmostEqual(taxpayer.impôt, 2815, delta=1)
30.         self.assertEqual(taxpayer.décôte, 0)
31.         self.assertEqual(taxpayer.réduction, 0)
32.         self.assertAlmostEqual(taxpayer.taux, 0.14, delta=0.01)
33.         self.assertEqual(taxpayer.surcôte, 0)
34.
```

```
35.     def test_2(self) -> None:
36.         from TaxPayer import TaxPayer
37.
38.         # {'marié': 'oui', 'enfants': 2, 'salaire': 50000,
39.         # 'impôt': 1384, 'surcôte': 0, 'décôte': 384, 'réduction': 347, 'taux': 0.14}
40.         taxpayer = TaxPayer().fromdict({'marié': 'oui', 'enfants': 2, 'salaire': 50000})
41.         self.métier.calculate_tax(taxpayer, self.admindata)
42.         # vérifications
43.         self.assertAlmostEqual(taxpayer.impôt, 1384, delta=1)
44.         self.assertAlmostEqual(taxpayer.décôte, 384, delta=1)
45.         self.assertAlmostEqual(taxpayer.réduction, 347, delta=1)
46.         self.assertAlmostEqual(taxpayer.taux, 0.14, delta=0.01)
47.         self.assertEqual(taxpayer.surcôte, 0)
48.
49.     def test_3(self) -> None:
50.         from TaxPayer import TaxPayer
51.
52.         # {'marié': 'oui', 'enfants': 3, 'salaire': 50000,
53.         # 'impôt': 0, 'surcôte': 0, 'décôte': 720, 'réduction': 0, 'taux': 0.14}
54.         taxpayer = TaxPayer().fromdict({'marié': 'oui', 'enfants': 3, 'salaire': 50000})
55.         self.métier.calculate_tax(taxpayer, self.admindata)
56.         # vérifications
57.         self.assertEqual(taxpayer.impôt, 0)
58.         self.assertAlmostEqual(taxpayer.décôte, 720, delta=1)
59.         self.assertEqual(taxpayer.réduction, 0)
60.         self.assertAlmostEqual(taxpayer.taux, 0.14, delta=0.01)
61.         self.assertEqual(taxpayer.surcôte, 0)
62.
63.     def test_4(self) -> None:
64.         from TaxPayer import TaxPayer
65.
66.         # {'marié': 'non', 'enfants': 2, 'salaire': 100000,
67.         # 'impôt': 19884, 'surcôte': 4480, 'décôte': 0, 'réduction': 0, 'taux': 0.41}
68.         taxpayer = TaxPayer().fromdict({'marié': 'non', 'enfants': 2, 'salaire': 100000})
69.         self.métier.calculate_tax(taxpayer, self.admindata)
70.         # vérifications
71.         self.assertAlmostEqual(taxpayer.impôt, 19884, delta=1)
72.         self.assertEqual(taxpayer.décôte, 0)
73.         self.assertEqual(taxpayer.réduction, 0)
74.         self.assertAlmostEqual(taxpayer.taux, 0.41, delta=0.01)
75.         self.assertAlmostEqual(taxpayer.surcôte, 4480, delta=1)
76.
77.     def test_5(self) -> None:
78.         from TaxPayer import TaxPayer
79.
80.         # {'marié': 'non', 'enfants': 3, 'salaire': 100000,
81.         # 'impôt': 16782, 'surcôte': 7176, 'décôte': 0, 'réduction': 0, 'taux': 0.41}
82.         taxpayer = TaxPayer().fromdict({'marié': 'non', 'enfants': 3, 'salaire': 100000})
83.         self.métier.calculate_tax(taxpayer, self.admindata)
84.         # vérifications
85.         self.assertAlmostEqual(taxpayer.impôt, 16782, delta=1)
86.         self.assertEqual(taxpayer.décôte, 0)
87.         self.assertEqual(taxpayer.réduction, 0)
88.         self.assertAlmostEqual(taxpayer.taux, 0.41, delta=0.01)
89.         self.assertAlmostEqual(taxpayer.surcôte, 7176, delta=1)
90.
91.     def test_6(self) -> None:
92.         from TaxPayer import TaxPayer
93.
94.         # {'marié': 'oui', 'enfants': 3, 'salaire': 100000,
95.         # 'impôt': 9200, 'surcôte': 2180, 'décôte': 0, 'réduction': 0, 'taux': 0.3}
96.         taxpayer = TaxPayer().fromdict({'marié': 'oui', 'enfants': 3, 'salaire': 100000})
97.         self.métier.calculate_tax(taxpayer, self.admindata)
98.         # vérifications
99.         self.assertAlmostEqual(taxpayer.impôt, 9200, delta=1)
100.        self.assertEqual(taxpayer.décôte, 0)
101.        self.assertEqual(taxpayer.réduction, 0)
102.        self.assertAlmostEqual(taxpayer.taux, 0.3, delta=0.01)
103.        self.assertAlmostEqual(taxpayer.surcôte, 2180, delta=1)
104.
105.    def test_7(self) -> None:
106.        from TaxPayer import TaxPayer
107.
108.        # {'marié': 'oui', 'enfants': 5, 'salaire': 100000,
109.        # 'impôt': 4230, 'surcôte': 0, 'décôte': 0, 'réduction': 0, 'taux': 0.14}
110.        taxpayer = TaxPayer().fromdict({'marié': 'oui', 'enfants': 5, 'salaire': 100000})
111.        self.métier.calculate_tax(taxpayer, self.admindata)
```

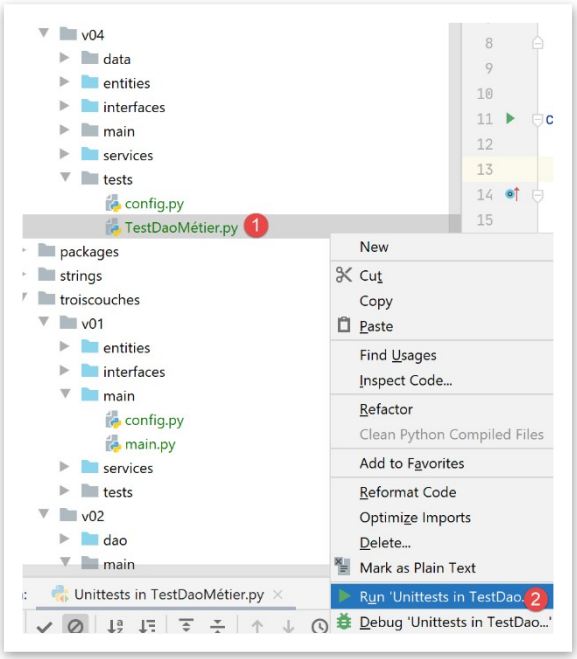
```
112.         # vérifications
113.         self.assertAlmostEqual(taxpayer.impôt, 4230, delta=1)
114.         self.assertEqual(taxpayer.décôte, 0)
115.         self.assertEqual(taxpayer.réduction, 0)
116.         self.assertAlmostEqual(taxpayer.taux, 0.14, delta=0.01)
117.         self.assertEqual(taxpayer.surcôte, 0)
118.
119.     def test_8(self) -> None:
120.         from TaxPayer import TaxPayer
121.
122.         # {'marié': 'non', 'enfants': 0, 'salaire': 100000,
123.         # 'impôt': 22986, 'surcôte': 0, 'décôte': 0, 'réduction': 0, 'taux': 0.41}
124.         taxpayer = TaxPayer().fromdict({'marié': 'non', 'enfants': 0, 'salaire': 100000})
125.         self.métier.calculate_tax(taxpayer, self.admindata)
126.         # vérifications
127.         self.assertAlmostEqual(taxpayer.impôt, 22986, delta=1)
128.         self.assertEqual(taxpayer.décôte, 0)
129.         self.assertEqual(taxpayer.réduction, 0)
130.         self.assertAlmostEqual(taxpayer.taux, 0.41, delta=0.01)
131.         self.assertEqual(taxpayer.surcôte, 0)
132.
133.     def test_9(self) -> None:
134.         from TaxPayer import TaxPayer
135.
136.         # {'marié': 'oui', 'enfants': 2, 'salaire': 30000,
137.         # 'impôt': 0, 'surcôte': 0, 'décôte': 0, 'réduction': 0, 'taux': 0}
138.         taxpayer = TaxPayer().fromdict({'marié': 'oui', 'enfants': 2, 'salaire': 30000})
139.         self.métier.calculate_tax(taxpayer, self.admindata)
140.         # vérifications
141.         self.assertEqual(taxpayer.impôt, 0)
142.         self.assertEqual(taxpayer.décôte, 0)
143.         self.assertEqual(taxpayer.réduction, 0)
144.         self.assertAlmostEqual(taxpayer.taux, 0.0, delta=0.01)
145.         self.assertEqual(taxpayer.surcôte, 0)
146.
147.     def test_10(self) -> None:
148.         from TaxPayer import TaxPayer
149.
150.         # {'marié': 'non', 'enfants': 0, 'salaire': 200000,
151.         # 'impôt': 64210, 'surcôte': 7498, 'décôte': 0, 'réduction': 0, 'taux': 0.45}
152.         taxpayer = TaxPayer().fromdict({'marié': 'non', 'enfants': 0, 'salaire': 200000})
153.         self.métier.calculate_tax(taxpayer, self.admindata)
154.         # vérifications
155.         self.assertAlmostEqual(taxpayer.impôt, 64210, 1)
156.         self.assertEqual(taxpayer.décôte, 0)
157.         self.assertEqual(taxpayer.réduction, 0)
158.         self.assertAlmostEqual(taxpayer.taux, 0.45, delta=0.01)
159.         self.assertAlmostEqual(taxpayer.surcôte, 7498, delta=1)
160.
161.     def test_11(self) -> None:
162.         from TaxPayer import TaxPayer
163.
164.         # {'marié': 'oui', 'enfants': 3, 'salaire': 200000,
165.         # 'impôt': 42842, 'surcôte': 17283, 'décôte': 0, 'réduction': 0, 'taux': 0.41}
166.         taxpayer = TaxPayer().fromdict({'marié': 'oui', 'enfants': 3, 'salaire': 200000})
167.         self.métier.calculate_tax(taxpayer, self.admindata)
168.         # vérifications
169.         self.assertAlmostEqual(taxpayer.impôt, 42842, 1)
170.         self.assertEqual(taxpayer.décôte, 0)
171.         self.assertEqual(taxpayer.réduction, 0)
172.         self.assertAlmostEqual(taxpayer.taux, 0.41, delta=0.01)
173.         self.assertAlmostEqual(taxpayer.surcôte, 17283, delta=1)
174.
175.
176. if __name__ == '__main__':
177.     unittest.main()
```

Commentaires

- ligne 11 : la classe de test étend la classe **[unittest.TestCase]** ;
- lignes 13-19 : dans un test UnitTest, la méthode **[setUp]** est exécutée avant chacune des méthodes **[test_]** ;
- ligne 16 : on récupère la configuration issue du script **[config]** étudié précédemment ;
- ligne 18 : on mémorise une référence sur la couche **[métier]** ;
- ligne 19 : on demande à la couche **[dao]** l'objet **[AdminData]** encapsulant les données de l'administration fiscale et on le mémorise ;

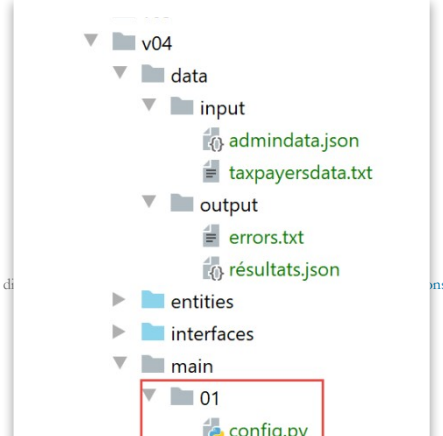
- lignes 21-173 : 11 tests dont les résultats ont été vérifiés sur le site officiel des impôts 2019 |https://www3.impots.gouv.fr/simulateur/calcul_impot/2019/simplifie/index.htm| ;
- lignes 21-33 : tous les tests ont été construits sur le même modèle ;
- ligne 22 : on importe la classe **[TaxPayer]** ;
- ligne 24 : contribuable testé ;
- ligne 25 : résultats attendus ;
- ligne 26 : construction de l'objet **[TaxPayer]** du contribuable ;
- ligne 27 : calcul de son impôt. Le résultat est dans **[taxpayer]** ;
- lignes 29-33 : vérification des résultats obtenus ;
- ligne 29 : on vérifie le montant de l'impôt à l'euro près. Les tests ont en effet montré que les résultats obtenus par l'algorithme de ce document pouvaient différer des chiffres officiels au plus d'un montant de 1 euro ;

L'exécution des tests donne les résultats suivants :




```
1.  Testing started at 16:08 ...
2.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe "C:\Program
    Files\JetBrains\PyCharm Community Edition 2020.1.2\plugins\python-
    ce\helpers\pycharm\_jb_unittest_runner.py" --path C:/Data/st-2020/dev/python/cours-2020/python3-
    flask-2020/impots/v04/tests/TestDaoMétier.py
3.  Launching unittests with arguments python -m unittest C:/Data/st-2020/dev/python/cours-2020/python3-
    flask-2020/impots/v04/tests/TestDaoMétier.py in C:\Data\st-2020\dev\python\cours-2020\python3-
    flask-2020\impots\v04\tests
4.
5.
6.
7.  Ran 11 tests in 0.055s
8.
9.  OK
10.
11. Process finished with exit code 0
```

15.1.5 Script principal



Le script principal est configuré par le script **[config]** suivant :

```
def configure():
    import os

    # étape 1 ------
    # configuration du python Path

    # dossier de ce fichier
    script_dir = os.path.dirname(os.path.abspath(__file__))

    # root_dir
    root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes"

    # dépendances de l'application
    absolute_dependencies = [
        # dépendances locales
        f"{script_dir}/../../entities",
        f"{script_dir}/../../interfaces",
        f"{script_dir}/../../services",
        f"{root_dir}/02/entities",
    ]

    # on fixe le syspath
    from myutils import set_syspath
    set_syspath(absolute_dependencies)

    # étape 2 ------
    # configuration de l'application
    config = {
        # chemins absolus des fichiers de l'application
        "taxpayersFilename": f"{script_dir}/../../data/input/taxpayersdata.txt",
        "resultsFilename": f"{script_dir}/../../data/output/résultats.json",
        "admindataFilename": f"{script_dir}/../../data/input/admindata.json",
        "errorsFilename": f"{script_dir}/../../data/output/errors.txt"
    }

    # instanciation des couches de l'application
    from ImpôtsDaoWithAdminDataInJsonFile import ImpôtsDaoWithAdminDataInJsonFile
    from ImpôtsMétier import ImpôtsMétier

    dao = ImpôtsDaoWithAdminDataInJsonFile(config)
    métier = ImpôtsMétier()

    # on met les instances de couches dans la config
    config["dao"] = dao
    config["métier"] = métier

    # on rend la config
    return config
```

Il est analogue à celui utilisé pour le test des couches **[métier]** et **[dao]**.

Le script principal **[main.py]** est le suivant :

```
# on configure l'application
import config

config = config.configure()

# imports
from ImpôtsError import ImpôtsError

# on récupère les couches de l'application (elles sont déjà instanciées)
dao = config["dao"]
métier = config["métier"]

try:
    # récupération des tranches de l'impôt
    admindata = dao.get_admindata()
    # lecture des données des contribuables
    taxpayers = dao.get_taxpayers_data()["taxpayers"]
    # des contribuables ?
    if not taxpayers:
        raise ImpôtsError(51, f"Pas de contribuables valides dans le fichier {config['taxpayersFilename']}")
    # calcul de l'impôt des contribuables
    for taxpayer in taxpayers:
```

```
23.         # taxpayer est à la fois un paramètre d'entrée et de sortie
24.         # taxpayer va être modifié
25.         métier.calculate_tax(taxpayer, admindata)
26.     # écriture des résultats dans un fichier texte
27.     dao.write_taxpayers_results(taxpayers)
28. except ImpôtsError as erreur:
29.     # affichage de l'erreur
30.     print(f"L'erreur suivante s'est produite : {erreur}")
31. finally:
32.     # terminé
33.     print("Travail terminé...")
```

Notes

- lignes 2-4 : on récupère la configuration de l'application. On sait également que le Python Path de l'application a été construit ;
- lignes 9-11 : on récupère des références sur les couches **[métier]** et **[dao]** ;
- ligne 15 : on obtient les données de l'administration fiscale ;
- ligne 17 : on obtient la liste des contribuables dont il faut calculer l'impôt ;
- lignes 19-20 : si cette liste est vide, on lève une exception ;
- lignes 22-25 : calcul de l'impôt des différents objets **[taxpayer]** grâce à la couche **[métier]** ;
- ligne 27 : **[taxpayers]** est désormais une liste d'objets **[TaxPayer]** où les attributs (impôt, décôte, surcôte, réduction, taux) ont reçu une valeur. Cette liste est écrite dans un fichier jSON ;
- lignes 28-30 : interception d'une éventuelle erreur ;
- lignes 31-33 : exécutées dans tous les cas ;

L'exécution du script donne les mêmes résultats que dans les versions précédentes. Le fichier des erreurs de contribuables était une nouveauté dans cette version. Après exécution du script **[main]** son contenu est le suivant :

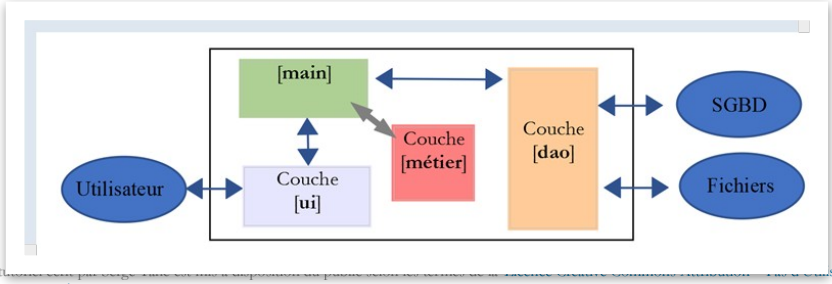
```
1. Analyse du fichier C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\impots\v04\main\01/../../data/input/taxpayersdata.txt
2.
3. Ligne 17, not enough values to unpack (expected 4, got 2)
4. Ligne 19, too many values to unpack (expected 4)
5. Ligne 21, MyException[1, L'identifiant d'une entité <class 'TaxPayer.TaxPayer'> doit être un entier >=0]
```

Les lignes erronées étaient les suivantes :

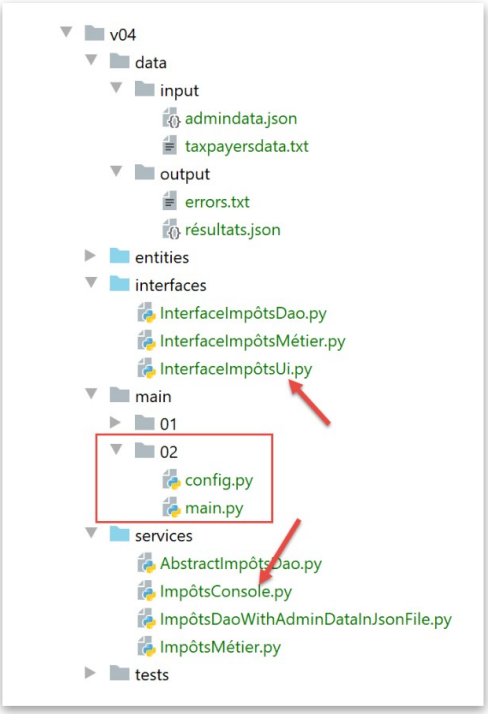
```
1.  # données valides : id, marié, enfants, salaire
2.  1,oui,2,55555
3.  2,oui,2,50000
4.  3,oui,3,50000
5.  4,non,2,100000
6.  5,non,3,100000
7.  6,oui,3,100000
8.  7,oui,5,100000
9.  8,non,0,100000
10. 9,oui,2,30000
11. 10,non,0,200000
12. 11,oui,3,200000
13. # on peut avoir des lignes vides
14.
15. # on crée des lignes erronées
16. # pas assez de valeurs
17. 11,12
18. # trop de valeurs
19. 12,oui,3,200000, x, y
20. # des valeurs erronées
21. x,x,x,x
```

15.2 Version 4 – application 2

Dans cette version, c'est l'utilisateur au clavier qui donne la liste des contribuables. L'architecture de l'application sera la suivante :



Un nouveau module apparaît : la couche **[ui]** (User Interface) qui va dialoguer avec l'utilisateur. Cette couche aura une interface et sera implémentée par une classe.



15.2.1 L'interface `[InterfaceImpôtsUi]`

```
1.  # imports
2.  from abc import ABC, abstractmethod
3.
4.
5.  # interface InterfaceImpôtsUI
6.  class InterfaceImpôtsUi(ABC):
7.      # exécution de la classe implémentant l'interface
8.      @abstractmethod
9.      def run(self):
10.         pass
```

L'interface **[InterfaceImpôtsUi]** n'aura qu'une méthode, celle des lignes 8-10. L'interface sera ici implémentée avec une application console mais on pourrait aussi l'implémenter avec une interface graphique. Les paramètres passés à la méthode **[run]** ne seraient pas les mêmes dans les deux implémentations. Pour contourner ce problème, la méthode habituelle est de :

- ne pas mettre de paramètres à la méthode **[run]** (ou le minimum de paramètres) ;
- passer des paramètres au constructeur de la classe implémentant l'interface. Ils peuvent être différents d'une implémentation à l'autre. Ces paramètres sont enregistrés comme attributs de la classe ;
- faire en sorte que la méthode **[run]** utilise ces attributs de classe (self.x) ;

Cette méthode permet d'avoir une interface très générale qui est précisée par les paramètres des constructeurs de chaque classe d'implémentation. Cette méthode a déjà été utilisée pour la version modulaire n° 1.

15.2.2 La classe `[ImpôtsConsole]`

La classe **[ImpôtsConsole]** implémente l'interface **[InterfaceImpôtsUi]** de la façon suivante :

```
1.  # imports
2.  import re
3.
4.  from InterfaceImpôtsUi import InterfaceImpôtsUi
5.  from TaxPayer import TaxPayer
6.
7.
8.  # couche [UI]
```

```
9.  class ImpôtsConsole(InterfaceImpôtsUi):
10.     # constructeur
11.     def __init__(self, config: dict):
12.         # on mémorise les paramètres
13.         self.admindata = config['dao'].get_admindata()
14.         self.métier = config['métier']
15.
16.     def run(self):
17.         # dialogue interactif avec l'utilisateur
18.         fini = False
19.         while not fini:
20.             # le contribuable est-il marié ?
21.             marié = input("Le contribuable est-il marié / pacsé (oui/non) (* pour arrêter) : ").strip().lower()
22.             # vérification de la validité de la saisie
23.             while marié != "oui" and marié != "non" and marié != "*":
24.                 # msg d'erreur
25.                 print("Tapez oui ou non ou *")
26.                 # question de nouveau
27.                 marié = input("Le contribuable est-il marié / pacsé (oui/non) (* pour arrêter) : ").strip().lower()
28.             # fini ?
29.             if marié == "*":
30.                 # dialogue terminé
31.                 return
32.             # nombre d'enfants
33.             enfants = input("Nombre d'enfants : ").strip()
34.             # vérification de la validité de la saisie
35.             if not re.match(r"^\d+$", enfants):
36.                 # msg d'erreur
37.                 print("Tapez un nombre entier positif ou nul")
38.                 # on recommence
39.                 enfants = input("Nombre d'enfants : ").strip()
40.             # salaire annuel
41.             salaire = input("Salaire annuel : ").strip()
42.             # vérification de la validité de la saisie
43.             if not re.match(r"^\d+$", salaire):
44.                 # msg d'erreur
45.                 print("Tapez un nombre entier positif ou nul")
46.                 # on recommence
47.                 salaire = input("Salaire annuel : ").strip()
48.             # calcul de l'impôt
49.             taxpayer = TaxPayer().fromdict({'id': 0, 'marié': marié, 'enfants': int(enfants), 'salaire': int(salaire)})
50.             self.métier.calculate_tax(taxpayer, self.admindata)
51.             # affichage
52.             print(f"Impôt du contribuable = {taxpayer}\n\n")
53.             # contribuable suivant
```

- ligne 9 : la classe **[ImpôtsConsole]** implémente l'interface **[InterfaceImpôtsUi]** ;
- ligne 11 : le constructeur de la classe reçoit un paramètre, le dictionnaire **[config]** de la configuration de l'application ;
  - ligne 13 : on récupère les données de l'administration fiscale permettant le calcul de l'impôt ;
  - ligne 14 : on mémorise une référence sur la couche **[métier]** ;
- ligne 16 : implémentation de la méthode **[run]** de l'interface ;
- lignes 19-53 : dialogue avec l'utilisateur. Il consiste
  - à demander les trois informations (marié, enfants, salaire) du contribuable ;
  - à calculer son impôt ;
  - à afficher celui-ci ;
  - le dialogue se termine lorsque l'utilisateur répond * à la première question ;
- lignes 20-27 : on demande si le contribuable est marié et on vérifie la validité de la réponse ;
- lignes 29-31 : si l'utilisateur a répondu '*' à la question le dialogue est arrêté ;
- lignes 32-39 : on demande le nombre d'enfants du contribuable et on vérifie la validité de la réponse ;
- lignes 40-47 : on demande le salaire annuel du contribuable et on vérifie la validité de la réponse ;
- lignes 48-50 : avec ces informations on fait calculer, par la couche **[métier]**, l'impôt du contribuable ;
- ligne 52 : le montant de l'impôt est affiché ;

15.2.3 Le script principal

Le script principal **[main]** est configuré par le fichier **[config]** suivant :

```
1.  def configure():
2.      import os
3.
4.      # étape 1 ------
5.      # configuration du python Path
6.
7.      # dossier de ce fichier
8.      script_dir = os.path.dirname(os.path.abspath(__file__))
9.
10.     # root_dir
11.     root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/classes"
```

```
12.
13.     # dépendances de l'application
14.     absolute_dependencies = [
15.         # dépendances locales
16.         f"{script_dir}/../../entities",
17.         f"{script_dir}/../../interfaces",
18.         f"{script_dir}/../../services",
19.         f"{root_dir}/02/entities",
20.     ]
21.
22.     # on fixe le syspath
23.     from myutils import set_syspath
24.     set_syspath(absolute_dependencies)
25.
26.     # étape 2 ------
27.     # configuration de l'application
28.     config = {
29.         # chemins absolus des fichiers de l'application
30.         "admindataFilename": f"{script_dir}/../../data/input/admindata.json",
31.     }
32.
33.     # instanciation des couches de l'application
34.     from ImpôtsDaoWithAdminDataInJsonFile import ImpôtsDaoWithAdminDataInJsonFile
35.     from ImpôtsMétier import ImpôtsMétier
36.     from ImpôtsConsole import ImpôtsConsole
37.
38.     # couche dao
39.     dao = ImpôtsDaoWithAdminDataInJsonFile(config)
40.     # couche métier
41.     métier = ImpôtsMétier()
42.     # on met les instances de couches dans la config
43.     config["dao"] = dao
44.     config["métier"] = métier
45.     # couche ui
46.     ui = ImpôtsConsole(config)
47.     config["ui"] = ui
48.
49.     # on rend la config
50.     return config
```

Le script chef d'orchestre est le suivant (main.py) :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # imports
7.  from ImpôtsError import ImpôtsError
8.
9.  # on récupère les couches de l'application (elles sont déjà instanciées)
10. ui = config["ui"]
11.
12. # code
13. try:
14.     # exécution de la couche [ui]
15.     ui.run()
16. except ImpôtsError as erreur:
17.     # on affiche le message d'erreur
18.     print(f"L'erreur suivante s'est produite : {erreur}")
19. finally:
20.     # exécuté dans tous les cas
21.     print("Travail terminé...")
```

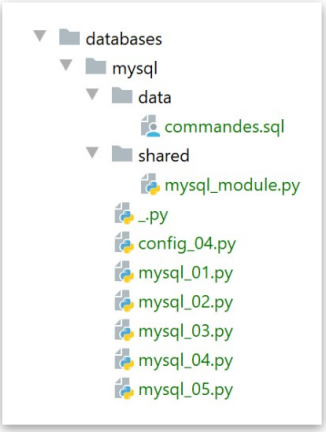
- lignes 1-4 : on récupère la configuration de l'application ;
- ligne 10 : on récupère une référence sur la couche **[ui]** ;
- lignes 12-21 : la structure du code est la même que dans l'application précédente : du code entouré d'un try / catch pour arrêter toute éventuelle exception ;
- ligne 15 : on demande à la couche **[ui]** de s'exécuter : le dialogue avec l'utilisateur commence alors ;
- lignes 16-18 : interception d'une éventuelle exception ;

Voici un exemple d'exécution :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/v04/main/02/main.py
```

```
2.  Le contribuable est-il marié / pacsé (oui/non) (* pour arrêter) : oui
3.  Nombre d'enfants : 3
4.  Salaire annuel : 200000
5.  Impôt du contribuable = {"id": 0, "marié": "oui", "enfants": 3, "salaire": 200000, "impôt": 42842,
    "surcôte": 17283, "taux": 0.41, "décôte": 0, "réduction": 0}
6.
7.
8.  Le contribuable est-il marié / pacsé (oui/non) (* pour arrêter) : *
9.  Travail terminé...
10.
11. Process finished with exit code 0
```

16 Utilisation du SGBD MySQL



16.1 Intallation du SGBD MySQL

Pour disposer du SGBD MySQL, nous allons installer le logiciel Laragon.

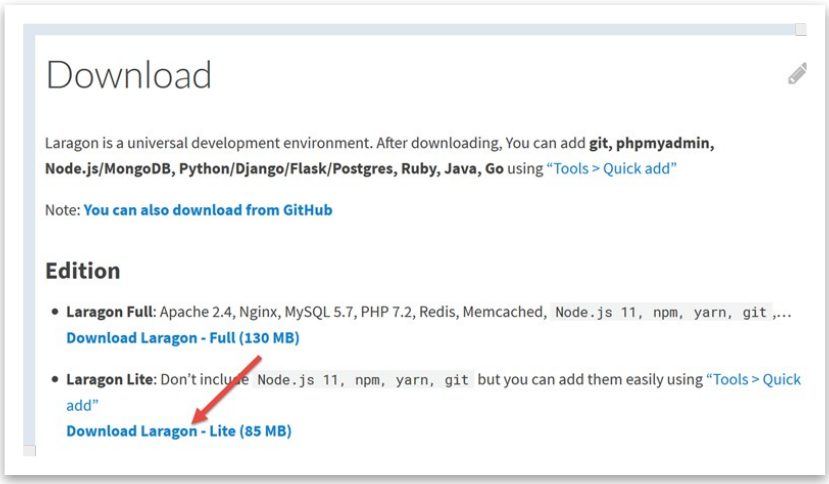
16.1.1 Installation de Laragon

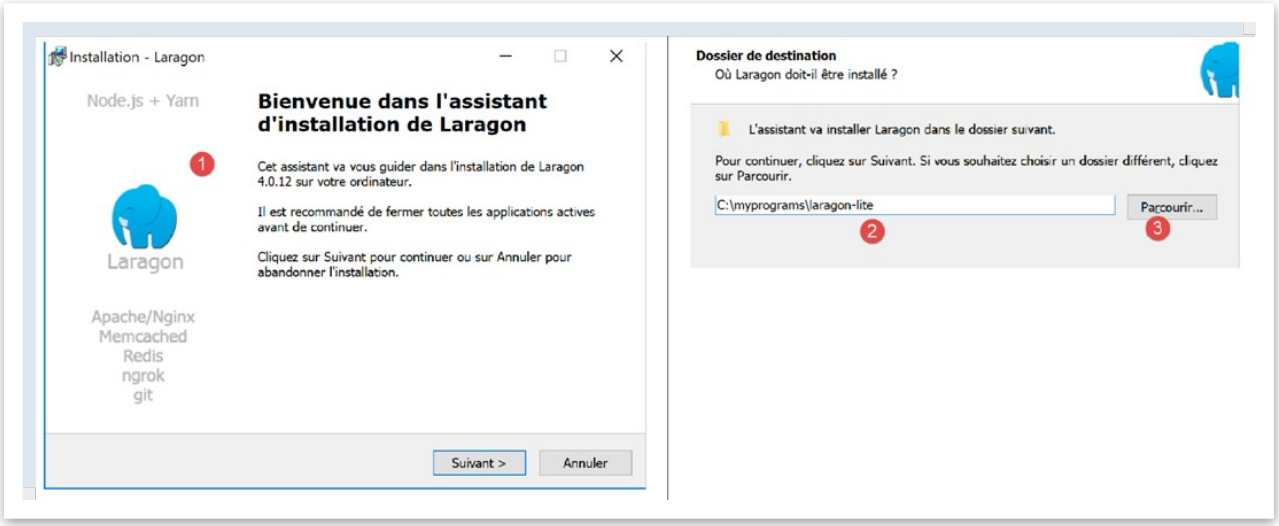
Laragon est un package réunissant plusieurs logiciels :

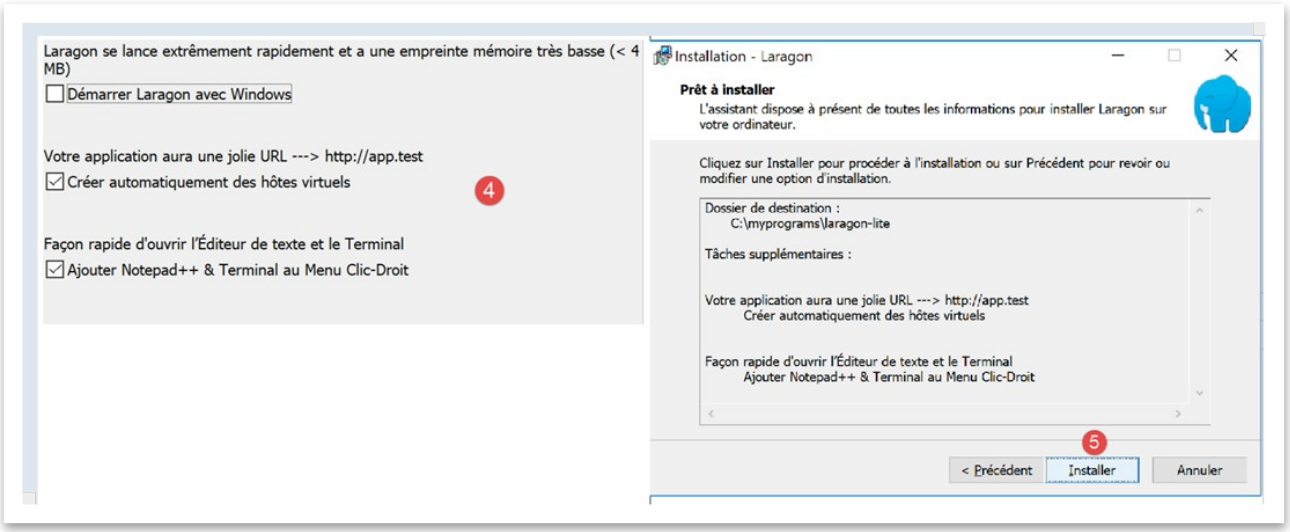
- un serveur web Apache. Nous l'utiliserons pour l'écriture de scripts web en Python ;
- le SGBD MySQL ;
- le langage de script PHP que nous n'utiliserons pas ;
- un serveur Redis implémentant un cache pour des applications web. Nous n'utiliserons ;

Laragon peut être téléchargé (février 2020) à l'adresse suivante :

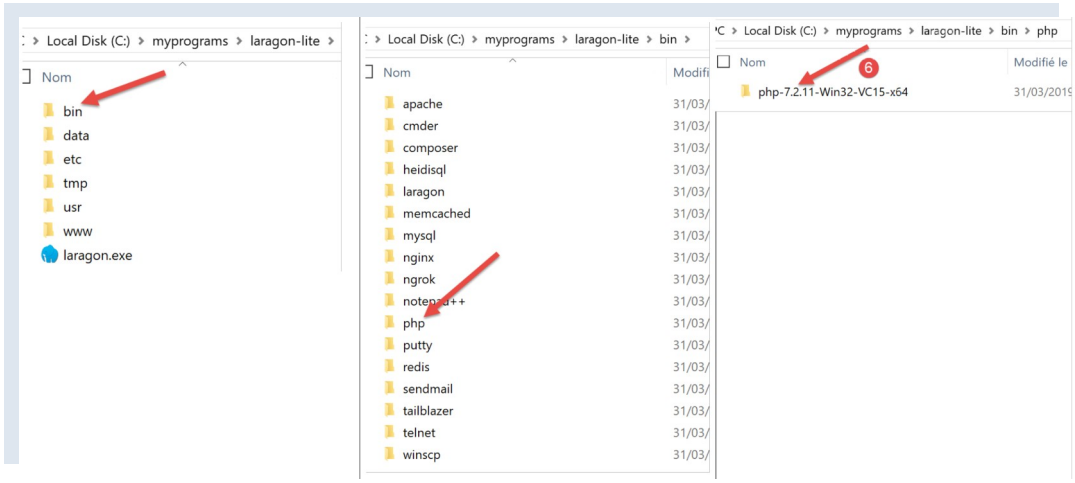
```
https://laragon.org/download/
```





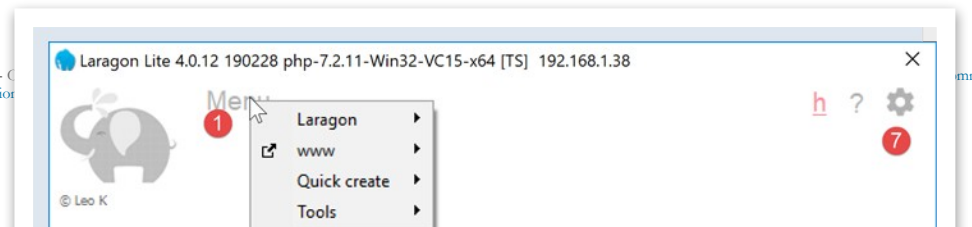


- l'installation **[1-5]** donne naissance à l'arborescence suivante :



- en **[6]** le dossier d'installation de PHP (pas utilisé dans ce document) ;

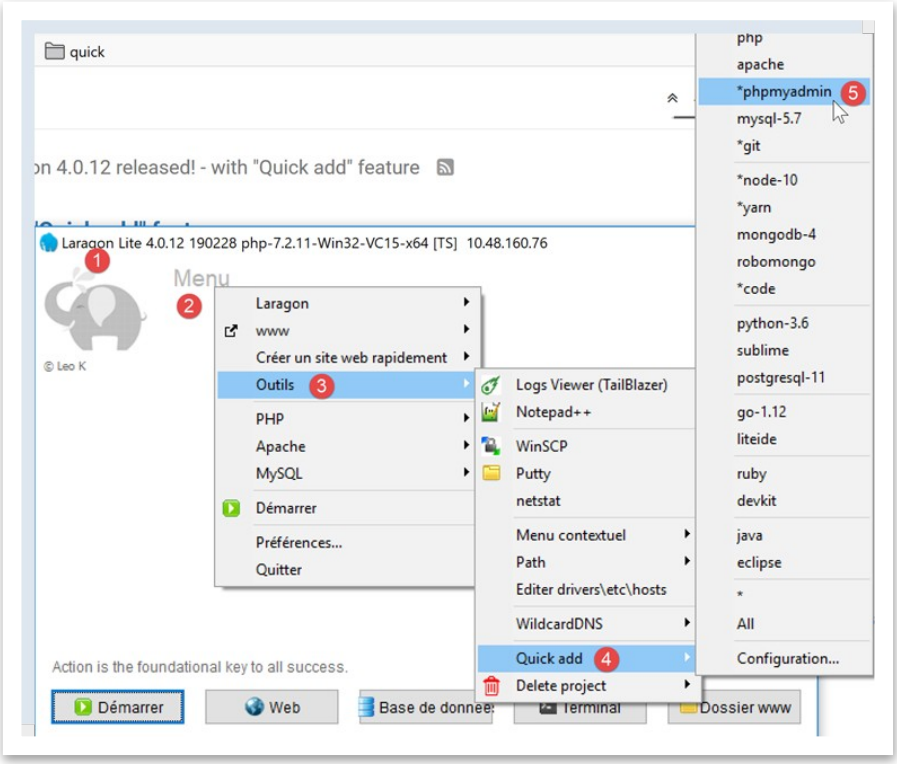
Le lancement de **[Laragon]** affiche la fenêtre suivante :



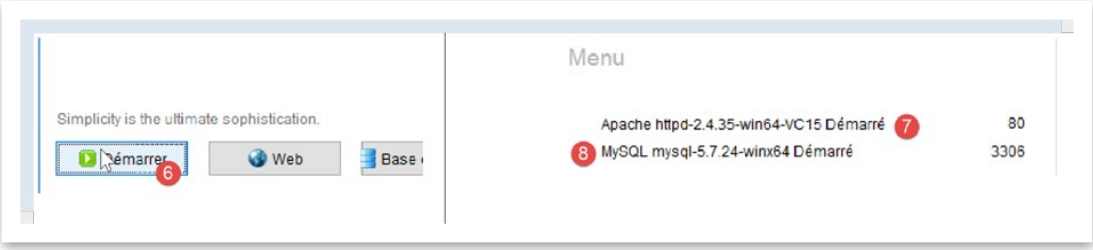
- **[1]** : le menu principal de Laragon ;
- **[2]** : le bouton **[Start All]** lance le serveur web Apache et le SGBD MySQL ;
- **[3]** : le bouton **[WEB]** affiche la page web **[http://localhost]** ;
- **[4]** : le bouton **[Database]** permet de gérer le SGBD MySQL avec l'outil **[phpMyAdmin]**. Il faut auparavant installer celui-ci ;
- **[5]** : le bouton **[Terminal]** ouvre un terminal de commandes ;
- **[6]** : le bouton **[Root]** ouvre un explorateur Windows positionné sur le dossier **[<laragon>/www]** qui est la racine du site web **[http://localhost]**. C'est là qu'il faut placer les applications web statiques gérées par le serveur Apache de Laragon ;

16.1.2 Création d'une base de données

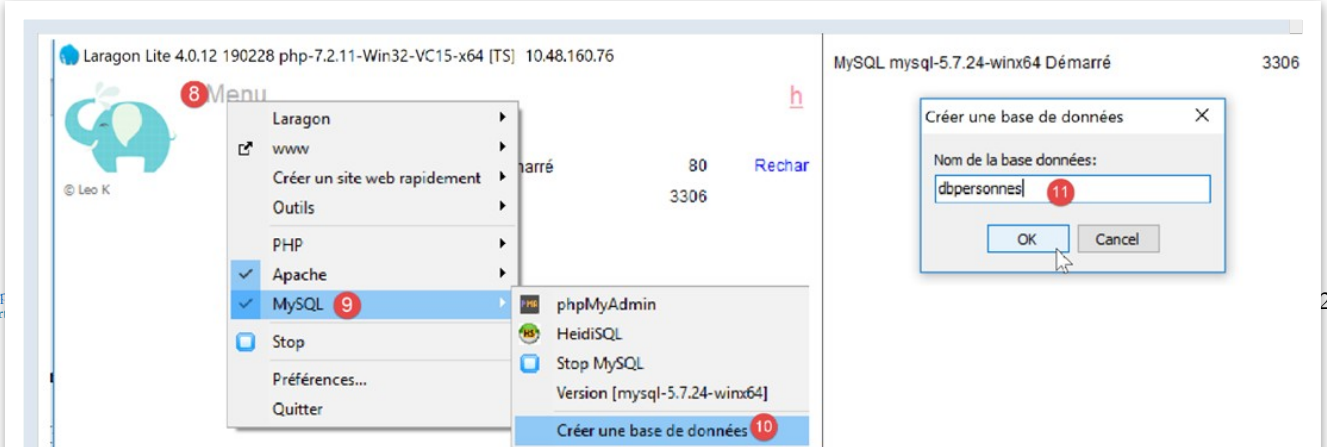
Nous montrons maintenant comment créer une base de données ainsi qu'un utilisateur MySQL avec l'outil Laragon.



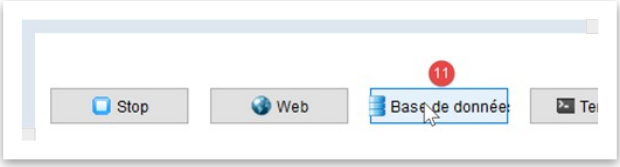
- une fois lancé, Laragon **[1]** peut être administré à partir d'un menu **[2]** ;
- en **[3-5]**, on installe l'outil **[phpMyAdmin]** d'administration de MySQL s'il n'a pas déjà été installé ;



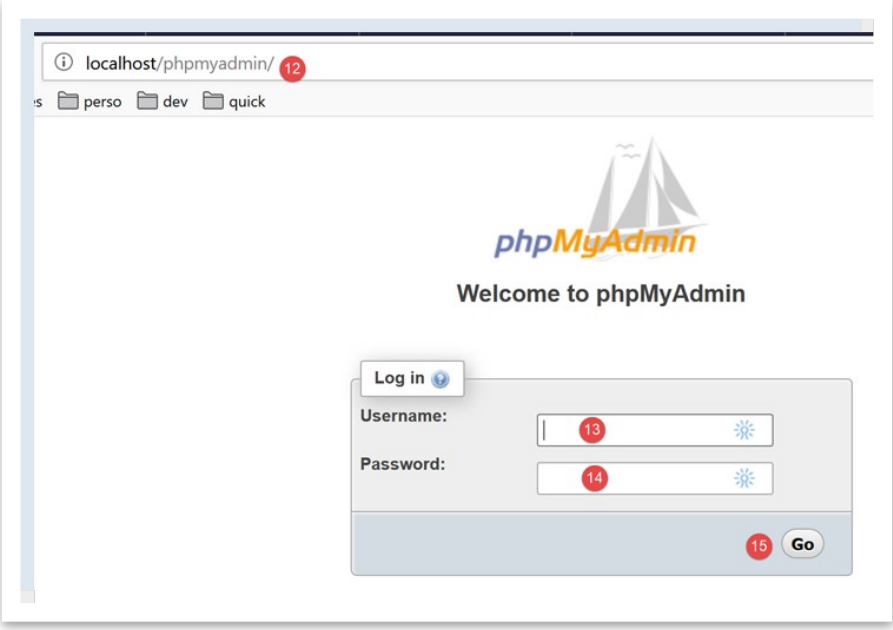
- en **[6]**, on démarre le serveur web Apache ainsi que le SGBD MySQL ;
- en **[7]**, le serveur Apache est lancé ;
- en **[8]**, le SGBD MySQL est lancé ;



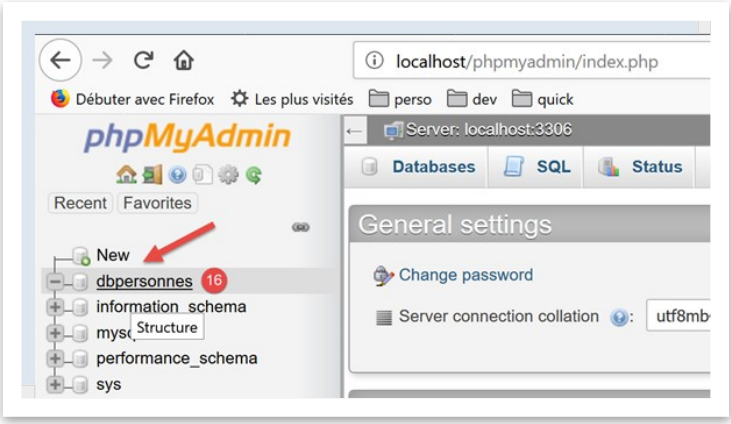
- en **[8-10]**, on crée une base de données qu'on nomme **[dbpersonnes] [11]**. On va construire une base de données de personnes ;



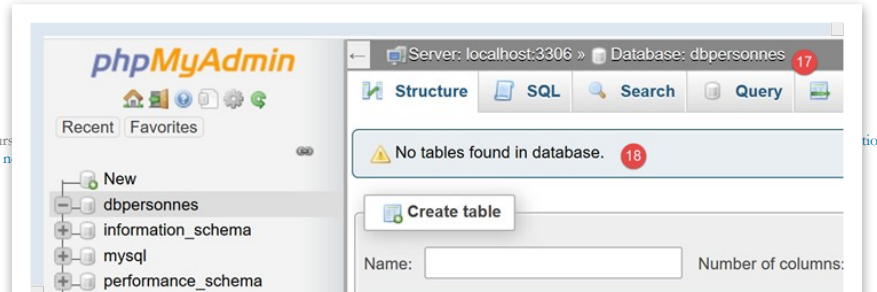
- en **[11]**, on va gérer la base de données qu'on vient de créer ;



- l'opération **[Bases de données]** émet une requête web vers l'URL **[http://localhost/phpmyadmin] [12]**. C'est le serveur web Apache de Laragon qui répond. L'URL **[http://localhost/phpmyadmin]** est l'URL de l'utilitaire **[phpMyAdmin]** que nous avons installé précédemment **[5]**. Cet utilitaire permet de gérer les bases de données MySQL ;
- par défaut, les identifiants de connexion de l'administrateur de la base sont : **root [13]** sans mot de passe **[14]** ;

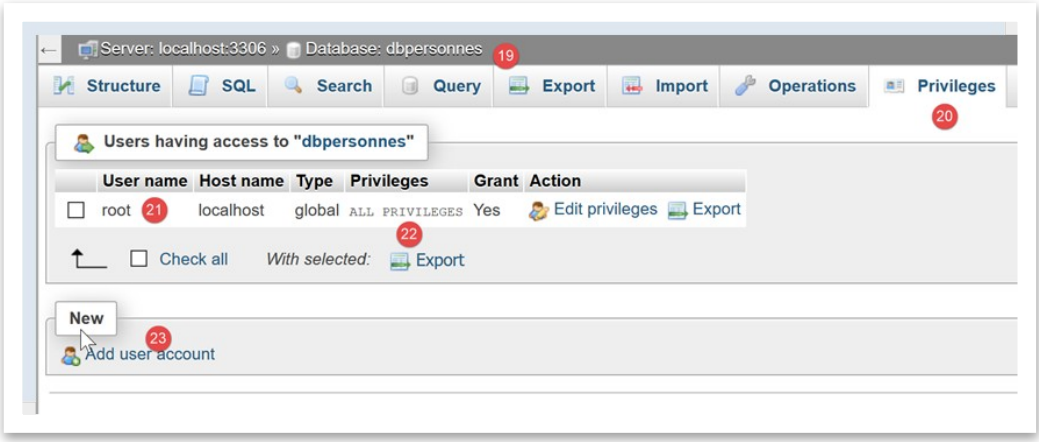


- en **[16]**, la base de données que nous avons créée précédemment ;

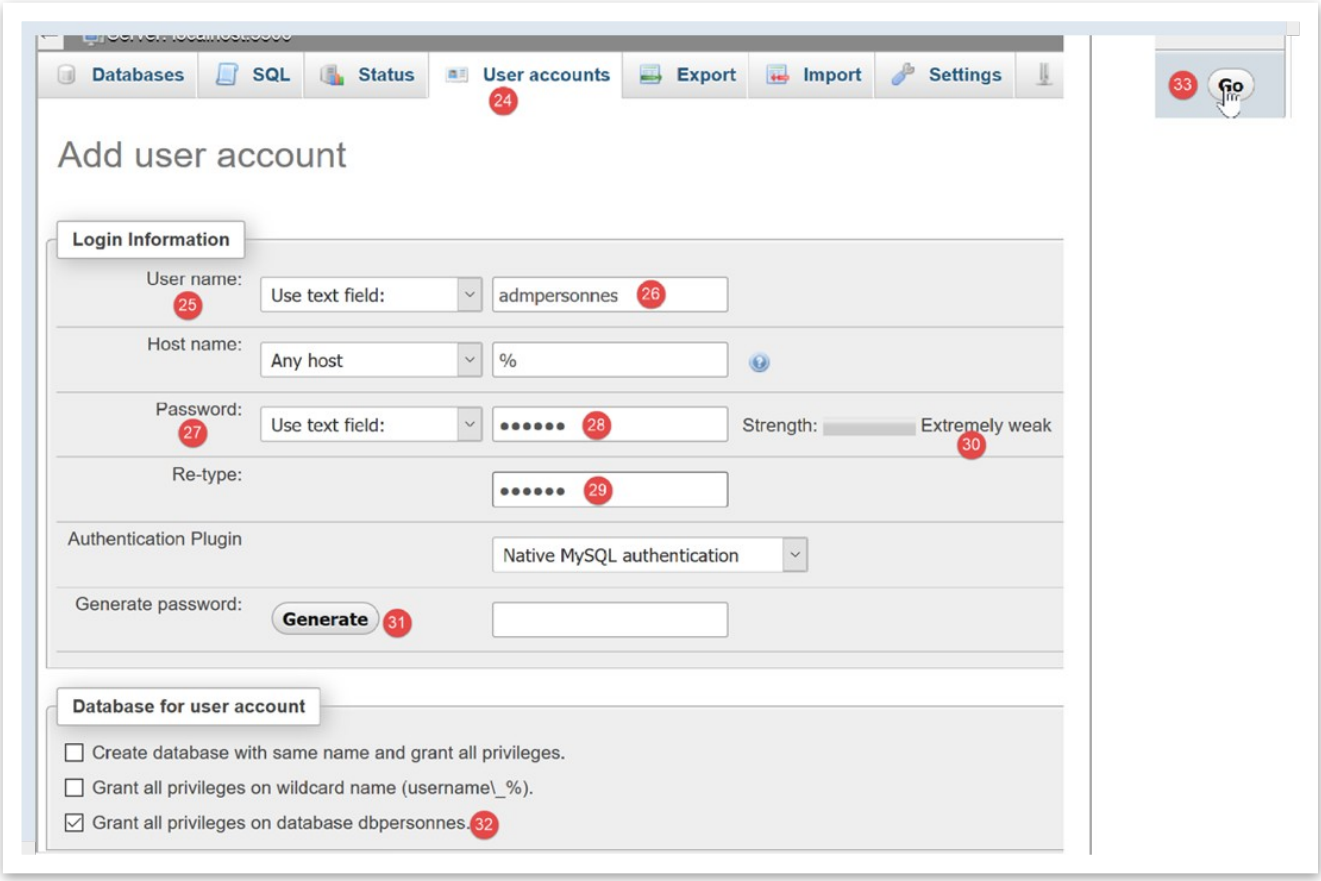


- on a pour l'instant une base **[dbpersonnes]** **[17]** qui est vide **[18]** ;

On crée un utilisateur **[admpersonnes]** avec le mot de passe **[nobody]** qui va avoir tous les droits sur la base de données **[dbpersonnes]** :



- en **[19]**, on est positionnés sur la base **[dbpersonnes]** ;
- en **[20]**, on sélectionne l'onglet **[Privileges]** ;
- en **[21-22]**, on voit que l'utilisateur **[root]** a tous les droits sur la base **[dbpersonnes]** ;
- en **[23]**, on crée un nouvel utilisateur ;



- en **[25-26]**, l'utilisateur aura l'identifiant **[admdbpersonnes]** ;
- en **[27-29]**, son mot de passe sera **[nobody]** ;
- en **[30]**, phpMyAdmin signale que le mot de passe est très faible (facile à craquer). En production, il est préférable de générer un mot de passe fort avec **[31]** ;

- en **[32]**, on indique que l'utilisateur **[admdbpersonnes]** doit avoir tous les droits sur la base **[dbpersonnes]** ;
- en **[33]**, on valide les renseignements donnés ;



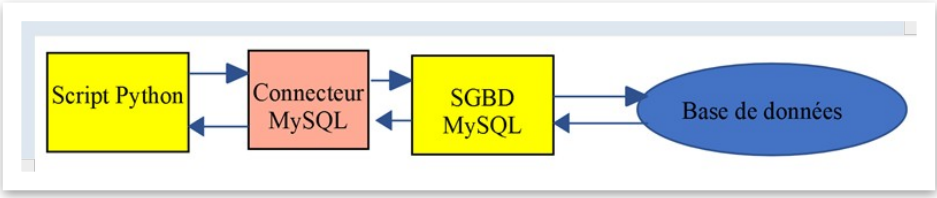
- en **[35]**, phpMyAdmin indique que l'utilisateur a été créé ;
- en **[36]**, l'ordre SQL qui a été émis sur la base ;
- en **[37]**, l'utilisateur **[admpersonnes]** a tous les droits sur la base de données **[dbpersonnes]** ;

Désormais nous avons :

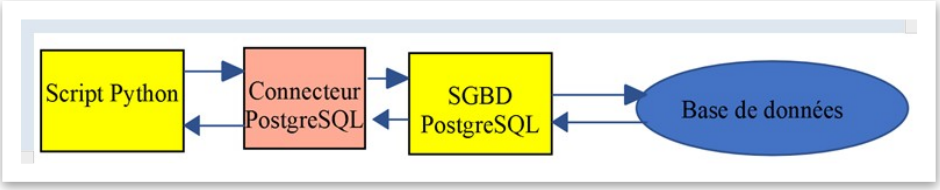
- une base de données MySQL **[dbpersonnes]** ;
- un utilisateur **[admpersonnes/nobody]** qui a tous les droits sur cette base de données ;

16.2 Intallation du package [mysql-connector-python]

Nous allons écrire des scripts Python pour exploiter la base de données créée précédemment avec l'architecture suivante :



Un connecteur sert à isoler le code Python du SGBD exploité. Il existe des connecteurs pour différents SGBD et ceux-ci respectent la même interface. Aussi lorsque ci-dessus, on remplace le SGBD MySQL par le SGBD PostgreSQL, l'architecture devient la suivante :



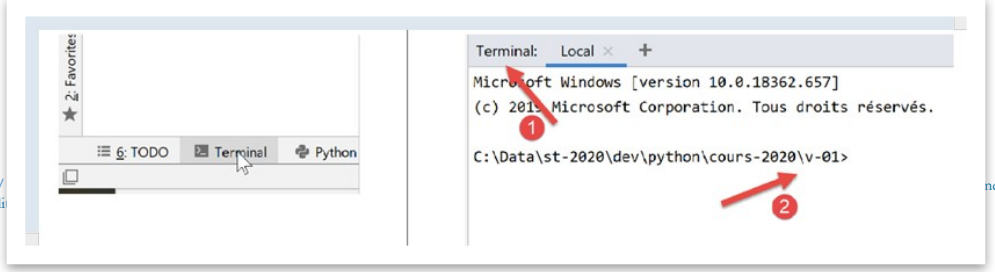
Parce que tous les connecteurs de SGBD respectent tous la même interface, le script Python **n'a normalement pas à être modifié**. Dans la réalité, la plupart des SGBD ont un SQL propriétaire :

- ils respectent la norme SQL (Structured Query Language) ;
- mais l'étendent, car elle n'est pas suffisante, avec des extensions du langage propriétaires ;

Aussi, est-il fréquent que lors d'un changement de SGBD il y ait des modifications de SQL à faire dans les scripts.

Nativement, Python n'offre pas la possibilité de gérer une base MySQL. Il faut pour cela télécharger un package. Il en existe plusieurs. Nous allons utiliser ici le package **[mysql-connector-python]** qui est le connecteur Officiel d'Oracle, l'entreprise qui possède MySQL.

L'installation du package se fera dans une fenêtre **[Terminal]** de Pycharm :



- le dossier en **[2]** n'a pas d'importance pour ce qui va suivre ;

Dans le terminal, on tape la commande **[pip search MySQL]** :

- **[pip]** (Package Installer for Python) est l'outil d'installation des packages Python. L'outil **[pip]** se connecte au dépôt contenant les packages Python ;
- **[search MySQL]** : demande la liste des packages contenant le terme **[MySQL]** (la casse n'a pas d'importance) dans leur nom ;

Les résultats de la commande sont les suivants :

```
1.  mysql (0.0.2)                                                - Virtual package for MySQL-python
2.  jx-mysql (3.49.20042)                                        - jx-mysql - JSON Expressions for MySQL
3.  weibo-mysql (0.1)                                            - insert mysql
4.  bits-mysql (1.0.3)                                           - BITS MySQL
5.  MySQL-python (1.2.5)                                         - Python interface to MySQL
6.  deployfish-mysql (0.2.13)                                    - Deployfish MySQL plugin
7.  mtstat-mysql (0.7.3.3)                                       - MySQL Plugins for mtstat
8.  bottle-mysql (0.3.1)                                         - MySQL integration for Bottle.
9.  WintxDriver-MySQL (2.0.0-1)                                  - MySQL support for Wintx
10. py-mysql (1.0)                                               - Operating Mysql for Python.
11. mysql-utilities (1.4.3)                                      - MySQL Utilities 1.4.3 (part of MySQL
    Workbench Distribution 6.0.0)
12. ….                                      - Tool to move slices of data from one MySQL store to another
13. mysql-tracer (2.0.2)                                         - A MySQL client to run queries, write
    execution reports and export results
14. mysql-utils (0.0.2)                                          - A simple MySQL library including a set of
    utility APIs for Python database programming
15. mysql-connector-repackaged (0.3.1)                           - MySQL driver written in Python
16. dffml-source-mysql (0.0.5)                                   - DFFML Source for MySQL Protocol
17. mysql-connector-python (8.0.19)                              - MySQL driver written in Python
18.   INSTALLED: 8.0.19 (latest)
19. prometheus-mysql-exporter (0.2.0)                            - MySQL query Prometheus exporter
20. backwork-backup-mysql (0.3.0)                                - Backwork plug-in for MySQL backups.
21. django-mysql-manager (0.1.4)                                 - django-mysql-manager is a Django based
    management interface for MySQL users and databases.
22. ….                                              - mysql operate
23.
24. C:\Data\st-2020\dev\python\cours-2020\v-01>
```

Ont été listés tous les modules dont le nom ou la description ont le mot clé *MySQL*. Celui que nous utiliserons (fév 2020) est **[mysql-connector-python]**, ligne 17. Pour l'installer, on tape dans le terminal la commande **[pip install -U mysql-connector-python]** :

```
1.  C:\Data\st-2020\dev\python\cours-2020\v-01>pip install -U mysql-connector-python
2.  Collecting mysql-connector-python
3.    Using cached mysql_connector_python-8.0.19-py2.py3-none-any.whl (355 kB)
4.  Requirement already satisfied, skipping upgrade: protobuf==3.6.1 in c:\myprograms\python38\lib\site-
    packages (from mysql-connector-python) (3.6.1)
5.  Requirement already satisfied, skipping upgrade: dnspython==1.16.0 in c:\myprograms\python38\lib\site-
    packages (from mysql-connector-python) (1.16.0)
6.  Requirement already satisfied, skipping upgrade: six>=1.9 in c:
    \users\serge\appdata\roaming\python\python38\site-packages (from protobuf==3.6.1->mysql-connector-python)
    (1.14.0)
7.  Requirement already satisfied, skipping upgrade: setuptools in c:\myprograms\python38\lib\site-packages
    (from protobuf==3.6.1->mysql-connector-python) (41.2.0)
8.  Installing collected packages: mysql-connector-python
9.  Successfully installed mysql-connector-python-8.0.19
```

- ligne 1 : l'option **[install -U]** (U=upgrade) demande la version la plus récente des différents packages associés au package **[mysql-connector-python]** ;

Pour connaître les packages installés dans l'environnement Python de notre macine, on tape la commande **[pip list]** :

```
1.  C:\Data\st-2020\dev\python\cours-2020\v-01>pip list
2.  Package               Version
3.  ---------------------- ----------
4.  asgiref               3.2.3
5.  astroid               2.3.3
6.  atomicwrites          1.3.0
7.  attrs                 19.3.0
8.  certifi               2019.11.28
9.  …
10. MarkupSafe            1.1.1
```

```
11. mccabe                0.6.1
12. more-itertools        8.1.0
13. mysql-connector-python 8.0.19
14. mysqlclient           1.4.6
15. packaging             20.0
16. pip                   20.0.1
17. pipenv                2018.11.26
18. …
```

- ligne 13 : on a bien le package **[mysql-connector-python]** ;

Pour savoir comment utiliser le package **[mysql-connector-python]** afin de gérer une base de données MySQL, on ira sur le site du package |https://dev.mysql.com/doc/connector-python/en/|. La suite présente une série d'exemples.

16.3 script [mysql_01] : connexion à une base MySQL - 1

Le script **[mysql_01]** présente la première étape de l'utilisation d'une base de données. Il va nous permettre de vérifier qu'on est capables de se connecter à la base **[dbpersonnes]** créée précédemment.

```
1.  # import du module mysql.connector
2.  from mysql.connector import connect, DatabaseError, InterfaceError
3.
4.  # connexion à une base MySql [dbpersonnes]
5.  # l'identité de l'utilisateur est (admpersonnes,nobody)
6.  USER = "admpersonnes"
7.  PWD = "nobody"
8.  HOST = "localhost"
9.  DATABASE = "dbpersonnes"
10.
11. # c'est parti
12. connexion = None
13. try:
14.     print("Connexion au SGBD MySQL en cours...")
15.     # connexion
16.     connexion = connect(host=HOST, user=USER, password=PWD, database=DATABASE)
17.     # suivi
18.     print(
19.         f"Connexion MySQL réussie à la base database={DATABASE}, host={HOST} sous l'identité user={USER},
    passwd={PWD}")
20. except (InterfaceError, DatabaseError) as erreur:
21.     # on affiche l'erreur
22.     print(f"L'erreur suivante s'est produite : {erreur}")
23. finally:
24.     # on ferme la connexion si elle a été ouverte
25.     if connexion:
26.         connexion.close()
```

Notes

- ligne 2 : on importe certaines fonctions et classes du module **[mysql.connector]** ;
- lignes 6-7 : les identifiants de l'utilisateur qui va se connecter ;
- ligne 8 : la machine qui héberge la base de données. En effet, le connecteur MySQL permet de travailler avec une base distante ;
- ligne 9 : le nom de la base de données à laquelle on veut se connecter ;
- lignes 11-26 : le script va connecter (ligne 16) l'utilisateur **[admpersonnes / nobody]** à la base de données **[dbpersonnes]** ;
- lignes 20-26 : la connexion peut échouer. Aussi la fait-on dans un try / except / finally ;
- ligne 16 : la méthode *connect* du module **[mysq.connector]** admet différents paramètres nommés :
  - *user* : utilisateur propriétaire de la connexion **[admpersonnes]** ;
  - *password* : mot de passe de l'utilisateur **[nobody]** ;
  - *host* : machine du SGBD MySQL **[localhost]** ;
  - *database* : la base de données à laquelle on se connecte. Optionnel.
- ligne 20 : si une exception est lancée, elle est de type **[DatabaseError]** ou **[InterfaceError]** ;
- lignes 23-26 : dans la clause **[finally]**, on ferme la connexion ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/mysql/mysql_01.py
2.  Connexion au SGBD MySQL en cours...
3.  Connexion MySQL réussie à la base database=dbpersonnes, host=localhost sous l'identité user=admpersonnes,
    passwd=nobody
```

```
4.
5.  Process finished with exit code 0
```

16.4 script [mysql_02] : connexion à une base MySQL - 2

Dans ce nouveau script, la connexion à la base est isolée dans une fonction :

```
1.  # import du module mysql.connector
2.  from mysql.connector import DatabaseError, InterfaceError, connect
3.
4.
5.  # ------------------------------------------------------------------------------
6.  def connexion(host: str, database: str, login: str, pwd: str):
7.      # connecte puis déconnecte (login,pwd) de la base [database] du serveur [host]
8.      # lance l'exception DatabaseError si problème
9.      connexion = None
10.     try:
11.         # connexion
12.         connexion = connect(host=host, user=login, password=pwd, database=database)
13.         print(
14.             f"Connexion réussie à la base database={database}, host={host} sous l'identité user={login}, passwd={pwd}")
15.     finally:
16.         # on ferme la connexion si elle a été ouverte
17.         if connexion:
18.             connexion.close()
19.             print("Déconnexion réussie\n")
20.
21.
22. # ---------------------------------------------- main
23. # identifiants de la connexion
24. USER = "admpersonnes"
25. PASSWD = "nobody"
26. HOST = "localhost"
27. DATABASE = "dbpersonnes"
28.
29. # connexion d'un utilisateur existant
30. try:
31.     connexion(host=HOST, login=USER, pwd=PASSWD, database=DATABASE)
32. except (InterfaceError, DatabaseError) as erreur:
33.     # on affiche l'erreur
34.     print(erreur)
35.
36. # connexion d'un utilisateur inexistant
37. try:
38.     connexion(host=HOST, login="xx", pwd="xx", database=DATABASE)
39. except (InterfaceError, DatabaseError) as erreur:
40.     # on affiche l'erreur
41.     print(erreur)
```

Notes :

- lignes 6-19 : une fonction **[connexion]** qui tente de connecter puis de déconnecter un utilisateur à la base de données **[dbpersonnes]**. Affiche le résultat ;
- lignes 29-41 : programme principal – appelle deux fois la méthode *connexion* et affiche les éventuelles exceptions ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/mysql/mysql_02.py
2.  Connexion MySQL réussie à la base database=dbpersonnes, host=localhost sous l'identité user=admpersonnes,
    passwd=nobody
3.  Déconnexion MySQL réussie
4.
5.  1045 (28000): Access denied for user 'xx'@'localhost' (using password: YES)
6.
7.  Process finished with exit code 0
```

16.5 script [mysql_03] : création d'une table MySQL

Maintenant qu'on sait créer une connexion avec un SGBD MySQL, on commence à émettre des ordres SQL sur cette connexion. Pour cela, nous allons nous connecter à la base créée **[dbpersonnes]** et utiliser la connexion pour créer une table dans la base.

```
1.  # imports
2.  import sys
3.
4.  from mysql.connector import DatabaseError, InterfaceError, connect
5.  from mysql.connector.connection import MySQLConnection
6.
```

```
7.
8.  # ------------------------------------------------------------------------------
9.  def execute_sql(connexion: MySQLConnection, update: str):
10.     # exécute une requête de mise à jour sur la connexion
11.     curseur = None
12.     try:
13.         # on demande un curseur
14.         curseur = connexion.cursor()
15.         # exécute la requête update sur la connexion
16.         curseur.execute(update)
17.     finally:
18.         # fermeture du curseur s'il a été obtenu
19.         if curseur:
20.             curseur.close()
21.
22.
23. # ------------------------------------------- main
24. # identifiants de la connexion
25. # l'identité de l'utilisateur
26. ID = "admpersonnes"
27. PWD = "nobody"
28. # la machine hôte du sgbd
29. HOST = "localhost"
30. # identité de la base
31. DATABASE = "dbpersonnes"
32.
33. # on y va étape par étape
34. try:
35.     # connexion
36.     connexion = connect(host=HOST, user=ID, password=PWD, database=DATABASE)
37.     # mode AUTOCOMMIT
38.     connexion.autocommit = True
39. except (InterfaceError, DatabaseError) as erreur:
40.     # on affiche l'erreur
41.     print(f"L'erreur suivante s'est produite : {erreur}")
42.     # on quitte
43.     sys.exit()
44.
45. # suppression de la table personnes si elle existe
46. # si elle n'existe pas une erreur se produira - on l'ignore
47. requête = "drop table personnes"
48. try:
49.     execute_sql(connexion, requête)
50. except (InterfaceError, DatabaseError):
51.     pass
52.
53. # création de la table personnes
54. requête = "create table personnes (id int PRIMARY KEY, prenom varchar(30) NOT NULL, nom varchar(30) NOT
    NULL, age integer NOT NULL, " \
55.             "unique(nom,prenom)) "
56. try:
57.     # exécution requête
58.     execute_sql(connexion, requête)
59.     # affichage
60.     print(f"{requête} : requête réussie")
61. except (InterfaceError, DatabaseError) as erreur:
62.     # on affiche l'erreur
63.     print(f"L'erreur suivante s'est produite : {erreur}")
64. finally:
65.     # on se déconnecte
66.     connexion.close()
```

Notes :

- ligne 9 : la fonction *execute_sql* exécute une requête SQL sur une connexion ouverte ;
- ligne 14 : les opérations SQL sur la connexion se font au travers d'un objet particulier appelé *curseur ;*
- ligne 14 : obtention d'un curseur ;
- ligne 16 : exécution de la requête SQL ;
- lignes 17-20 : qu'il y ait erreur ou non, le curseur est fermé. Cela libère les ressources qui lui sont associées. Si exception il y a, alors elle n'est pas gérée ici. Elle va remonter au code appelant ;
- lignes 33-43 : création d'une connexion avec la base de données ;
- ligne 38 : le mode AUTOCOMMIT=True pour une connexion signifie que chaque exécution d'une requête s'exécute dans une transaction automatique. Le mode par défaut est AUTOCOMMIT=False où c'est le développeur qui a la responsabilité de gérer les transactions. Une transaction est un mécanisme qui englobe l'exécution de plusieurs requêtes 1 à n. Soit celles-ci

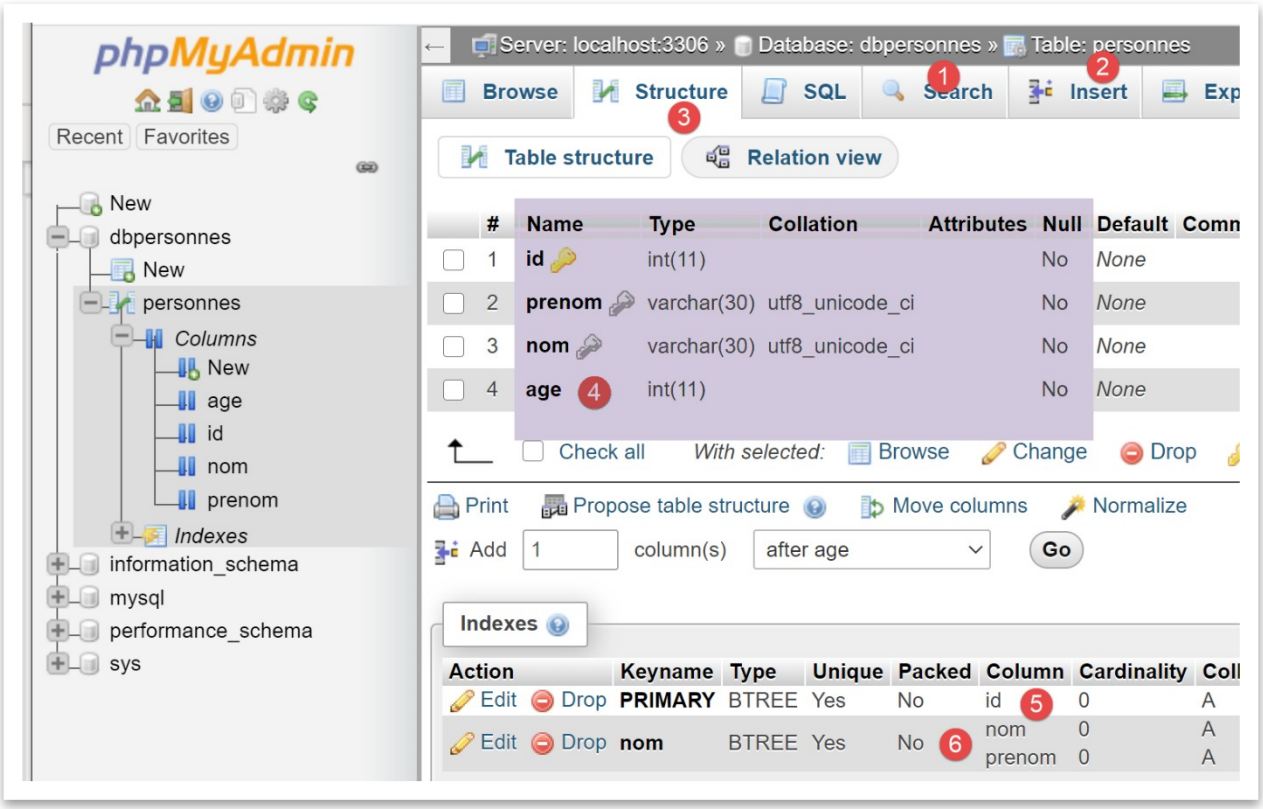
réussissent toutes, soit aucune ne réussit. Ainsi si les requêtes 1 à i réussissent mais que la requête i+1 échoue, alors les requêtes 1 à i vont être 'défaites' pour que la base retrouve l'état qu'elle avait avant l'exécution de la requête 1 ;

- ici, il y a deux requêtes SQL (lignes 49, 58). Elles seront chacune exécutées dans une transaction. Que la seconde échoue n'a aucun impact sur la première ;
- lignes 45-51 : l'ordre SQL **[drop table personnes]** est exécuté. Elle supprime la table appelée **[personnes]**. Si celle-ci n'existe pas, une erreur peut être signalée. Celle-ci est ignorée (ligne 51) ;
- lignes 53-55 : l'ordre de création de la table **[personnes]**. Une table peut être vue comme un ensemble de lignes et de colonnes. L'ordre de création précise le nom des colonnes :
  - **[id]** : un identifiant entier. Il sera unique pour chaque personne. Ce sera la clé primaire (PRIMARY KEY). Cela signifie que dans la table, cette colonne n'a pas deux fois la même valeur et qu'elle peut être utilisée pour identifier une personne ;
  - **[nom]** : une chaîne d'au plus 30 caractères ;
  - **[prenom]** : une chaîne d'au plus 30 caractères ;
  - **[age]** : un nombre entier ;
  - l'attribut **[NOT NULL]** pour chacune de ces colonnes signifie que dans une ligne de la table, aucune des trois colonnes ne peut être vide ;
  - le paramètre **[unique(nom,prenom)]** s'appelle une **contrainte**. Ici la contrainte sur les lignes est que le tuple (nom, prenom) de la ligne doit être unique dans la table. Cela signifie qu'on peut repérer de façon unique dans la table, un individu dont on connaît les nom et prénom ;
- lignes 56-60 : exécution de l'ordre SQL ;
- lignes 61-63 : gestion de l'éventuelle exception ;
- lignes 64-66 : on se déconnecte de la base de données ;

Résultats

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/mysql/mysql_03.py
2.  create table personnes (id int PRIMARY KEY, prenom varchar(30) NOT NULL, nom varchar(30) NOT NULL, age
    integer NOT NULL, unique(nom,prenom))  : requête réussie
3.
4.  Process finished with exit code 0
```

Vérification avec **[phpMyAdmin]** :

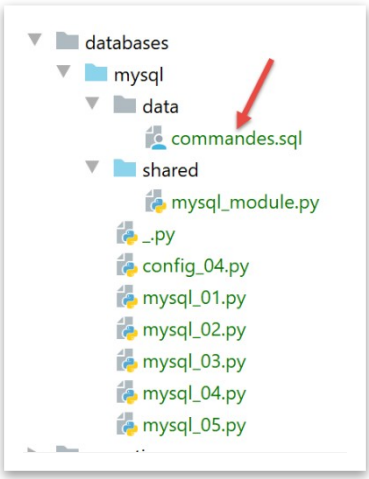


- la base de données **[dbpersonnes] [1]** a une table **[personnes] [2]** qui a la structure **[3-4]**, la clé primaire **[5]** et la contrainte d'unicité **[6]** ;

16.6 script [mysql_04] : exécution d'un fichier d'ordres SQL

Après avoir créé précédemment la table **[personnes]**, maintenant nous la remplissons puis l'exploitons à l'aide d'ordres SQL.

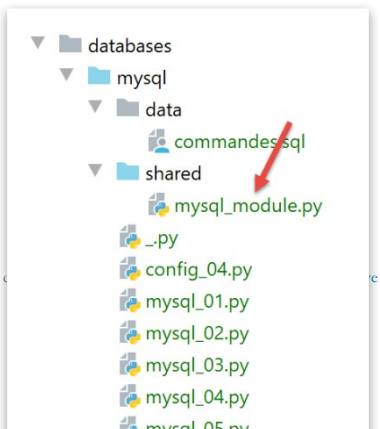
Nous souhaitons exécuter les ordres SQL d'un fichier texte :



Le contenu du fichier **[commandes.sql]** est le suivant :

```
1.  # suppression de la table [personnes]
2.  drop table personnes
3.  # création de la table personnes
4.  create table personnes (prenom varchar(30) not null, nom varchar(30) not null, age integer not null,
    primary key (nom,prenom))
5.  # insertion de deux personnes
6.  insert into personnes(prenom, nom, age) values('Paul','Langevin',48)
7.  insert into personnes(prenom, nom, age) values ('Sylvie','Lefur',70)
8.  # affichage de la table
9.  select prenom, nom, age from personnes
10. # erreur volontaire
11. xx
12. # insertion de trois personnes
13. insert into personnes(prenom, nom, age) values ('Pierre','Nicazou',35)
14. insert into personnes(prenom, nom, age) values ('Geraldine','Colou',26)
15. insert into personnes(prenom, nom, age) values ('Paulette','Girond',56)
16. # affichage de la table
17. select prenom, nom, age from personnes
18. # liste des personnes par ordre alphabétique des noms et à nom égal par ordre alphabétique des prénoms
19. select nom,prenom from personnes order by nom asc, prenom desc
20. # liste des personnes ayant un âge dans l'intervalle [20,40] par ordre décroissant de l'âge
21. # puis à âge égal par ordre alphabétique des noms et à nom égal par ordre alphabétique des prénoms
22. select nom,prenom,age from personnes where age between 20 and 40 order by age desc, nom asc, prenom asc
23. # insertion de mme Bruneau
24. insert into personnes(prenom, nom, age) values('Josette','Bruneau',46)
25. # mise à jour de son âge
26. update personnes set age=47 where nom='Bruneau'
27. # liste des personnes ayant Bruneau pour nom
28. select nom,prenom,age from personnes where nom='Bruneau'
29. # suppression de Mme Bruneau
30. delete from personnes where nom='Bruneau'
31. # liste des personnes ayant Bruneau pour nom
32. select nom,prenom,age from personnes where nom='Bruneau'
```

Nous définissons tout d'abord des fonctions que nous installons dans un module afin de pouvoir les réutiliser :



Le script **[mysql_module]** est le suivant :

```
1.  # imports
2.  from mysql.connector import DatabaseError, InterfaceError
3.  from mysql.connector.connection import MySQLConnection
4.  from mysql.connector.cursor import MySQLCursor
5.
6.
7.  # ------------------------------------------------------------------------------
8.  def afficher_infos(curseur: MySQLCursor):
9.      # affiche le résultat d'une command sql
10.     …
11.
12.
13. # ------------------------------------------------------------------------------
14. def execute_list_of_commands(connexion: MySQLConnection, sql_commands: list,
15.                              suivi: bool = False, arrêt: bool = True, with_transaction: bool = True):
16.     # utilise la connexion ouverte [connexion]
17.     # exécute sur cette connexion les commandes SQL contenues dans la liste [sql_commands]
18.     # ce fichier est un fichier de commandes SQL à exécuter à raison d'une par ligne
19.     # si suivi=True alors chaque exécution d'un ordre SQL fait l'objet d'un affichage indiquant sa réussite ou son échec
20.     # si arrêt=True, la fonction s'arrête sur la 1ère erreur rencontrée sinon elle exécute ttes les commandes sql
21.     # si with_transaction=True alors toute erreur annule l'ensemble des ordres SQL exécutés auparavant
22.     # si with_transaction=False alors une erreur n'a aucun impact sur les ordres SQL exécutés auparavant
23.     # la fonction rend une liste [erreur1, erreur2, ...]
24.
25.     ….
26.
27.
28. # ------------------------------------------------------------------------------
29. def execute_file_of_commands(connexion: MySQLConnection, sql_filename: str,
30.                              suivi: bool = False, arrêt: bool = True, with_transaction: bool = True):
31.     # utilise la connexion ouverte [connexion]
32.     # exécute sur cette connexion les commandes SQL contenues dans le fichier texte sql_filename
33.     # ce fichier est un fichier de commandes SQL à exécuter à raison d'une par ligne
34.     # si suivi=True alors chaque exécution d'un ordre SQL fait l'objet d'un affichage indiquant sa réussite ou son échec
35.     # si arrêt=True, la fonction s'arrête sur la 1ère erreur rencontrée sinon elle exécute ttes les commandes sql
36.     # si with_transaction=True alors toute erreur annule l'ensemble des ordres SQL exécutés auparavant
37.     # si with_transaction=False alors une erreur n'a aucun impact sur les ordres SQL exécutés auparavant
38.     # la fonction rend une liste [erreur1, erreur2, ...]
39.
40.     # exploitation du fichier SQL
41.     try:
42.         # ouverture du fichier en lecture
43.         file = open(sql_filename, "r")
44.         # exploitation
45.         return execute_list_of_commands(connexion, file.readlines(), suivi, arrêt, with_transaction)
46.     except BaseException as erreur:
47.         # on rend un tableau d'erreurs
48.         return [f"Le fichier {sql_filename} n'a pu être être exploité : {erreur}"]
```

Notes :

- ligne 29 : la fonction **[execute_file_of_commands]** exécute les ordres SQL contenu dans le fichier texte nommé **[sql_filename]** :
- on lira les commentaires des lignes 31-38 pour connaître la signification des paramètres ;
- lignes 40-48 : on exploite le fichier texte **[sql_filename]** ;
- ligne 43 : ouverture du fichier ;
- ligne 34 : exécution de la fonction **[execute_list_of_commands]** qui exécute les commandes SQL qu'on lui passe dans une liste. Cette liste est ici constituée par la liste de toutes les lignes du fichier texte **[file.readlines()]** (ligne 45) ;

La fonction **[execute_list_of_commands]** est la suivante :

```
1.  # ------------------------------------------------------------------------------
2.  def execute_list_of_commands(connexion: MySQLConnection, sql_commands: list,
3.                               suivi: bool = False, arrêt: bool = True, with_transaction: bool = True):
4.      # utilise la connexion ouverte [connexion]
5.      # exécute sur cette connexion les commandes SQL contenues dans la liste [sql_commands]
6.      # ce fichier est un fichier de commandes SQL à exécuter à raison d'une par ligne
7.      # si suivi=True alors chaque exécution d'un ordre SQL fait l'objet d'un affichage indiquant sa réussite ou son échec
8.      # si arrêt=True, la fonction s'arrête sur la 1ère erreur rencontrée sinon elle exécute ttes les commandes sql
9.      # si with_transaction=True alors toute erreur annule l'ensemble des ordres SQL exécutés auparavant
10.     # si with_transaction=False alors une erreur n'a aucun impact sur les ordres SQL exécutés auparavant
11.     # la fonction rend une liste [erreur1, erreur2, ...]
12.
13.     # initialisations
14.     curseur = None
15.     connexion.autocommit = not with_transaction
16.     erreurs = []
17.     try:
```

```
18.         # on demande un curseur
19.         curseur = connexion.cursor()
20.         # exécution des sql_commands SQL contenues dans sql_commands
21.         # on les exécute une à une
22.         for command in sql_commands:
23.             # on élimine les blancs de début et de fin de la commande courante
24.             command = command.strip()
25.             # a-t-on une commande vide ou un commentaire ? Si oui, on passe à la commande suivante
26.             if command == '' or command[0] == "#":
27.                 continue
28.             # exécution de la commande courante
29.             error = None
30.             try:
31.                 curseur.execute(command)
32.             except (InterfaceError, DatabaseError) as erreur:
33.                 error = erreur
34.             # y-a-t-il eu une erreur ?
35.             if error:
36.                 # une erreur de plus
37.                 msg = f"{command} : Erreur ({error})"
38.                 erreurs.append(msg)
39.                 # suivi écran ou non ?
40.                 if suivi:
41.                     print(msg)
42.                 # on s'arrête ?
43.                 if with_transaction or arrêt:
44.                     # on rend la liste d'erreurs
45.                     return erreurs
46.             else:
47.                 # pas d'erreur
48.                 if suivi:
49.                     print(f"[{command}] : Exécution réussie")
50.                 # on affiche le résultat de la command
51.                 afficher_infos(curseur)
52.         # on rend le tableau des erreurs
53.         return erreurs
54.     finally:
55.         # fermeture du curseur
56.         if curseur:
57.             curseur.close()
58.         # on valide / annule la transaction si elle existe
59.         if with_transaction:
60.             if erreurs:
61.                 # annulation
62.                 connexion.rollback()
63.             else:
64.                 # validation
65.                 connexion.commit()
```

Notes

- ligne 2 : la fonction **[execute_list_of_commands]** exécute les ordres SQL contenu dans la liste **[sql_commands]** :
- on lira les commentaires des lignes 4-11 pour connaître la signification des paramètres ;
- ligne 2 : la connexion reçue est une connexion ouverte vers une base de données ;
- ligne 15 : si on veut que l'ensemble des commandes de la liste **[sql_commands]** s'exécute au sein d'une transaction alors il faut travailler en mode AUTOCOMMIT=False. Sinon, on travaillera en mode AUTOCOMMIT=True et alors chacune des commandes de la liste **[sqlCommands]** s'exécutera au sein d'une transaction automatique et il n'y aura pas de transaction globale ;
- ligne 19 : on demande un curseur pour exécuter les différentes commandes SQL ;
- lignes 22-51 : on exécute les commandes une par une ;
- lignes 26-27 : on accepte les lignes blanches ainsi que les commentaires dans la liste des commandes SQL. Dans ce cas, on ignore simplement la commande ;
- lignes 30-33 : exécution de la requête courante ;
- lignes 35-45 : on traite le cas d'une éventuelle erreur d'exécution de la requête courante ;
- lignes 37-38 : l'erreur est ajoutée au tableau des erreurs ;
- lignes 40-41 : si un suivi a été demandé, alors le message d'erreur est affiché ;
- lignes 43-45 : si le code appelant a demandé un arrêt après la première erreur ou s'il a demandé l'utilisation d'une transaction, alors il faut s'arrêter. On rend le tableau des erreurs ;
- lignes 46-51 : cas où il n'y a pas eu d'erreur d'exécution de la requête courante ;
- lignes 48-49 : si un suivi a été demandé, on affiche la requête exécutée avec la mention 'réussie' ;
- lignes 50-51 : on affiche le résultat de la requête exécutée. On va revenir sur la fonction **[afficher_infos]** un peu plus loin ;
- lignes 54-65 : la clause **[finally]** est exécutée dans tous les cas, qu'il y ait eu une exception ou pas ;
- lignes 56-57 : fermeture du curseur. Cela libère les ressources allouées à celui-ci ;
- lignes 59-65 : on traite le cas où le code appelant a demandé à ce que les commandes SQL soient exécutées dans une transaction ;
- ligne 60 : on regarde si la liste **[erreurs]** est vide, ce qui signifie qu'aucune exception n'a eu lieu. Dans ce cas, la transaction est validée (ligne 65), sinon elle est annulée (ligne 62) ;

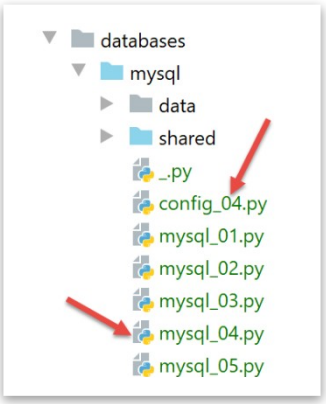
La fonction **[afficher_infos]** affiche le résultat d'une requête :

```
1.  # ------------------------------------------------------------------------------
2.  def afficher_infos(curseur: MySQLCursor):
3.      print(type(curseur))
4.      # affiche le résultat d'une command sql
5.      # s'agissait-il d'un select ?
6.      if curseur.description:
7.          # le curseur a une description - donc il a exécuté un select
8.          # description[i] est la description de la colonne n° i du select
9.          # description[i][0] est le nom de la colonne n° i du select
10.         # on affiche les noms des champs
11.         titre = ""
12.         for i in range(len(curseur.description)):
13.             titre += curseur.description[i][0] + ", "
14.         # on affiche la liste des champs sans la virgule de fin
15.         print(titre[0:len(titre) - 1])
16.         # ligne séparatrice
17.         print("*" * (len(titre) - 1))
18.         # ligne courante du select
19.         ligne = curseur.fetchone()
20.         while ligne:
21.             # on l'affiche
22.             print(ligne)
23.             # ligne suivante du select
24.             ligne = curseur.fetchone()
25.         # ligne séparatrice
26.         print("*" * (len(titre) - 1))
27.     else:
28.         # le curseur n'a pas de champ [description] - il a donc exécuté un ordre SQL
29.         # de mise à jour (insert, delete, update)
30.         print(f"nombre de lignes modifiées : {curseur.rowcount}")
```

Notes

- ligne 1 : le paramètre de la fonction est le curseur qui vient d'exécuter un ordre SQL. Selon que cet ordre est un SELECT ou un ordre de mise à jour INSERT, UPDATE, DELETE, le contenu du curseur n'est pas le même ;
- ligne 6 : si le curseur a le champ **[description]** alors il a exécuté un SELECT et **[description]** décrit les champs demandés dans le SELECT :
  - **description[i]** décrit le champ n° i demandé par le SELECT. C'est une liste ;
  - **description[i][0]** est le nom du champ n° i ;
- lignes 11-17 : on affiche le nom des champs demandés par le SELECT ;
- lignes 18-24 : on exploite le résultat du SELECT ;
- lignes 20, 24 : le résultat d'un SELECT s'exploite séquentiellement. Ce résultat est un ensemble de lignes. La ligne courante est obtenue par **[curseur.fetchone()]** (ligne 19). On obtient alors un tuple ;
- lignes 27-30 : si le curseur n'a pas le champ **[description]** alors il a exécuté un ordre de mise à jour INSERT, UPDATE, DELETE. On peut alors savoir combien de lignes de la table ont été modifiées par l'exécution de cet ordre ;
- ligne 30 : **[curseur.rowcount]** est ce nombre ;

Le script principal **[mysql-04]** utilise le module **[mysql_module]** que nous venons de décrire :



Le fichier **[config_04]** configure le contexte d'exécution du script **[mysql_04]** :

```
def configure():
    import os

    # chemin absolu du dossier du fichier de configuration
    script_dir = os.path.dirname(os.path.abspath(__file__))
    # configuration des dossiers du syspath
    absolute_dependencies = [
        # dossiers locaux
        f"{script_dir}/shared",
    ]

    # fixation du syspath
    from myutils import set_syspath
    set_syspath(absolute_dependencies)

    # on rend la config
    return {
        # fichier des commandes SQL
        "commands_filename": f"{script_dir}/data/commandes.sql",
        # identifiants de la connexion à la base de données
        "host": "localhost",
        "database": "dbpersonnes",
        "user": "admpersonnes",
        "password": "nobody"
    }
```

Le script **[mysql_04]** est le suivant :

```
# on récupère la configuration de l'application
import config_04

config = config_04.configure()

# le syspath est configuré - on peut faire les imports
import sys
from mysql_module import execute_file_of_commands
from mysql.connector import connect, DatabaseError, InterfaceError

# ------------------------------------------- main
# vérification de la syntaxe de l'appel
# argv[0] true / false
args = sys.argv
erreur = len(args) != 2
if not erreur:
    with_transaction = args[1].lower()
    erreur = with_transaction != "true" and with_transaction != "false"
# erreur ?
if erreur:
    print(f"syntaxe : {args[0]} true / false")
    sys.exit()

# calcul d'un texte
with_transaction = with_transaction == "true"
if with_transaction:
    texte = "avec transaction"
else:
    texte = "sans transaction"

# logs écran
print("-----------------------------------------------------------------------")
print(f"Exécution du fichier SQL {config['commands_filename']} {texte}")
print("-----------------------------------------------------------------------")

# exécution des ordres SQL du fichier
connexion = None
try:
    # connexion à la bd
    connexion = connect(host=config['host'], user=config['user'], password=config['password'],
                        database=config['database'])
    # exécution du fichier des commandes SQL
    erreurs = execute_file_of_commands(connexion, config["commands_filename"], suivi=True, arrêt=False,
                                       with_transaction=with_transaction)
except (InterfaceError, DatabaseError) as erreur:
    # affichage de l'erreur
    print(f"L'erreur fatale suivante s'est produite : {erreur}")
    # on s'arrête
    sys.exit()
```

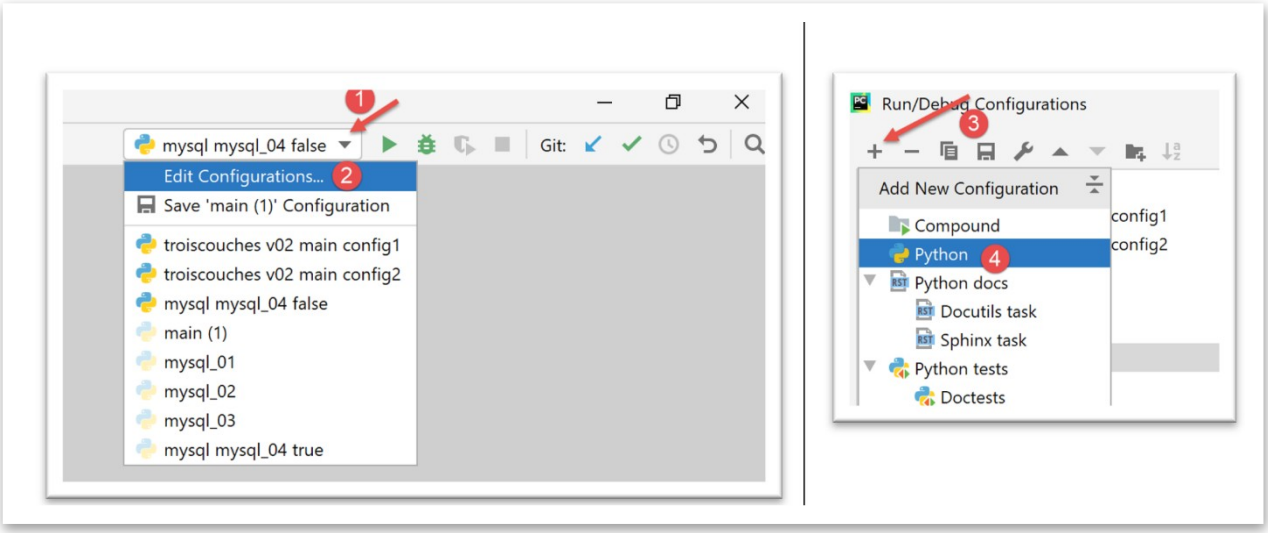
```
50. finally:
51.     # fermeture de la connexion si elle a été ouverte
52.     if connexion:
53.         connexion.close()
54.
55. # affichage nombre d'erreurs
56. print("-------------------------------------------------------------------------")
57. print(f"Exécution terminée")
58. print("-------------------------------------------------------------------------")
59. print(f"Il y a eu {len(erreurs)} erreur(s)")
60. # affichage des erreurs
61. for erreur in erreurs:
62.     print(erreur)
```

Notes

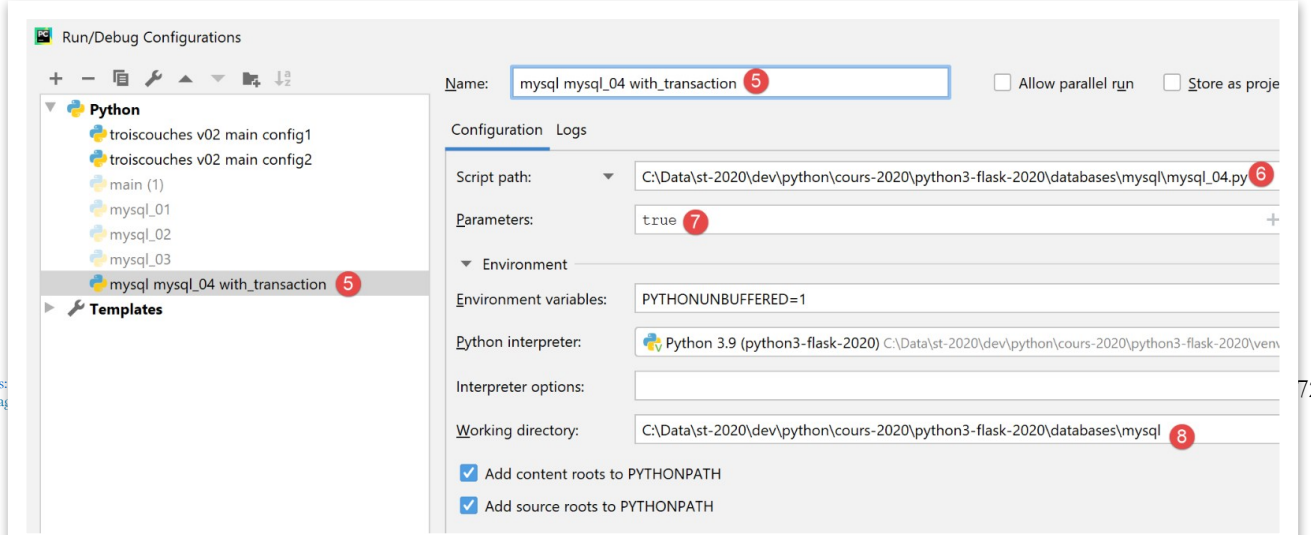
- lignes 1-4 : configuration du script ;
- ligne 8 : import du module **[mysql_module]** décrit précédemment :
- lignes 12-22 : le script **[mysql-04]** attend un paramètre qui doit avoir l'une des valeurs **[true / false]**. Ce paramètre indique si le fichier d'ordres SQL doit être exécuté au sein d'une transaction (true) ou pas (false) ;
- ligne 14 : les paramètres passés par l'utilisateur au script sont trouvés dans la liste **[sys.argv]** ;
- ligne 15 : il faut deux paramètres, par exemple **[mysql-04 true]**. Le nom du script compte comme un paramètre ;
- lignes 17-18 : s'il y a bien deux paramètres, il faut que le 2e soit une chaîne de caractères de valeur 'true' ou 'false' ;
- lignes 24-29 : calcul d'un texte affiché ligne 33 ;
- lignes 39-44 : on exécute les commandes du fichier **[./data/commandes.sql]** ;
- lignes 45-49 : s'il se produit une erreur à la connexion (ligne 40) ou non gérée par le script **[execute_file_of_commands]** on affiche l'erreur et on arrête tout ;
- lignes 55-62 : en cas d'exécution réussie, on affiche le nombre d'erreurs rencontrées dans l'exécution des commandes SQL ;

Exécution n° 1

On fait d'abord une exécution sans transaction. Pour cela, on va créer une configuration d'exécution comme il a été fait dans le paragraphe |configuration d'un contexte d'exécution|:



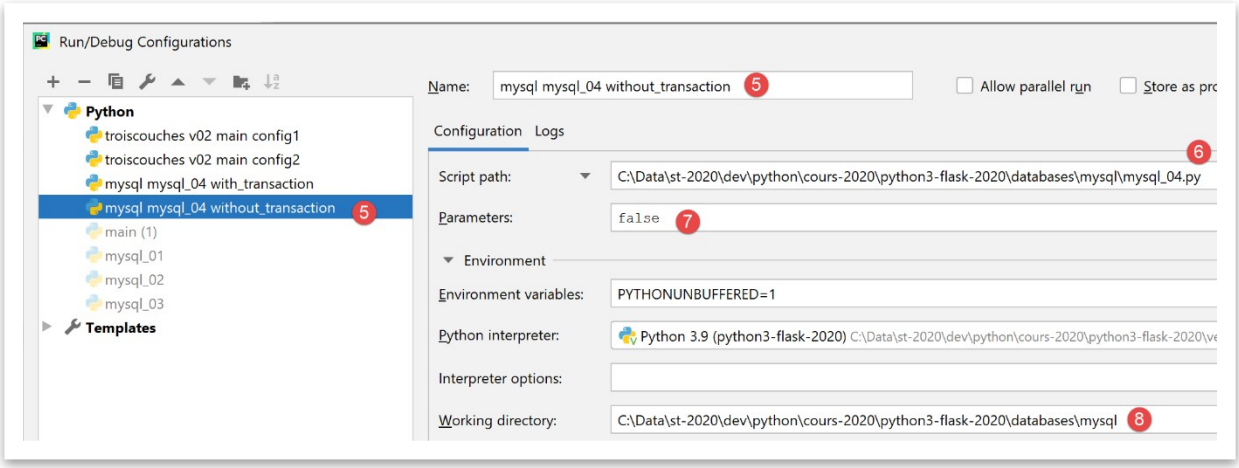
- en **[1-4]**, on crée une configuration d'exécution Python ;



- **[5]** : nom de la configuration d'xécution ;
- **[6]** : chemin du script à exécuter ;
- **[7]** : paramètres du script ;
- **[8]** : dossier d'exécution ;

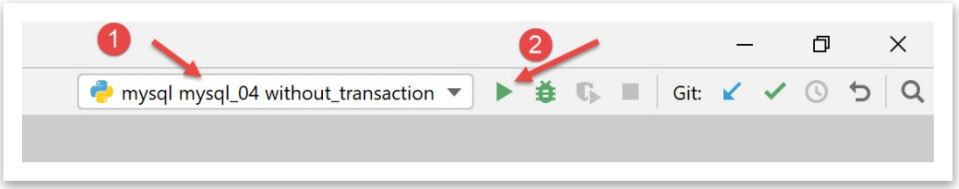
Cette configuration correspond donc à une exécution du fichier SQL avec une transaction. Utiliser le bouton **[Apply]** pour valider la configuration.

Nous créons de la même façon la configuration d'exécution **[mysql mysql-04 without_transaction]** :



Cette configuration correspond donc à une exécution du fichier SQL sans transaction. Utiliser le bouton **[Apply]** pour valider la configuration.

Nous exécutons d'abord la version sans transaction :



Les résultats sont alors les suivants :

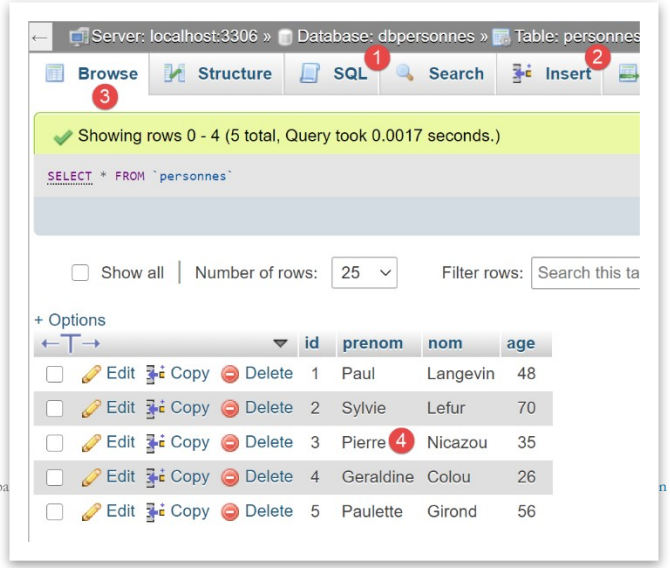
```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/mysql/mysql_04.py false
2.  ----------------------------------------------------------------------
3.  Exécution du fichier SQL C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\databases\mysql/data/
    commandes.sql sans transaction
4.  ----------------------------------------------------------------------
5.  [drop table personnes] : Exécution réussie
6.  nombre de lignes modifiées : 0
7.  [create table personnes (id int primary key, prenom varchar(30) not null, nom varchar(30) not null, age
    integer not null, unique (nom,prenom))] : Exécution réussie
8.  nombre de lignes modifiées : 0
9.  [insert into personnes(id, prenom, nom, age) values(1, 'Paul','Langevin',48)] : Exécution réussie
10. nombre de lignes modifiées : 1
11. [insert into personnes(id, prenom, nom, age) values (2, 'Sylvie','Lefur',70)] : Exécution réussie
12. nombre de lignes modifiées : 1
13. [select prenom, nom, age from personnes] : Exécution réussie
14. prenom, nom, age,
15. *****************
16. ('Paul', 'Langevin', 48)
17. ('Sylvie', 'Lefur', 70)
18. *****************
19. xx : Erreur (1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your
    MySQL server version for the right syntax to use near 'xx' at line 1)
20. [insert into personnes(id, prenom, nom, age) values (3, 'Pierre','Nicazou',35)] : Exécution réussie
21. nombre de lignes modifiées : 1
22. [insert into personnes(id, prenom, nom, age) values (4, 'Geraldine','Colou',26)] : Exécution réussie
23. nombre de lignes modifiées : 1
```

```
24. [insert into personnes(id, prenom, nom, age) values (5, 'Paulette','Girond',56)] : Exécution réussie
25. nombre de lignes modifiées : 1
26. [select prenom, nom, age from personnes] : Exécution réussie
27. prenom, nom, age,
28. *****************
29. ('Paul', 'Langevin', 48)
30. ('Sylvie', 'Lefur', 70)
31. ('Pierre', 'Nicazou', 35)
32. ('Geraldine', 'Colou', 26)
33. ('Paulette', 'Girond', 56)
34. *****************
35. [select nom,prenom from personnes order by nom asc, prenom desc] : Exécution réussie
36. nom, prenom,
37. ************
38. ('Colou', 'Geraldine')
39. ('Girond', 'Paulette')
40. ('Langevin', 'Paul')
41. ('Lefur', 'Sylvie')
42. ('Nicazou', 'Pierre')
43. ************
44. [select nom,prenom,age from personnes where age between 20 and 40 order by age desc, nom asc, prenom asc] :
    Exécution réussie
45. nom, prenom, age,
46. *****************
47. ('Nicazou', 'Pierre', 35)
48. ('Colou', 'Geraldine', 26)
49. *****************
50. [insert into personnes(id, prenom, nom, age) values(6, 'Josette','Bruneau',46)] : Exécution réussie
51. nombre de lignes modifiées : 1
52. [update personnes set age=47 where nom='Bruneau'] : Exécution réussie
53. nombre de lignes modifiées : 1
54. [select nom,prenom,age from personnes where nom='Bruneau'] : Exécution réussie
55. nom, prenom, age,
56. *****************
57. ('Bruneau', 'Josette', 47)
58. *****************
59. [delete from personnes where nom='Bruneau'] : Exécution réussie
60. nombre de lignes modifiées : 1
61. [select nom,prenom,age from personnes where nom='Bruneau'] : Exécution réussie
62. nom, prenom, age,
63. *****************
64. *****************
65. ------------------------------------------------------------------------
66. Exécution terminée
67. ------------------------------------------------------------------------
68. Il y a eu 1 erreur(s)
69. xx : Erreur (1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your
    MySQL server version for the right syntax to use near 'xx' at line 1)
70.
71. Process finished with exit code 0
```

Notes :

- ligne 19 : on voit qu'après l'erreur l'exécution des ordres SQL a continué, ceci parce que l'exécution s'est faite sans transaction et avec le paramètre **[arrêt=False]**. Tous les ordres SQL ont donc été exécutés. On devrait donc avoir une table **[personnes]** reflétant cette exécution ;

Vérification avec phpMyAdmin :



Exécution n° 2

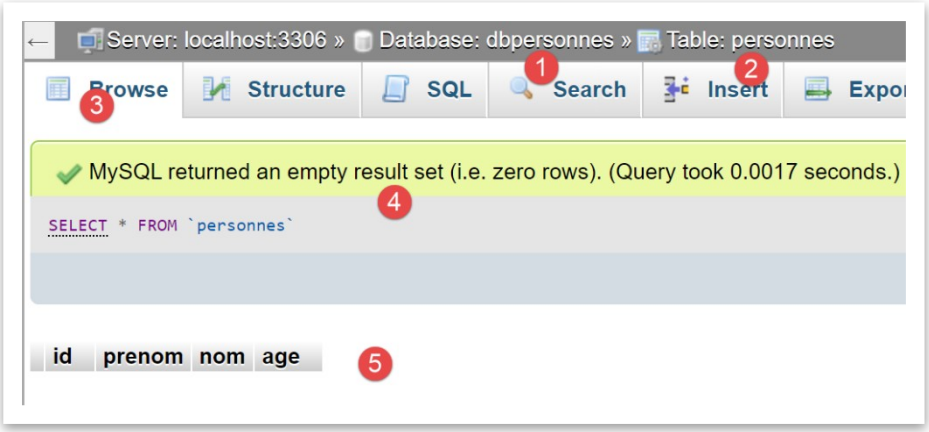
Nous exécutons maintenant la configuration **[mysql mysql-04 with_transaction]**. Les résultats sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/mysql/mysql_04.py true
2.  ----------------------------------------------------------------------
3.  Exécution du fichier SQL C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\databases\mysql/data/
    commandes.sql avec transaction
4.  ----------------------------------------------------------------------
5.  [drop table personnes] : Exécution réussie
6.  nombre de lignes modifiées : 0
7.  [create table personnes (id int primary key, prenom varchar(30) not null, nom varchar(30) not null, age
    integer not null, unique (nom,prenom))] : Exécution réussie
8.  nombre de lignes modifiées : 0
9.  [insert into personnes(id, prenom, nom, age) values(1, 'Paul','Langevin',48)] : Exécution réussie
10. nombre de lignes modifiées : 1
11. [insert into personnes(id, prenom, nom, age) values (2, 'Sylvie','Lefur',70)] : Exécution réussie
12. nombre de lignes modifiées : 1
13. [select prenom, nom, age from personnes] : Exécution réussie
14. prenom, nom, age,
15. **********
16. ('Paul', 'Langevin', 48)
17. ('Sylvie', 'Lefur', 70)
18. **********
19. xx : Erreur (1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your
    MySQL server version for the right syntax to use near 'xx' at line 1)
20. ----------------------------------------------------------------------
21. Exécution terminée
22. ----------------------------------------------------------------------
23. Il y a eu 1 erreur(s)
24. xx : Erreur (1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your
    MySQL server version for the right syntax to use near 'xx' at line 1)
25.
26. Process finished with exit code 0
```

Notes :

- ligne 19 : on voit qu'après l'erreur il n'y a plus d'exécution d'ordres SQL, ceci parce que l'exécution s'est faite dans une transaction et qu'à la 1re erreur rencontrée nous avons défait la transaction et arrêté l'exécution des ordres SQL. Ceci signifie que le résultat des ordres des lignes 9, 11, 13 a été défait. On devrait donc avoir une table **[personnes]** vide ;

Vérifications avec phpMyAdmin :



- en **[5]**, on voit que la table **[personnes]** **[2]** est vide ;

16.7 script [mysql_05] : utilisation de requêtes paramétrées

Le script **[mysql_05]** introduit la notion de requêtes paramétrées :

```
1.  # imports
2.  from mysql.connector import connect, DatabaseError, InterfaceError
3.
```

```
4.  # l'identité de l'utilisateur
5.  ID = "admpersonnes"
6.  PWD = "nobody"
7.  # la machine hôte du sgbd
8.  HOST = "localhost"
9.  # identité de la base
10. BASE = "dbpersonnes"
11.
12. # liste de personnes (nom,prenom,age)
13. personnes = []
14. for i in range(5):
15.     personnes.append((i, f"n0{i}", f"p0{i}", i + 10))
16. personnes.append((40, "d'Aboot", "Y'éna", 18))
17. # autre liste de personnes
18. autresPersonnes = []
19. for i in range(5):
20.     autresPersonnes.append((i + 100, f"n1{i}", f"p1{i}", i + 20))
21. autresPersonnes.append((200, "d'Aboot", "F'ilhem", 34))
22.
23. # accès au SGBD
24. connexion = None
25. try:
26.     # connexion
27.     connexion = connect(host=HOST, user=ID, password=PWD, database=BASE)
28.     # curseur
29.     curseur = connexion.cursor()
30.     # suppression des enregistrement existants
31.     curseur.execute("delete from personnes")
32.     # insertions personne par personne avec une requête préparée
33.     for personne in personnes:
34.         curseur.execute("insert into personnes(id,nom,prenom,age) values(%s,%s,%s,%s)", personne)
35.     # insertion en bloc d'une liste de personnes
36.     curseur.executemany("insert into personnes(id,nom,prenom,age) values(%s, %s,%s,%s)", autresPersonnes)
37.     # validation de la transaction
38.     connexion.commit()
39. except (DatabaseError, InterfaceError) as erreur:
40.     # affichage erreur
41.     print(f"L'erreur suivante s'est produite : {erreur}")
42.     # annulation transaction
43.     if connexion:
44.         connexion.rollback()
45. finally:
46.     # fermeture connexion
47.     if connexion:
48.         connexion.close()
```

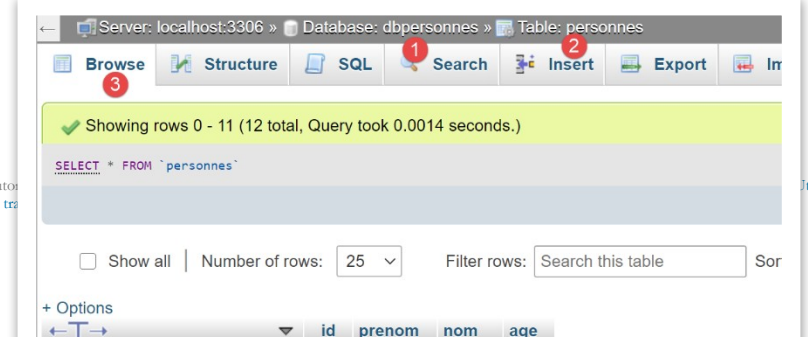
Notes

- lignes 12-21 : on crée deux listes de personnes à inclure dans la base de données **[dbpersonnes]** ;
- ligne 27 : connexion à la base de données ;
- ligne 31 : suppression du contenu de la table **[personnes]** ;
- lignes 33-34 : insertion de personnes avec une requête paramétrée. Ligne 34, le 1er paramètre est l'ordre SQL à exécuter. Celui-ci est incomplet. Il contient des paramètres **[%s]** qui vont être remplacés un par un et dans l'ordre par les valeurs de la liste du second paramètre ;
- ligne 36 : insertion de personnes avec cette fois une seule instruction **[curseur.executemany]**. Le second paramètre de **[executemany]** est alors une liste de listes ;

L'intérêt des requêtes paramétrées réside en deux points :

- elles sont exécutées plus rapidement que des requêtes en 'dur' qu'il faut analyser à chaque exécution. La requête paramétrée **[executemany]** n'est analysée qu'une fois. Ensuite elle est exécutée n fois sans être analysée de nouveau ;
- les paramètres injectés dans la requête paramétrée sont vérifiés. S'ils contiennent des caractères réservés, comme l'apostrophe par exemple, ceux-ci sont 'protégés' afin qu'ils n'interfèrent pas dans l'exécution de l'ordre SQL. C'est pour vérifier ce point qu'on a inclus des noms et prénoms avec apostrophes dans la liste (lignes 16 et 21) ;

Les résultats obtenus dans phpMyAdmin sont les suivants :



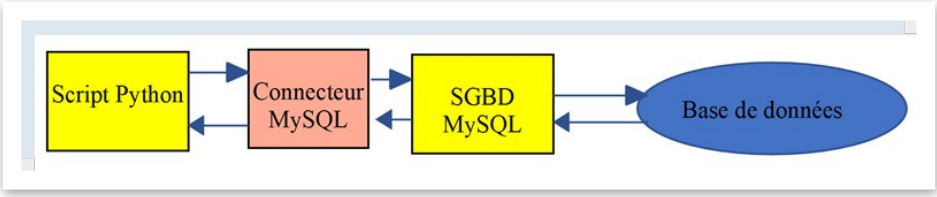
- on notera que les chaînes ayant une apostrophe, caractère réservé en SQL, ont été correctement insérées. La requête paramétrée les a 'protégées'. Sans requête paramétrée, il aurait fallu faire ce travail nous-mêmes ;

17 Utilisation du SGBD PostgreSQL

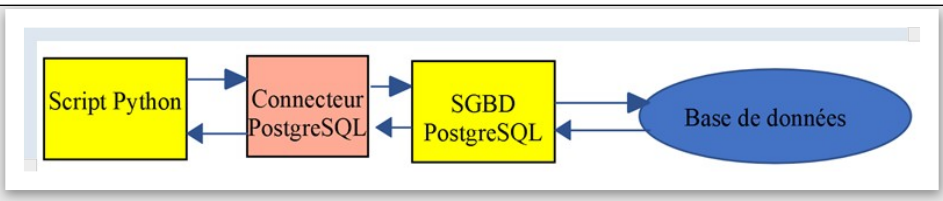
Le SGBD PostgreSQL est librement disponible. Il est une alternative à la version 'community' de MySQL.

Nous l'utilisons ici pour montrer qu'il est assez simple de migrer des scripts Python / MySQL vers des scripts Python / PostgreSQL.

Avec le SGBD MySQL, l'architecture de nos scripts était la suivante :

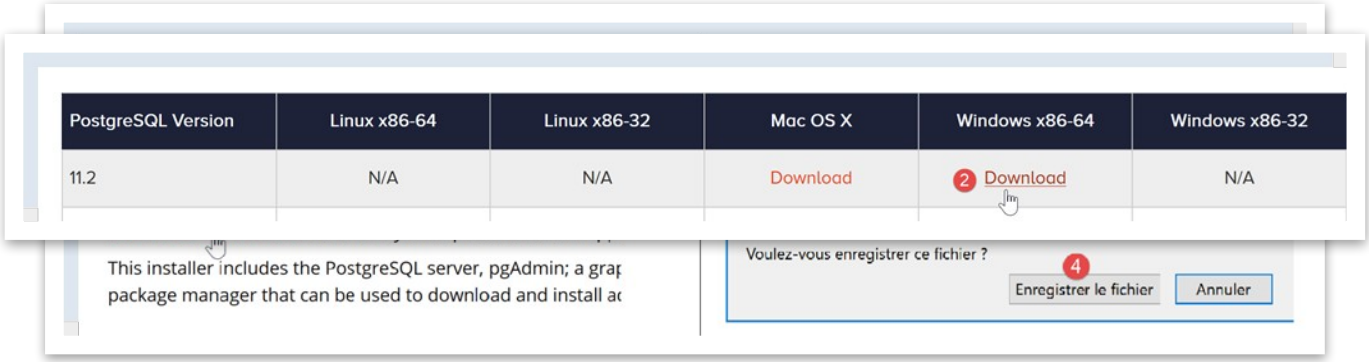


Avec le SGBD PostgreSQL, elle sera la suivante :



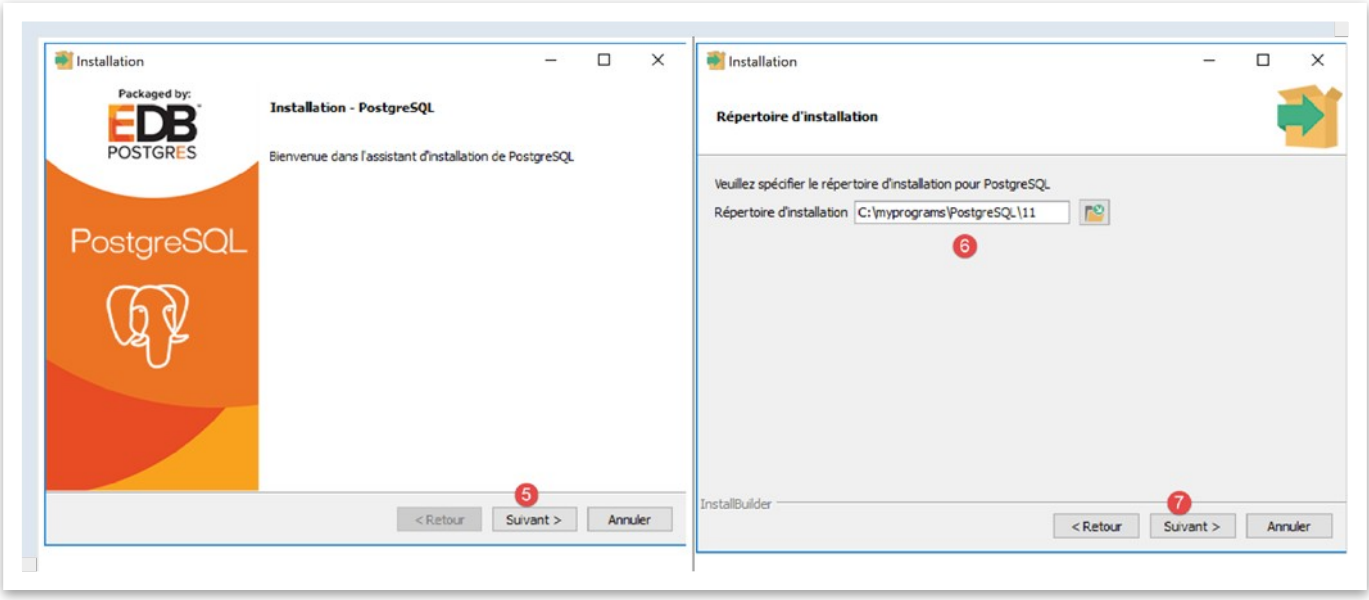
17.1 Installation du SGBD PostgreSQL

Les distributions du SGBD PostgreSQL sont disponibles à l'URL **`[https://www.postgresql.org/download/]`** (mai 2019). Nous montrons l'installation de la version pour Windows 64 bits :

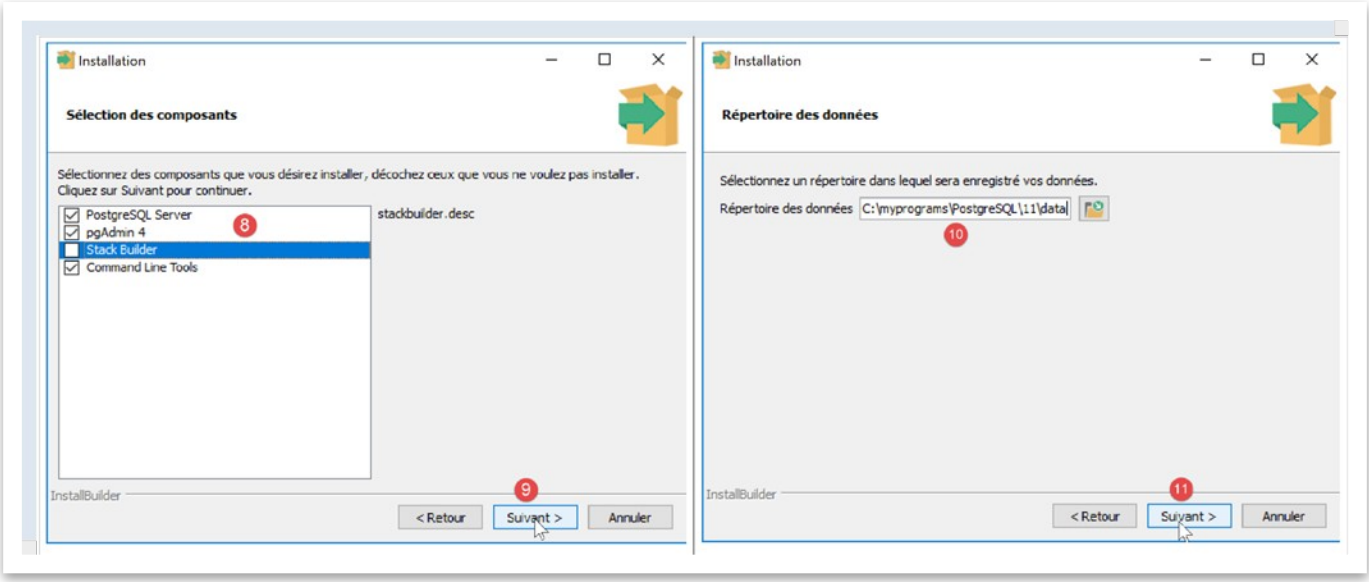


- en **`[1-4]`**, on télécharge l'installateur du SGBD ;

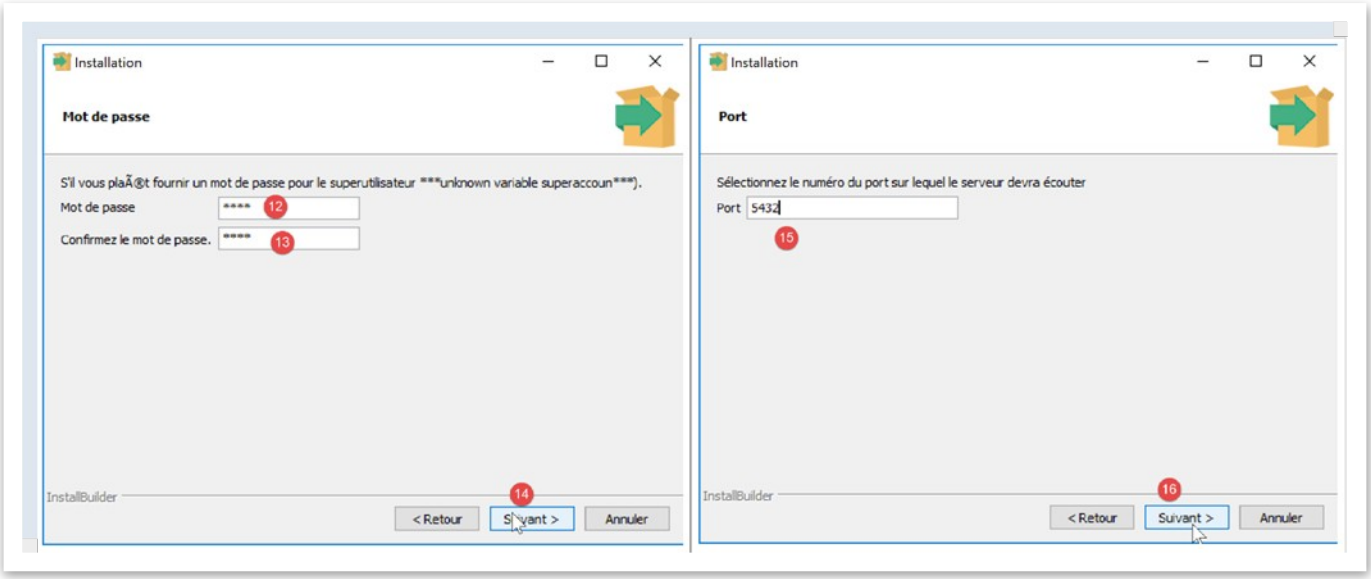
On lance l'installateur téléchargé :



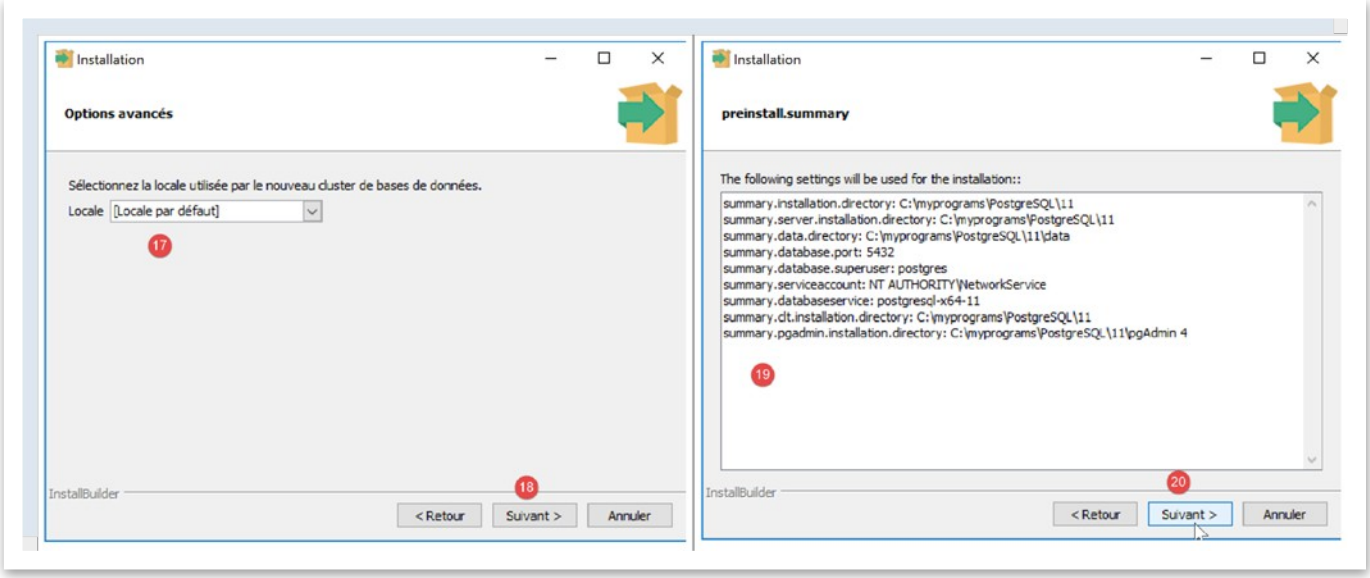
- en **[6]**, indiquez un dossier d'installation ;



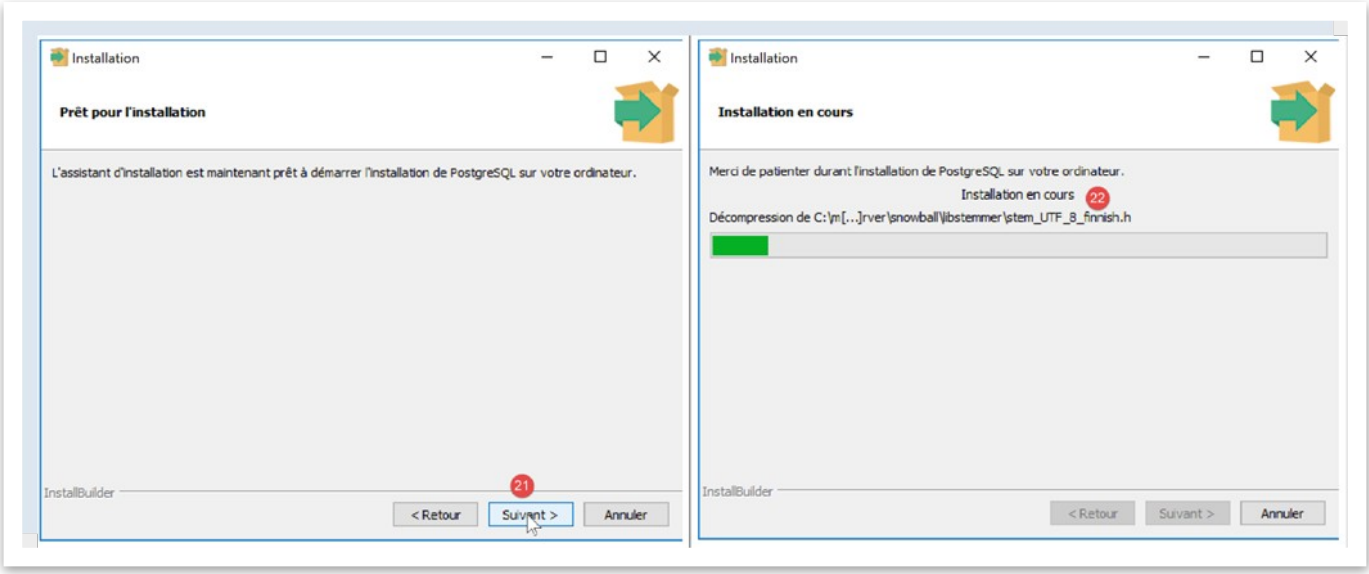
- en **[8]**, l'option **[Stack Builder]** est inutile pour ce qu'on veut faire ici ;
- en **[10]**, laissez la valeur qui vous sera présentée ;

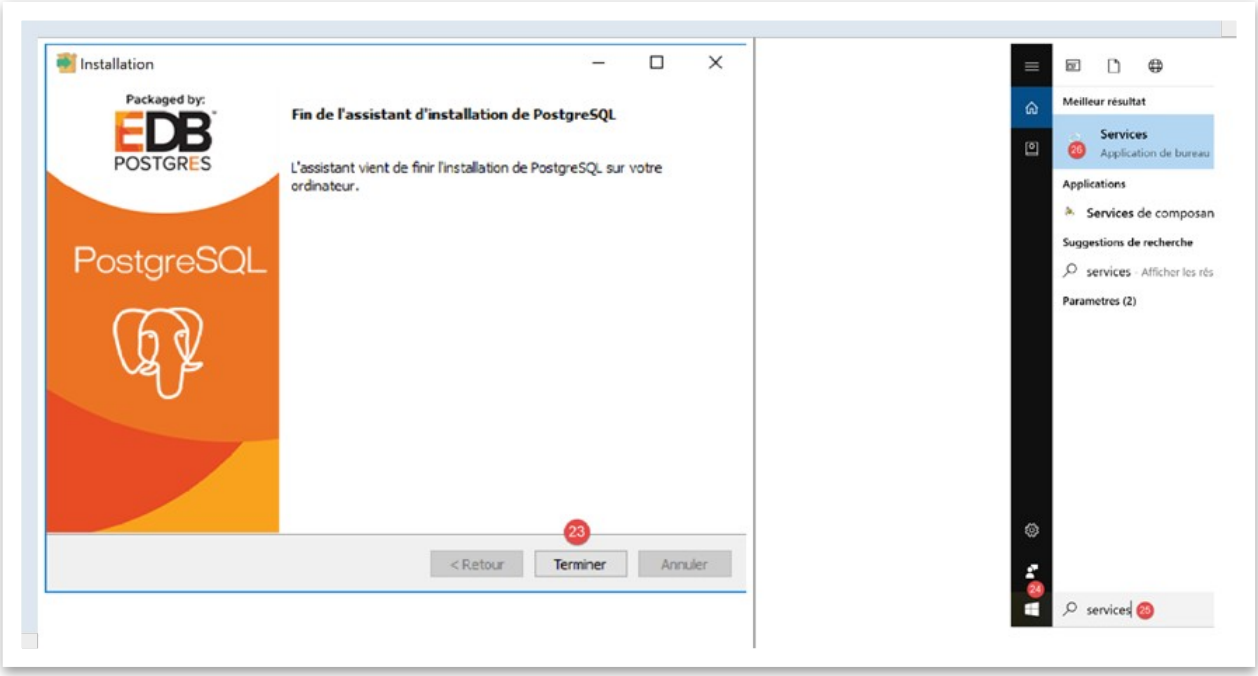


- en **[12-13]**, on a mis ici le mot de passe **[root]**. Ce sera le mot de passe de l'administrateur du SGBD qui s'appelle **[postgres]**. PostgreSQL l'appelle également le super-utilisateur ;
- en **[15]**, laissez la valeur par défaut : c'est le port d'écoute du SGBD ;

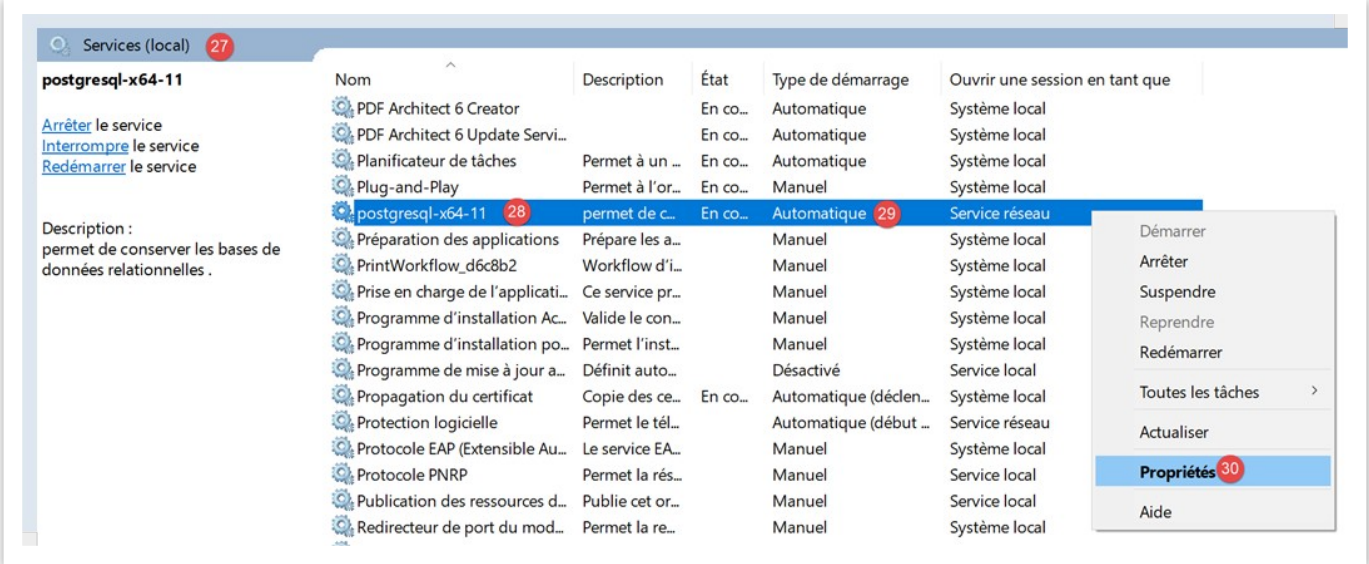


- en **[17]**, laissez la valeur par défaut ;
- en **[19]**, le résumé de la configuration de l'installation ;

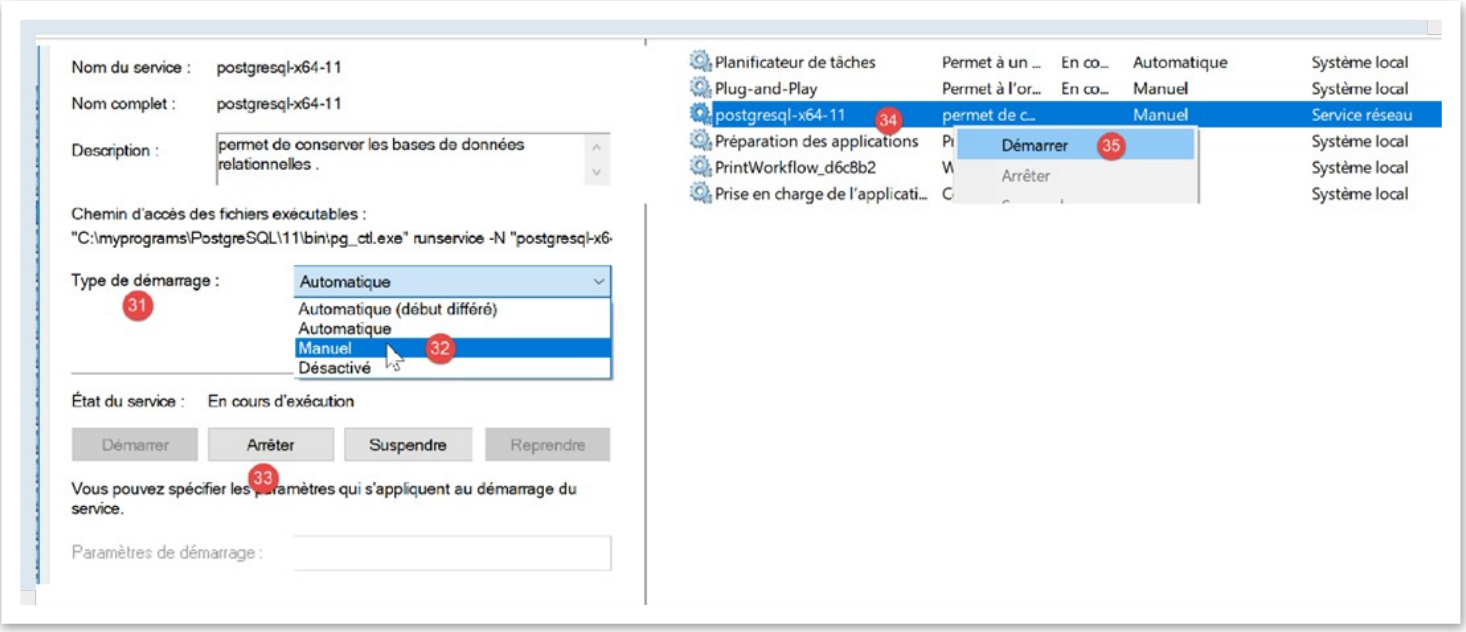




Sous windows, le SGBD PostgreSQL est installé comme un service windows lancé automatiquement. La plupart du temps ce n'est pas souhaitable. Nous allons modifier cette configuration. Tapez **[services]** dans la barre de recherche de Windows **[24-26]** :



- en **[29]**, on voit que le service du SGBD PostgreSQL est en mode automatique. On change cela en accédant aux propriétés du service **[30]** :

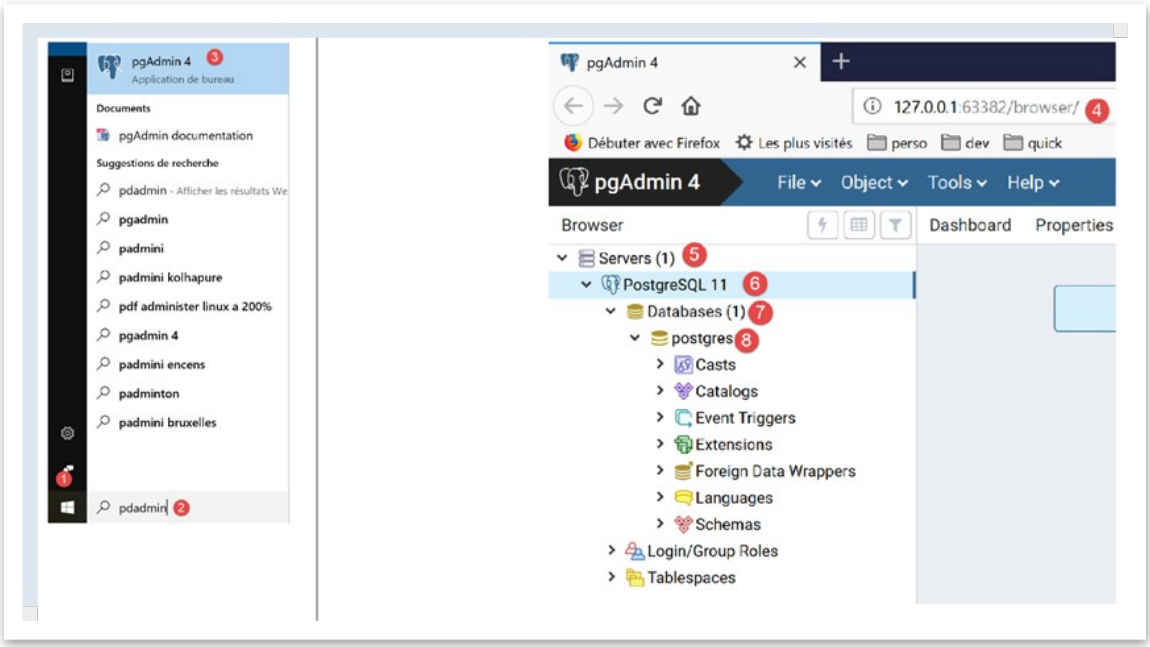


- en **[31-32]**, mettez le démarrage en mode manuel ;
- en **[33]**, arrêtez le service ;

Lorsque vous voudrez démarrer manuellement le SGBD, revenez à l'application **[services]**, cliquez droit sur le service **[postgresql]** (34) et lancez le (35).

17.2 Administrer PostgreSQL avec l'outil [pgAdmin]

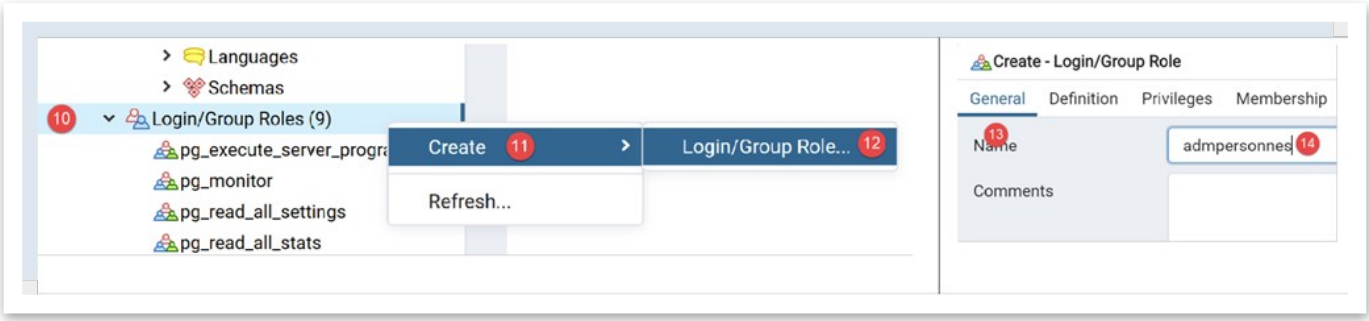
Lancez le service windows du SGBD PostgreSQL (paragraphe précédent). Puis de la même façon que vous avez lancé l'outil **[services]**, lancez l'outil **[pgadmin]** qui permet d'administrer le SGBD PostgreSQL **[1-3]** :

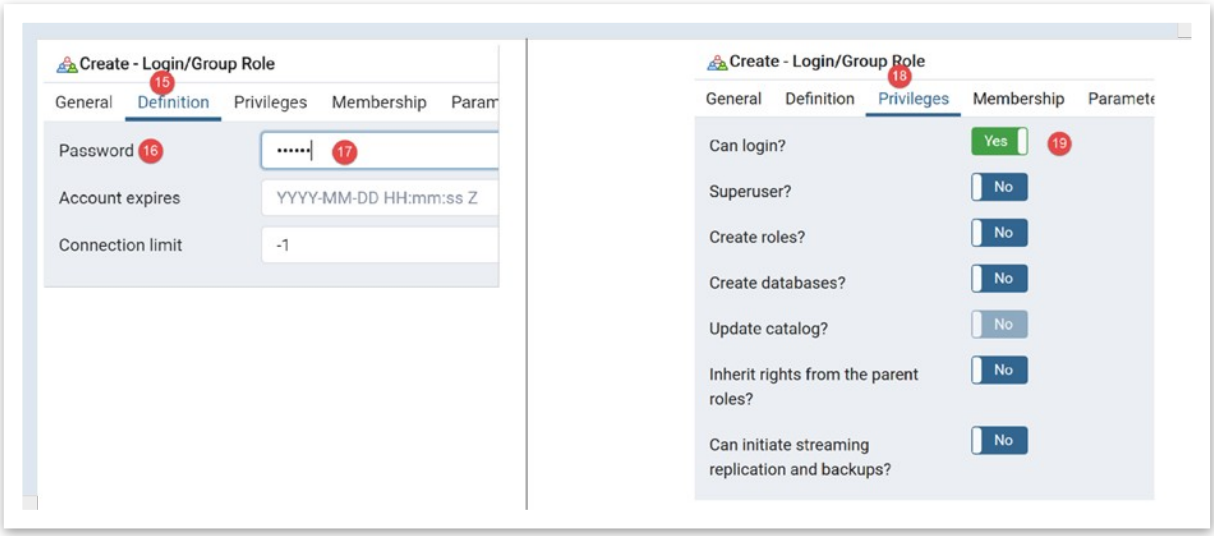


Il est possible qu'à un moment donné on vous demande le mot de passe du super-utilisateur. Celui-ci s'appelle **[postgres]**. Vous avez défini son mot de passe lors de l'installation du SGBD. Dans ce document, nous avons donné le mot de passe **[root]** au super-utilisateur lors de l'installation.

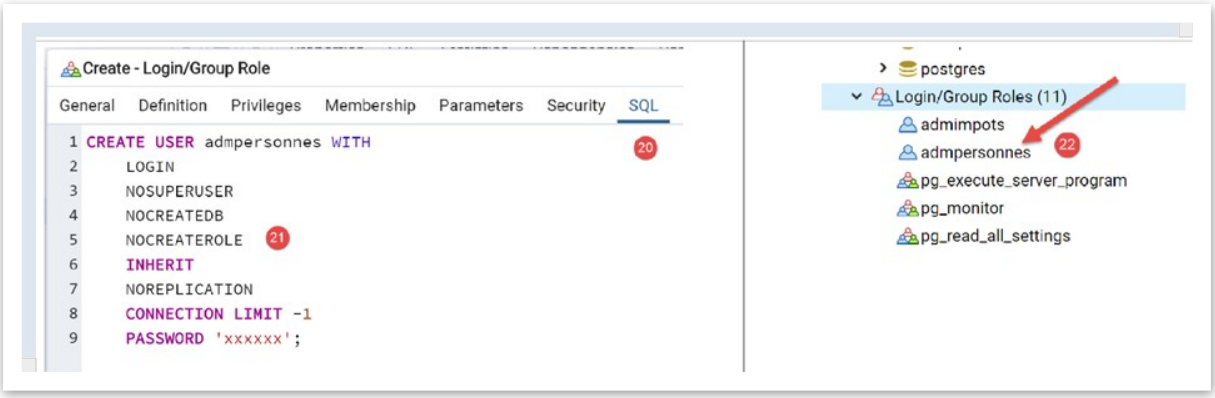
- en **[4]**, **[pgAdmin]** est une application web ;
- en **[5]**, la liste des serveurs PostgreSQL détectés par **[pgAdmin]**, ici 1 ;
- en **[6]**, le serveur PostgreSQL que nous avons lancé ;
- en **[7]**, les bases de données du SGBD, ici 1 ;
- en **[8]**, la base **[postgresql]** est gérée par le super-utilisateur **[postgres]** ;

Créons tout d'abord un utilisateur **[admpersonnes]** avec le mot de passe **[nobody]** :



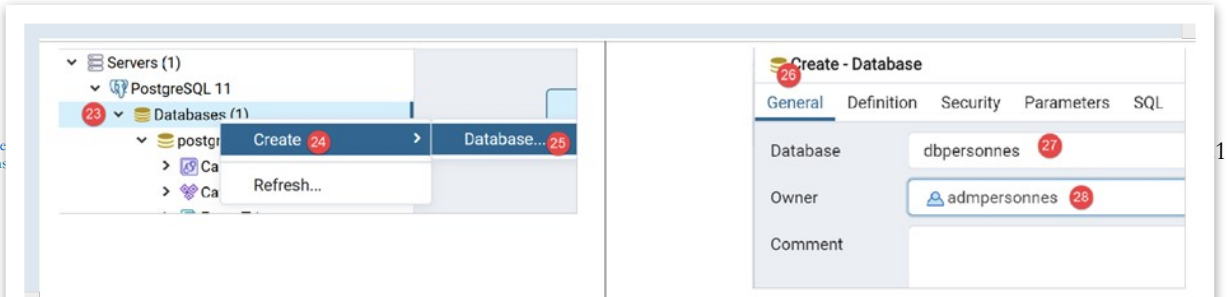


- en **[17]**, on a mis **[nobody]** ;

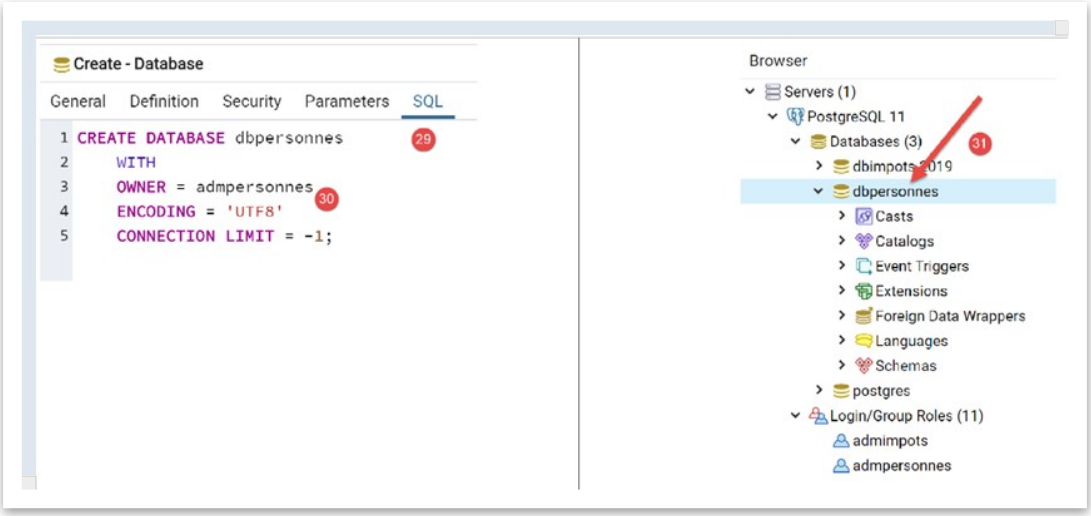


- en **[21]**, le code SQL que va émettre l'outil **[pgAdmin]** vers le SGBD PostgreSQL. C'est une façon d'apprendre le langage SQL propriétaire de PostgreSQL ;
- en **[22]**, après validation de l'assistant **[Save]**, l'utilisateur **[admpersonnes]** a été créé ;

Maintenant nous créons la base **[dbpersonnes]** :



On clique droit sur **[23]**, puis **[24-25]** pour créer une nouvelle base de données. Dans l'onglet **[26]**, on définit le nom de la base **[27]** et son propriétaire **[admpersonnes] [28]**.



- en **[30]**, le code SQL de création de la base ;
- en **[31]**, après validation de l'assistant **[Save]**, la base **[dbpersonnes]** est créée ;

Nous allons exploiter la base **[dbpersonnes]** avec des scripts Python.

17.3 Installation du connecteur Python du SGBD PostgreSQL



Dans le schéma ci-dessus est représenté un connecteur faisant le lien entre les scripts Python et le SGBD PostgreSQL. Il en existe plusieurs. Nous installons le connecteur **[psycopg2]**. Cela se fait dans un terminal Python (peu importe le dossier dans lequel est ouvert ce terminal). Le connecteur est installé par la commande **[pip install psycopg2]** :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\troiscouches\v01\tests>pip install psycopg2
2.  Collecting psycopg2
3.    Downloading psycopg2-2.8.5-cp38-cp38-win_amd64.whl (1.1 MB)
4.       || 1.1 MB 3.2 MB/s
5.  Installing collected packages: psycopg2
6.  Successfully installed psycopg2-2.8.5
```

17.4 Portage des scripts MySQL vers des scripts PostgreSQL



- le dossier **[1]** des scripts MySQL est dupliqué (Ctrl-C / Ctrl-V), puis les noms des fichiers sont changés mais par leur contenu ;

17.4.1 module [pgres_module]

Ce module est la copie du module **[mysql_module]** (cf paragraphe |script [mysql-04] : exécution d'un fichier d'ordres SQL|). On change les imports :

Au lieu de :

```
1.  # imports
2.  from mysql.connector import DatabaseError, InterfaceError
3.  from mysql.connector.connection import MySQLConnection
4.  from mysql.connector.cursor import MySQLCursor
```

on écrit :

```
1.  # imports
2.  from psycopg2 import DatabaseError, InterfaceError
3.  from psycopg2.extensions import connection, cursor
```

La signature de la fonction **[afficher_infos]** était :

```
def afficher_infos(curseur: MySQLCursor):
```

Elle devient :

```
def afficher_infos(curseur: cursor)
```

La signature de la fonction **[execute_list_of_commands]** était :

```
def execute_list_of_commands(connexion: MySQLConnection, sql_commands: list,
                             suivi: bool = False, arrêt: bool = True, with_transaction: bool = True)
```

Elle devient :

```
def execute_list_of_commands(connexion: connection, sql_commands: list,
                             suivi: bool = False, arrêt: bool = True, with_transaction: bool = True):
```

Sinon rien d'autre ne change.

17.4.2 script [pgres_01]

Le script **[pgres_01]** est la copie du script **[mysql_01]** (cf paragraphe |script [mysql-01] : connexion à une base MySQL - 1|). On y fait les modifications suivantes :

Au lieu de :

```
1.  # import du module mysql.connector
2.  from mysql.connector import connect, DatabaseError, InterfaceError
```

on écrit :

```
1.  # import du module psycopg2
2.  from psycopg2 import connect, DatabaseError, InterfaceError
```

Le reste ne change pas. Les résultats sont les mêmes qu'avec MySQL.

17.4.3 script [pgres_02]

Le script **[pgres_02]** est la copie du script **[mysql_02]** (cf paragraphe |script [mysql-02] : connexion à une base MySQL - 2|). On y fait les modifications suivantes :

Au lieu de :

```
1.  # import du module mysql.connector
2.  from mysql.connector import DatabaseError, InterfaceError, connect
```

on écrit :

```
1.  # import du module psycopg2
2.  from psycopg2 import DatabaseError, InterfaceError, connect
```

Les résultats ne sont pas les mêmes que ceux du script **[mysql_02]** :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/postgresql/pgres_02.py
2.  Connexion MySQL réussie à la base database=dbpersonnes, host=localhost sous l'identité user=admpersonnes,
    passwd=nobody
3.  Déconnexion MySQL réussie
4.
5.  Process finished with exit code 0
```

Le script **[pgres_02]** est le suivant :

```
1.  # import du module mysql.connector
2.  from psycopg2 import DatabaseError, InterfaceError, connect
3.
4.
5.  # ------------------------------------------------------------------------------
6.  def connexion(host: str, database: str, login: str, pwd: str):
7.      # connecte puis déconnecte (login,pwd) de la base [database] du serveur [host]
8.      # lance l'exception DatabaseError si problème
9.      connexion = None
10.     try:
11.         # connexion
12.         connexion = connect(host=host, user=login, password=pwd, database=database)
13.         print(
14.             f"Connexion réussie à la base database={database}, host={host} sous l'identité user={login}, passwd={pwd}")
15.     finally:
16.         # on ferme la connexion si elle a été ouverte
17.         if connexion:
18.             connexion.close()
19.             print("Déconnexion réussie\n")
20.
21.
22. # ------------------------------------------- main
23. # identifiants de la connexion
24. USER = "admpersonnes"
25. PASSWD = "nobody"
26. HOST = "localhost"
27. DATABASE = "dbpersonnes"
28.
29. # connexion d'un utilisateur existant
30. try:
31.     connexion(host=HOST, login=USER, pwd=PASSWD, database=DATABASE)
32. except (InterfaceError, DatabaseError) as erreur:
33.     # on affiche l'erreur
34.     print(erreur)
35.
36. # connexion d'un utilisateur inexistant
37. try:
38.     connexion(host=HOST, login="xx", pwd="yy", database=DATABASE)
39. except (InterfaceError, DatabaseError) as erreur:
40.     # on affiche l'erreur
41.     print(erreur)
```

Alors que les lignes 36-41 auraient dû afficher un message d'erreur indiquant que la connexion au SGBD avait échoué, rien n'est affiché. En fait lorsqu'on creuse la question, on voit qu'on passe bien dans le **[except]** des lignes 35-37 mais que la variable **[erreur]** vaut **[None]**. Ceci avec la versions 2.8.4 du connecteur **[psycopg2]**.

On peut contourner ce problème en écrivant un message générique mais moins précis :

```
1.  # connexion d'un utilisateur inexistant
2.  try:
3.      connexion(host=HOST, login="xx", pwd="yy", database=DATABASE)
4.  except (InterfaceError, DatabaseError) as erreur:
5.      # on affiche l'erreur
6.      print(f"Erreur de connexion à la base [{DATABASE}] par l'utilisateur [xx/yy]")
```

Les résultats sont alors les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/postgresql/pgres_02.py
2.  Connexion réussie à la base database=dbpersonnes, host=localhost sous l'identité user=admpersonnes,
    passwd=nobody
```

```
3.  Déconnexion réussie
4.
5.  Erreur de connexion à la base [dbpersonnes] par l'utilisateur [xx/yy]
6.
7.  Process finished with exit code 0
```

17.4.4 script [pgres_03]

Le script **[pgres_03]** est la copie du script **[mysql_03]** (cf paragraphe |script [mysql-03] : création d'une table MySQL|). On y fait les modifications suivantes :

Au lieu de :

```
1.  from mysql.connector import DatabaseError, InterfaceError, connect
2.  from mysql.connector.connection import MySQLConnection
```

on écrit :

```
1.  from psycopg2 import DatabaseError, InterfaceError, connect
2.  from psycopg2.extensions import connection
```

Par ailleurs, la signature de la fonction **[execute_sql]** qui était :

```
def execute_sql(connexion: MySQLConnection, update: str):
```

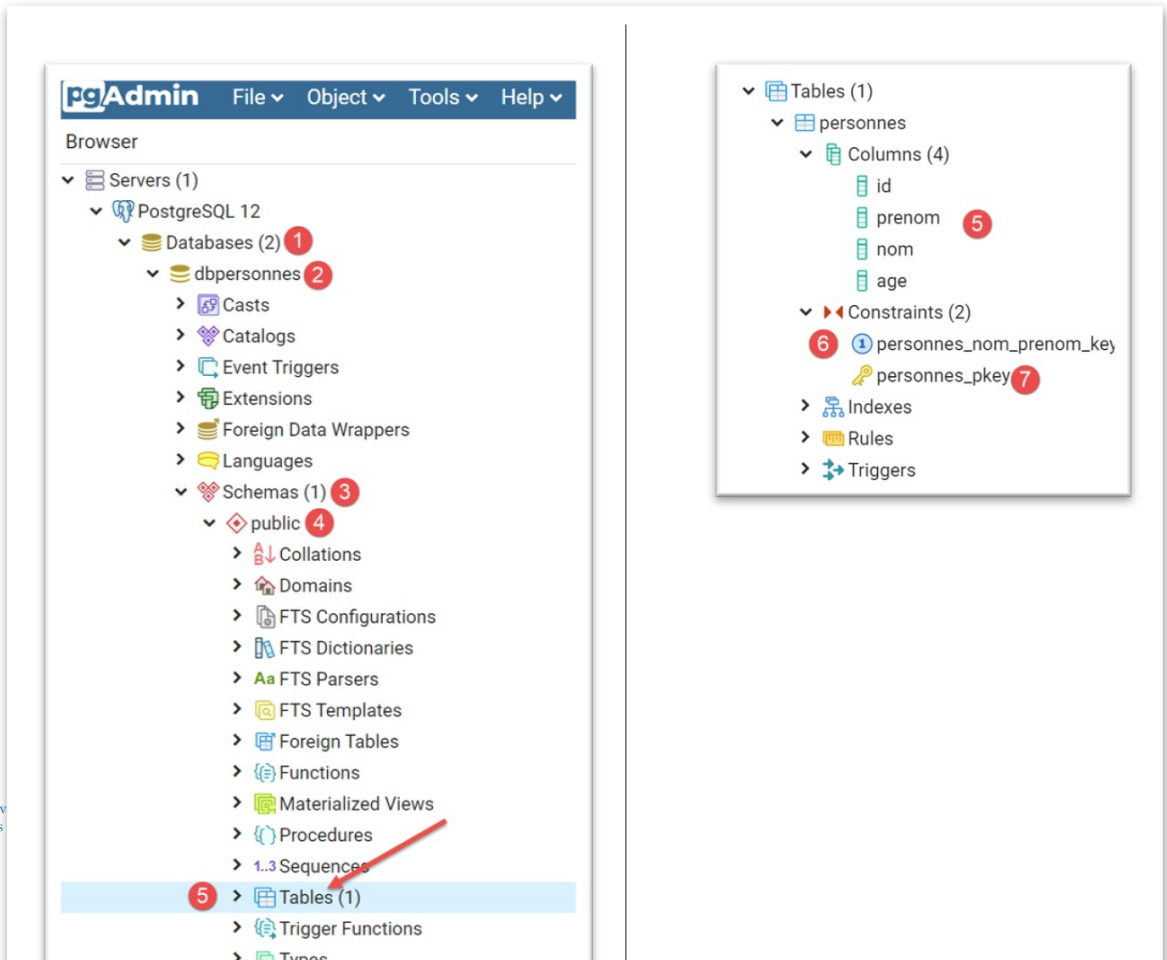
devient :

```
def execute_sql(connexion: connection, update: str):
```

Le reste ne change pas. Le résultat est le suivant :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/postgresql/pgres_03.py
2.  create table personnes (id int PRIMARY KEY, prenom varchar(30) NOT NULL, nom varchar(30) NOT NULL, age
    integer NOT NULL, unique(nom,prenom))  : requête réussie
3.
4.  Process finished with exit code 0
```

On peut vérifier la présence de la table **[personnes]** avec l'outil d'administration **[pgAdmin]** :



17.4.5 script [pgres_04]

Le script **[pgres_04]** est une copie du script **[mysql_04]** (cf paragraphe |script [mysql-04] : exécution d'un fichier d'ordres SQL|). Il utilise le module **[pgres_module]** :

```
# on récupère la configuration de l'application
import config_04

config = config_04.configure()

# le syspath est configuré - on peut faire les imports
import sys
from pgres_module import execute_file_of_commands
from psycopg2 import connect, DatabaseError, InterfaceError
```

Le reste ne change pas.

On crée une configuration **[pgres pgres-04 without_transaction]** comme il a été fait au paragraphe |script [mysql-04] : exécution d'un fichier d'ordres SQL|. On crée de même une configuration **[pgres pgres-04 with_transaction]**.

L'exécution de la configuration **[pgres pgres-04 without_transaction]** donne les résultats suivants :

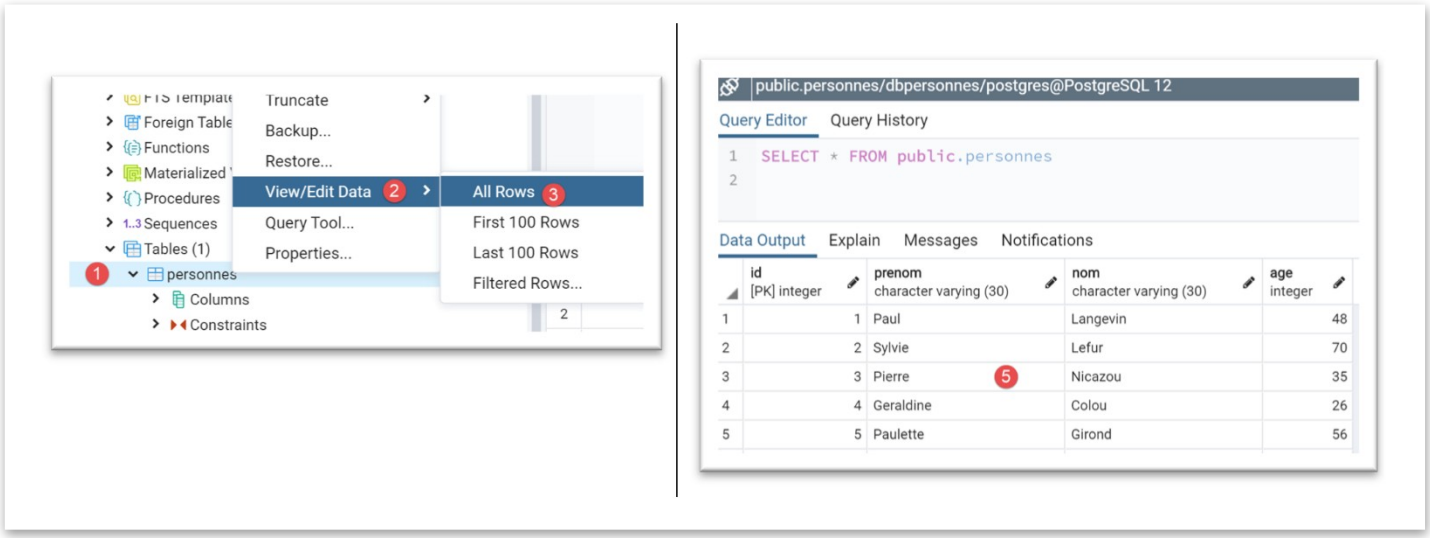
```
C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/postgresql/pgres_04.py false
----------------------------------------------------------------------
Exécution du fichier SQL C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\databases\postgresql/data/commandes.sql sans transaction
----------------------------------------------------------------------
[drop table if exists personnes] : Exécution réussie
nombre de lignes modifiées : -1
[create table personnes (id int primary key, prenom varchar(30) not null, nom varchar(30) not null, age integer not null, unique (nom,prenom))] : Exécution réussie
nombre de lignes modifiées : -1
[insert into personnes(id, prenom, nom, age) values(1, 'Paul','Langevin',48)] : Exécution réussie
nombre de lignes modifiées : 1
[insert into personnes(id, prenom, nom, age) values (2, 'Sylvie','Lefur',70)] : Exécution réussie
nombre de lignes modifiées : 1
[select prenom, nom, age from personnes] : Exécution réussie
prenom, nom, age,
**********
('Paul', 'Langevin', 48)
('Sylvie', 'Lefur', 70)
**********
xx : Erreur (ERREUR:  erreur de syntaxe sur ou près de « xx »
LINE 1: xx
        ^
)
[insert into personnes(id, prenom, nom, age) values (3, 'Pierre','Nicazou',35)] : Exécution réussie
nombre de lignes modifiées : 1
[insert into personnes(id, prenom, nom, age) values (4, 'Geraldine','Colou',26)] : Exécution réussie
nombre de lignes modifiées : 1
[insert into personnes(id, prenom, nom, age) values (5, 'Paulette','Girond',56)] : Exécution réussie
nombre de lignes modifiées : 1
[select prenom, nom, age from personnes] : Exécution réussie
prenom, nom, age,
**********
('Paul', 'Langevin', 48)
('Sylvie', 'Lefur', 70)
('Pierre', 'Nicazou', 35)
('Geraldine', 'Colou', 26)
('Paulette', 'Girond', 56)
**********
[select nom,prenom from personnes order by nom asc, prenom desc] : Exécution réussie
nom, prenom,
**********
('Colou', 'Geraldine')
('Girond', 'Paulette')
('Langevin', 'Paul')
('Lefur', 'Sylvie')
('Nicazou', 'Pierre')
**********
[select nom,prenom,age from personnes where age between 20 and 40 order by age desc, nom asc, prenom asc] : Exécution réussie
nom, prenom, age,
**********
```

```
('Nicazou', 'Pierre', 35)
('Colou', 'Geraldine', 26)
**********
[insert into personnes(id, prenom, nom, age) values(6, 'Josette','Bruneau',46)] : Exécution réussie
nombre de lignes modifiées : 1
[update personnes set age=47 where nom='Bruneau'] : Exécution réussie
nombre de lignes modifiées : 1
[select nom,prenom,age from personnes where nom='Bruneau'] : Exécution réussie
nom, prenom, age,
**********
('Bruneau', 'Josette', 47)
**********
[delete from personnes where nom='Bruneau'] : Exécution réussie
nombre de lignes modifiées : 1
[select nom,prenom,age from personnes where nom='Bruneau'] : Exécution réussie
nom, prenom, age,
**********
**********
------------------------------------------------------------------------
Exécution terminée
------------------------------------------------------------------------
Il y a eu 1 erreur(s)
xx : Erreur (ERREUR:  erreur de syntaxe sur ou près de « xx »
LINE 1: xx
        ^
)

Process finished with exit code 0
```

- ligne 5 : on a dû modifier la commande de suppression de la table **[personnes]**. Contrairement au connecteur de MySQL le connecteur de PostgreSQL lance une exception si la table à supprimer n'existe pas. La commande **[drop table]** a une variante **[drop table if exists]** qui ne lance pas d'exception si la table n'existe pas. Nous l'avons utilisée ici. C'est un exemple où deux SGBD ne se comportent pas de la même façon dans des situations analogues ;

La table **[personnes]** dans l'outil **[pgAdmin]** est la suivante :

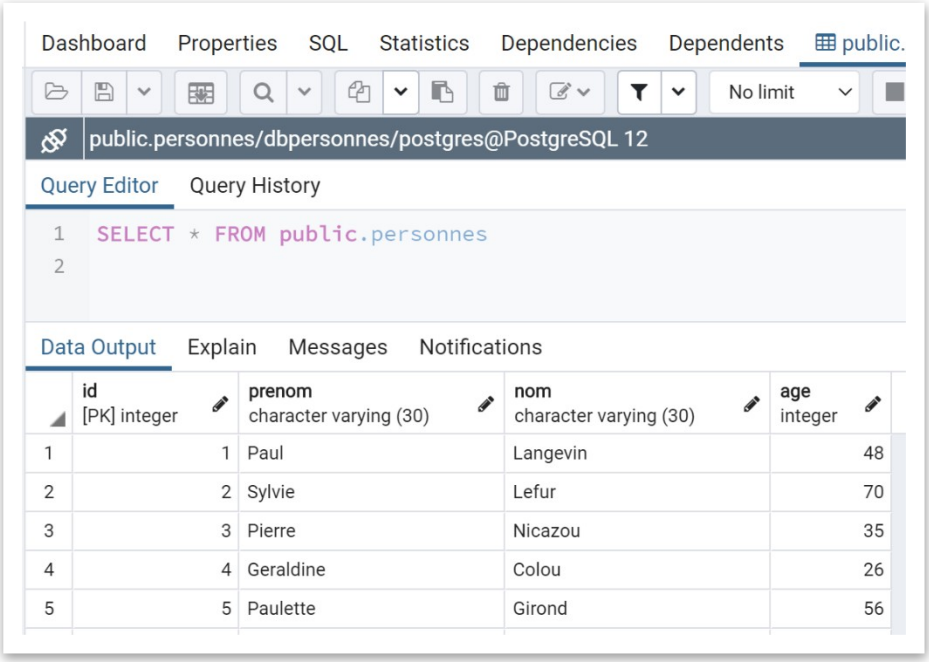


L'exécution de la configuration **[pgres pgres_04 with_transaction]** donne les résultats suivants :

```
C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/postgresql/pgres_04.py true
------------------------------------------------------------------------
Exécution du fichier SQL C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\databases\postgresql/
data/commandes.sql avec transaction
------------------------------------------------------------------------
[drop table if exists personnes] : Exécution réussie
nombre de lignes modifiées : -1
[create table personnes (id int primary key, prenom varchar(30) not null, nom varchar(30) not null, age
integer not null, unique (nom,prenom))] : Exécution réussie
nombre de lignes modifiées : -1
[insert into personnes(id, prenom, nom, age) values(1, 'Paul','Langevin',48)] : Exécution réussie
nombre de lignes modifiées : 1
```

```
11. [insert into personnes(id, prenom, nom, age) values (2, 'Sylvie','Lefur',70)] : Exécution réussie
12. nombre de lignes modifiées : 1
13. [select prenom, nom, age from personnes] : Exécution réussie
14. prenom, nom, age,
15. *****************
16. ('Paul', 'Langevin', 48)
17. ('Sylvie', 'Lefur', 70)
18. *****************
19. xx : Erreur (ERREUR:  erreur de syntaxe sur ou près de « xx »
20. LINE 1: xx
21.         ^
22. )
23. ------------------------------------------------------------------------
24. Exécution terminée
25. ------------------------------------------------------------------------
26. Il y a eu 1 erreur(s)
27. xx : Erreur (ERREUR:  erreur de syntaxe sur ou près de « xx »
28. LINE 1: xx
29.         ^
30. )
31.
32. Process finished with exit code 0
```

La table **[personnes]** dans l'outil **[pgAdmin]** est la suivante :



Ici, le résultat est différent de celui obtenu avec MySQL. Si on exécute les scripts dans les mêmes conditions, ç-à-d après exécution du script sans transaction, on a les résultats suivants :

- avec MySQL, la table **[personnes]** est vide ;
- avec PostgreSQL, la table **[personnes]** ne l'est pas ;

La différence repose sur les façons différentes qu'ont ces deux SGBD de défaire la transaction :

- MySQL ne défait pas les ordres **[drop table]** et **[create table]**. On se retrouve avec une table **[personnes]** vide ;
- PostgreSQL défait les ordres **[drop table]** et **[create table]**. On retrouve la table dans l'état où elle était avant l'exécution du script avec transaction ;

17.4.6 script [pgres_05]

Le script **[pgres_05]** est une copie du script **[mysql_05]** (cf paragraphe |script **[mysql-05]** : utilisation de requêtes paramétrées|). Le script est modifié de la façon suivante :

Au lieu de :

```
1.  # imports
```

```
2.  from mysql.connector import connect, DatabaseError, InterfaceError
```

on écrit :

```
1.  # imports
2.  from psycopg2 import connect, DatabaseError, InterfaceError
```

Le reste ne change pas.

Les résultats obtenus dans **[pgAdmin]** sont les suivants :

Data Output | Explain | Messages | Notifications

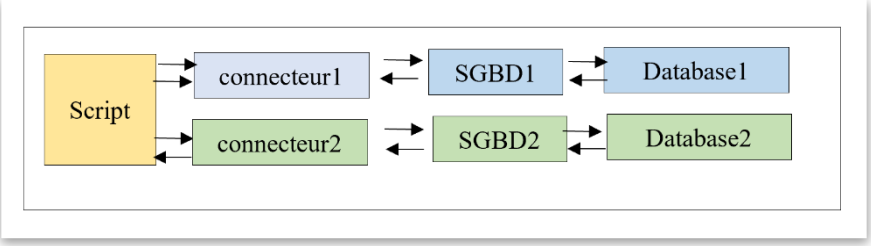
	id [PK] integer	prenom character varying (30)	nom character varying (30)	age integer
1	0	p00	n00	10
2	1	p01	n01	11
3	2	p02	n02	12
4	3	p03	n03	13
5	4	p04	n04	14
6	40	Y'éna	d'Aboot	18
7	100	p10	n10	20
8	101	p11	n11	21
9	102	p12	n12	22
10	103	p13	n13	23
11	104	p14	n14	24
12	200	F'ilhem	d'Aboot	34

17.5 Conclusion

Le portage des scripts MySQL vers des scripts PostgreSQL s'est réalisé plutôt facilement. C'est une exception. Les deux SGBD ne supportent pas les mêmes règles de nommage des objets SQL (bases, tables, colonnes, contraintes, types des données…), ont des extensions SQL incompatibles… Pour assurer un portage simple, il faut s'en tenir dans les deux cas à la norme SQL sans essayer d'utiliser les extensions propriétaires des SGBD. Cela se fait alors aux dépens des performances.

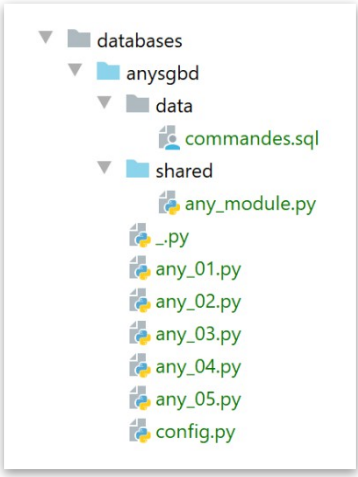
18 Ecrire du code indépendant du SGBD

Nous avons vu précédemment que dans certains cas il était possible de migrer simplement du code Python écrit pour le SGBD MySQL vers du code écrit pour le SGBD PostgreSQL. Dans ce chapitre, nous montrons comment systématiser cette approche. L'architecture proposée devient la suivante :



On souhaite que le choix du connecteur et donc du SGBD se fasse par configuration et ne nécessite pas de réécriture du script. On rappelle que cela n'est possible que dans les cas où le script n'utilise pas d'extensions propriétaires du SGBD.

L'arborescence des scripts sera la suivante :



Les scripts **[any_xx]** reprennent les scripts déjà étudiés pour les SGBD MySQL et PostgreSQL. Nous n'allons pas tous les reprendre. Nous allons nous concentrer sur le script **[any_04]** qui est le plus complexe. On rappelle que ce script exécute les commandes SQL du fichier **[data/commandes.sql]** suivant :

```
1.  # suppression de la table [personnes]
2.  drop table if exists personnes
3.  # création de la table personnes
4.  create table personnes (id int primary key, prenom varchar(30) not null, nom varchar(30) not null, age
    integer not null, unique (nom,prenom))
5.  # insertion de deux personnes
6.  insert into personnes(id, prenom, nom, age) values(1, 'Paul','Langevin',48)
7.  insert into personnes(id, prenom, nom, age) values (2, 'Sylvie','Lefur',70)
8.  # affichage de la table
9.  select prenom, nom, age from personnes
10. # erreur volontaire
11. xx
12. # insertion de trois personnes
13. insert into personnes(id, prenom, nom, age) values (3, 'Pierre','Nicazou',35)
14. insert into personnes(id, prenom, nom, age) values (4, 'Geraldine','Colou',26)
15. insert into personnes(id, prenom, nom, age) values (5, 'Paulette','Girond',56)
16. # affichage de la table
17. select prenom, nom, age from personnes
18. # liste des personnes par ordre alphabétique des noms et à nom égal par ordre alphabétique des prénoms
19. select nom,prenom from personnes order by nom asc, prenom desc
20. # liste des personnes ayant un âge dans l'intervalle [20,40] par ordre décroissant de l'âge
21. # puis à âge égal par ordre alphabétique des noms et à nom égal par ordre alphabétique des prénoms
22. select nom,prenom,age from personnes where age between 20 and 40 order by age desc, nom asc, prenom asc
23. # insertion de mme Bruneau
24. insert into personnes(id, prenom, nom, age) values(6, 'Josette','Bruneau',46)
25. # mise à jour de son âge
26. update personnes set age=47 where nom='Bruneau'
```

```
27. # liste des personnes ayant Bruneau pour nom
28. select nom,prenom,age from personnes where nom='Bruneau'
29. # suppression de Mme Bruneau
30. delete from personnes where nom='Bruneau'
31. # liste des personnes ayant Bruneau pour nom
32. select nom,prenom,age from personnes where nom='Bruneau'
```

Nous avons modifié la ligne 2 pour que la commande ait le même comportement pour les SGBD MySQL et PostgreSQL si la table **[personnes]** n'existe pas.

Le script **[any_04]** est configuré par le script **[config.py]** suivant :

```
1.  def configure():
2.      import os
3.
4.      # chemin absolu du dossier de ce script
5.      script_dir = os.path.dirname(os.path.abspath(__file__))
6.
7.      # configuration des dossiers du syspath
8.      absolute_dependencies = [
9.          # dossiers locaux
10.         f"{script_dir}/shared",
11.     ]
12.
13.     # fixation du syspath
14.     from myutils import set_syspath
15.     set_syspath(absolute_dependencies)
16.
17.     # configuration de l'application
18.     config = {
19.         # sgbd gérés
20.         "sgbds": {
21.             "mysql": {
22.                 # connecteur du sgbd
23.                 "sgbd_connector": "mysql.connector",
24.                 # nom du module rassemblant des fonctions de gestion du sgbd
25.                 "sgbd_fonctions": "any_module",
26.                 # identifiants de la connexion
27.                 "user": "admpersonnes",
28.                 "password": "nobody",
29.                 "host": "localhost",
30.                 "database": "dbpersonnes"
31.             },
32.             "postgresql": {
33.                 # connecteur du sgbd
34.                 "sgbd_connector": "psycopg2",
35.                 # nom du module rassemblant des fonctions de gestion du sgbd
36.                 "sgbd_fonctions": "any_module",
37.                 # identifiants de la connexion
38.                 "user": "admpersonnes",
39.                 "password": "nobody",
40.                 "host": "localhost",
41.                 "database": "dbpersonnes"
42.             }
43.         },
44.         # fichier de commandes SQL
45.         "commands_filename": f"{script_dir}/data/commandes.sql"
46.     }
47.     # syspath de l'application
48.     from myutils import set_syspath
49.     set_syspath(absolute_dependencies)
50.
51.     # on rend la config
52.     return config
```

La nouveauté réside dans les lignes 18-43 :

- ligne 20 : **[sgbds]** est un dictionnaire avec deux clés **[mysql]** ligne 21 et **[postgresql]** ligne 32 ;
- la valeur associée à ces clés est un dictionnaire donnant les éléments permettant la connexion à un SGBD :
- lignes 21-32 : les éléments d'une connexion au SGBD MySQL ;
  - ligne 23 : le connecteur Python à utiliser ;
  - ligne 25 : le module contenant des fonctions partagées ;
  - lignes 26-30 : les identifiants de la connexion ;
- lignes 32-41 : les mêmes éléments pour une connexion au SGBD PostgreSQL ;

Le script **[any_04]** qui exécute le fichier de commandes SQL **[data/commandes.sql]** est le suivant :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on peut faire les imports
7.  import importlib
8.  import sys
9.
10. # vérification de la syntaxe de l'appel
11. # argv[0] sgbd_name true / false
12. # il faut 3 paramètres
13. args = sys.argv
14. erreur = len(args) != 3
15. if not erreur:
16.     # on récupère les deux paramètres qui nous intéressent
17.     sgbd_name = args[1].lower()
18.     with_transaction = args[2].lower()
19.     # vérification des deux paramètres
20.     erreur = (with_transaction != "true" and with_transaction != "false") \
21.              or sgbd_name not in config["sgbds"].keys()
22. # erreur ?
23. if erreur:
24.     print(f"syntaxe : {args[0]} (1) sgbd_name (2) true / false")
25.     sys.exit()
26.
27. # configuration du sgbd_name
28. sgbd_config = config["sgbds"][sgbd_name]
29. # connecteur du sgbd_name
30. sgbd_connector = importlib.import_module(sgbd_config["sgbd_connector"])
31. # bibliothèque de fonctions
32. lib = importlib.import_module(sgbd_config["sgbd_fonctions"])
33.
34.
35. # calcul d'un texte à afficher
36. with_transaction = with_transaction == "true"
37. if with_transaction:
38.     texte = "avec transaction"
39. else:
40.     texte = "sans transaction"
41.
42. # affichage
43. commands_filename=config['commands_filename']
44. print("------------------------------------------------------------------------")
45. print(f"Exécution du fichier SQL {commands_filename} {texte}")
46. print("------------------------------------------------------------------------")
47.
48. # exécution des ordres SQL du fichier
49. connexion = None
50. try:
51.     # connexion
52.     connexion = sgbd_connector.connect(
53.         host=sgbd_config['host'],
54.         user=sgbd_config['user'],
55.         password=sgbd_config['password'],
56.         database=sgbd_config['database'])
57.     # exécution du fichier des commandes SQL
58.     erreurs = lib.execute_file_of_commands(sgbd_connector, connexion, commands_filename, suivi=True, arrêt=False,
59.                                            with_transaction=with_transaction)
60. except (sgbd_connector.InterfaceError, sgbd_connector.DatabaseError) as erreur:
61.     print(f"L'erreur suivante s'est produite : {erreur}")
62. finally:
63.     # fermeture de la connexion
64.     if connexion:
65.         connexion.close()
66.
67. # affichage nombre d'erreurs
68. print("------------------------------------------------------------------------")
69. print(f"Exécution terminée")
70. print("------------------------------------------------------------------------")
71. print(f"Il y a eu {len(erreurs)} erreur(s)")
72. # affichage des erreurs
73. for erreur in erreurs:
74.     print(erreur)
```

Commentaires

- lignes 1-4 : on récupère la configuration **[config]** de l'application ;
- lignes 10-21 : le script s'appelle avec deux paramètres **[sgbd_name with_transaction]** :
  - **[sgbd_name]** : le nom du SGBD à utiliser ;

    - **[with_transaction]** : True si on veut exécuter le fichier de commandes SQL au sein d'une transaction, False sinon ;
- lignes 10-25 : les paramètres sont récupérés et vérifiés ;
- ligne 28 : la configuration du SGBD choisi ;
- ligne 30 : on importe le connecteur du SGBD choisi. On utilise pour cela la bibliothèque **[importlib]** (ligne 7) qui permet d'importer un module dont le nom est dans une variable. Le résultat de l'opération **[importlib.import_module]** est un module. Ainsi après la ligne 30, tout se passe comme si l'instruction exécutée avait été :

```
import sgbd_connector
```

  Ceci va nous permettre d'écrire ligne 52 **[sgbd_connector.connect]** où on utilise la fonction **[connect]** du module **[sgbd_connector]**. Il faut se rappeler ici que **[sgbd_connector]** est soit **[mysql.connector]** ou **[psycopg2]**. Ces deux modules ont la fonction **[connect]**. De même, ligne 60, on peut écrire **[sgbd_connector.InterfaceError, sgbd_connector.DatabaseError]**.
- ligne 32 : on importe le module des fonctions utilisées par le script ;
- ligne 58 : on exécute la fonction **[execute_file_of_commands]** du module des fonctions utilisées par le script. Par rapport aux versions précédentes, la signature de cette fonction a un paramètre de plus, le premier. On passe à la fonction le connecteur Python **[sgbd_connector]** qu'elle doit utiliser ;
- en-dehors de ces points, le script **[any_04]** reste ce qu'il était dans les versions précédentes ;

La bibliothèque de fonctions **[any_module]** est la suivante :

```
1.  # ---------------------------------------------------------------------------
2.
3.  def afficher_infos(curseur):
4.      …
5.
6.
7.  # ---------------------------------------------------------------------------
8.  def execute_list_of_commands(sgbd_connector, connexion, sql_commands: list,
9.                               suivi: bool = False, arrêt: bool = True, with_transaction: bool = True):
10.     …
11.
12.     # initialisations
13.     curseur = None
14.     connexion.autocommit = not with_transaction
15.     erreurs = []
16.     try:
17.         # on demande un curseur
18.         curseur = connexion.cursor()
19.         # exécution des sql_commands SQL contenues dans sql_commands
20.         # on les exécute une à une
21.         for command in sql_commands:
22.             # on élimine les blancs de début et de fin de la commande courante
23.             command = command.strip()
24.             # a-t-on une commande vide ou un commentaire ? Si oui, on passe à la commande suivante
25.             if command == '' or command[0] == "#":
26.                 continue
27.             # exécution de la commande courante
28.             error = None
29.             try:
30.                 curseur.execute(command)
31.             except (sgbd_connector.InterfaceError, sgbd_connector.DatabaseError) as erreur:
32.                 error = erreur
33.             # y-a-t-il eu une erreur ?
34.             …
35.
36.
37. # ---------------------------------------------------------------------------
38. def execute_file_of_commands(sgbd_connector, connexion, sql_filename: str,
39.                               suivi: bool = False, arrêt: bool = True, with_transaction: bool = True):
40.     …
41.
42.     # exploitation du fichier SQL
43.     try:
44.         # ouverture du fichier en lecture
45.         file = open(sql_filename, "r")
46.         # exploitation
47.         return execute_list_of_commands(sgbd_connector, connexion, file.readlines(), suivi, arrêt, with_transaction)
48.     except BaseException as erreur:
49.         # on rend un tableau d'erreurs
50.         return [f"Le fichier {sql_filename} n'a pu être être exploité : {erreur}"]
51.     finally:
52.         pass
```

Le paramètre **[sgbd_connector]** a été utilisé ligne 31 pour préciser le type des exceptions interceptées.

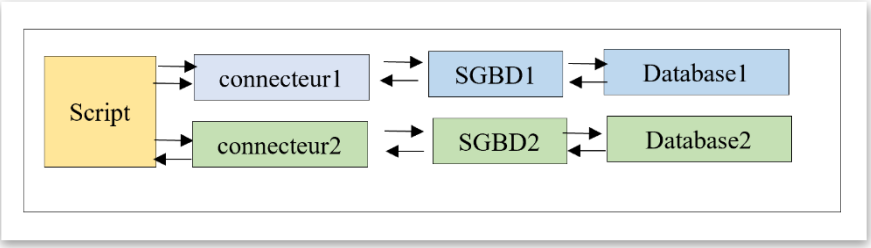
L'exécution du script **[any_04]** avec les paramètres **[mysql false]** donne les résultats suivants :

```
C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/anysgbd/any_04.py mysql false
----------------------------------------------------------------------
Exécution du fichier SQL C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\databases\anysgbd/data/commandes.sql sans transaction
----------------------------------------------------------------------
[drop table if exists personnes] : Exécution réussie
nombre de lignes modifiées : 0
[create table personnes (id int primary key, prenom varchar(30) not null, nom varchar(30) not null, age integer not null, unique (nom,prenom))] : Exécution réussie
nombre de lignes modifiées : 0
[insert into personnes(id, prenom, nom, age) values(1, 'Paul','Langevin',48)] : Exécution réussie
nombre de lignes modifiées : 1
[insert into personnes(id, prenom, nom, age) values (2, 'Sylvie','Lefur',70)] : Exécution réussie
nombre de lignes modifiées : 1
[select prenom, nom, age from personnes] : Exécution réussie
prenom, nom, age,
**********
('Paul', 'Langevin', 48)
('Sylvie', 'Lefur', 70)
**********
xx : Erreur (1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'xx' at line 1)
[insert into personnes(id, prenom, nom, age) values (3, 'Pierre','Nicazou',35)] : Exécution réussie
nombre de lignes modifiées : 1
[insert into personnes(id, prenom, nom, age) values (4, 'Geraldine','Colou',26)] : Exécution réussie
nombre de lignes modifiées : 1
[insert into personnes(id, prenom, nom, age) values (5, 'Paulette','Girond',56)] : Exécution réussie
nombre de lignes modifiées : 1
[select prenom, nom, age from personnes] : Exécution réussie
prenom, nom, age,
**********
('Paul', 'Langevin', 48)
('Sylvie', 'Lefur', 70)
('Pierre', 'Nicazou', 35)
('Geraldine', 'Colou', 26)
('Paulette', 'Girond', 56)
**********
[select nom,prenom from personnes order by nom asc, prenom desc] : Exécution réussie
nom, prenom,
**********
('Colou', 'Geraldine')
('Girond', 'Paulette')
('Langevin', 'Paul')
('Lefur', 'Sylvie')
('Nicazou', 'Pierre')
**********
[select nom,prenom,age from personnes where age between 20 and 40 order by age desc, nom asc, prenom asc] : Exécution réussie
nom, prenom, age,
**********
('Nicazou', 'Pierre', 35)
('Colou', 'Geraldine', 26)
**********
[insert into personnes(id, prenom, nom, age) values(6, 'Josette','Bruneau',46)] : Exécution réussie
nombre de lignes modifiées : 1
[update personnes set age=47 where nom='Bruneau'] : Exécution réussie
nombre de lignes modifiées : 1
[select nom,prenom,age from personnes where nom='Bruneau'] : Exécution réussie
nom, prenom, age,
**********
('Bruneau', 'Josette', 47)
**********
[delete from personnes where nom='Bruneau'] : Exécution réussie
nombre de lignes modifiées : 1
[select nom,prenom,age from personnes where nom='Bruneau'] : Exécution réussie
nom, prenom, age,
**********
**********
----------------------------------------------------------------------
Exécution terminée
----------------------------------------------------------------------
Il y a eu 1 erreur(s)
xx : Erreur (1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'xx' at line 1)

Process finished with exit code 0
```

19 Utilisation de l'ORM SQLALCHEMY

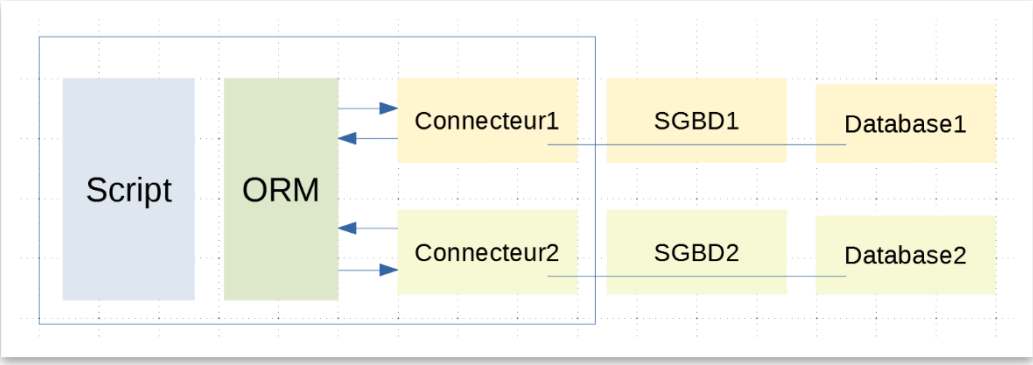
Le chapitre précédent a montré que dans certains cas on pouvait écrire du code indépendant du SGBD utilisé avec l'architecture suivante :



Dans ce chapitre nous allons utiliser l'ORM (Object Relational Mapper) **[sqlalchemy]** pour accéder aux SGBD de façon uniforme quelque soit le SGBD utilisé. Un ORM permet deux choses :

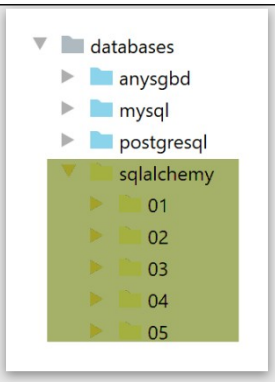
- il permet à un script de dialoguer avec le SGBD sans émettre d'ordres SQL ;
- il masque au script les particularités de chaque SGBD ;

L'architecture devient la suivante :



Le script est désormais séparé des connecteurs par l'ORM. Il dialogue avec l'ORM avec des classes et des méthodes. Il n'exécute pas de code SQL. C'est l'ORM qui le fait avec les connecteurs auxquels il est relié. Il cache au script les particularités de ces connecteurs. Aussi le code du script est-il insensible à un changement de connecteur (donc du SGBD) ;

L'arborescence des scripts étudiés sera la suivante :

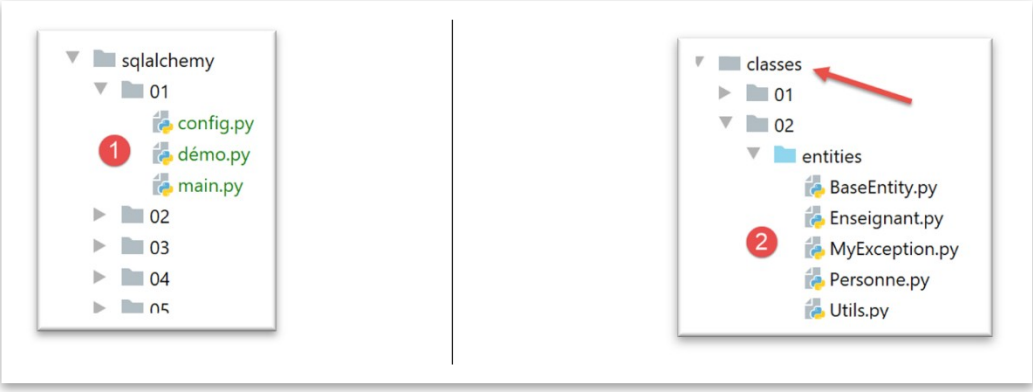


19.1 Installation de l'ORM [sqlalchemy]

L'ORM **[sqlalchemy]** vient sous la forme d'un package python qu'il faut installer dans un terminal Python :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\databases\sqlalchemy>pip install sqlalchemy
2.  Collecting sqlalchemy
3.    Downloading SQLAlchemy-1.3.18-cp38-cp38-win_amd64.whl (1.2 MB)
4.       || 1.2 MB 3.3 MB/s
5.  Installing collected packages: sqlalchemy
6.  Successfully installed sqlalchemy-1.3.18
```

19.2 Scripts 01 : les bases



- en **[1]**, les scripts qui vont être étudiés. Ces scripts vont utiliser les classes de **[2]** : BaseEntity, MyException, Personne, Utils ;

19.2.1 Configuration

Le fichier **[config]** configure l'application de la façon suivante :

```
1.  def configure():
2.      # root_dir
3.      # chemin absolu référence des chemins relatifs de la configuration
4.      root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
5.      # chemins absolus des dépendances
6.      absolute_dependencies = [
7.          # BaseEntity, MyException, Personne, Utils
8.          f"{root_dir}/classes/02/entities",
9.      ]
10.
11.     # on fixe le syspath
12.     from myutils import set_syspath
13.     set_syspath(absolute_dependencies)
14.
15.     # configuration des classes
16.     from Personne import Personne
17.     Personne.excluded_keys = ['_sa_instance_state']
18.
19.     # on rend la config
20.     return {}
```

Commentaires

- ligne 8 : on met dans le Python Path le dossier contenant les classes **[BaseEntity, MyException, Personne, Utils]** ;
- lignes 12-13 : on fixe le Python Path de l'application ;
- lignes 16-17 : on se rappelle peut-être que la classe |BaseEntity| a un attribut de classe nommé **[excluded_keys]**. Cet attribut est une liste dans laquelle on met les propriétés de la classe qu'on ne veut pas voir apparaître dans le dictionnaire de celle-ci (fonction asdict). Ici on exclut la propriété **[_sa_instance_state]** de l'état de la classe **[Personne]**. On verra bientôt pourquoi ;

19.2.2 Script [démo]

Le script **[démo]** montre une première utilisation de l'ORM **[sqlalchemy]** :

```
1.  # on récupère la configuration de l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # imports
7.  from sqlalchemy import Table, Column, Integer, String, MetaData, UniqueConstraint
8.  from sqlalchemy.orm import mapper
9.
10. from Personne import Personne
11.
12. # metadata
```

```
13. metadata = MetaData()
14.
15. # la table
16. personnes_table = Table("personnes", metadata,
17.                        Column('id', Integer, primary_key=True),
18.                        Column('prenom', String(30), nullable=False),
19.                        Column("nom", String(30), nullable=False),
20.                        Column("age", Integer, nullable=False),
21.                        UniqueConstraint('nom', 'prenom', name='uix_1')
22.                        )
23. # la classe Personne avant le mapping
24. personne1 = Personne().fromdict({"id": 67, "prénom": "x", "nom": "y", "âge": 10})
25. print(f"personne1={personne1.__dict__}")
26.
27. # le mapping
28. mapper(Personne, personnes_table, properties={
29.     'id': personnes_table.c.id,
30.     'prénom': personnes_table.c.prenom,
31.     'nom': personnes_table.c.nom,
32.     'âge': personnes_table.c.age
33. })
34.
35. # personne1 n'a pas été modifiée
36. print(f"personne1={personne1.__dict__}")
37. # la classe Personne a elle été modifiée - elle a été "enrichie"
38. personne2 = Personne().fromdict({"id": 68, "prénom": "x1", "nom": "y1", "âge": 11})
39. print(f"personne2={personne2.__dict__}")
```

Commentaires

- lignes 1-4 : on configure l'application ;
- lignes 6-10 : on importe les modules nécessaires au script ;
- ligne 13 : **[MetaData]** est une classe de **[sqlalchemy]** ;
- lignes 15-22 : **[Table]** est une classe de **[sqlalchemy]**. Elle permet de décrire une table d'une base de données. Ici, nous allons décrire la table **[personnes]** de la base MySQL **[dbpersonnes]** étudiée au chapitre |MySQL| ;
  - ligne 16 : le 1er paramètre **[personnes]** est le nom de la table décrite ;
  - ligne 16 : le second paramètre **[metadata]** est l'instance **[MetaData]** créée ligne 13 ;
  - lignes 17-22 : chacun des paramètres suivants décrit une colonne de la table avec une syntaxe propre à **[sqlalchemy]** mais proche de la syntaxe SQL ;
  - chaque colonne se décrit avec une instance de la classe **[Column]** de **[sqlalchemy]** ;
    - le 1er paramètre est le nom de la colonne ;
    - le second paramètre est son type ;
    - les paramètres suivants sont des paramètres nommés :
      - ligne 17 : **[primary_key=True]** pour indiquer que la colonne **[id]** est clé primaire de la table **[personnes]** ;
      - ligne 18 : **[nullable=False]** pour indiquer qu'uen colonne doit forcément avoir une valeur lorsqu'une ligne est insérée dans la table ;
  - ligne 21 : enfin la classe **[UniqueConstraint]** permet de décrire une contrainte d'unicité. Ici on indique que les colonnes (nom, prenom) doivent être uniques dans la table. La propriété nommée **[name]** permet de donner un nom à cette constrainte. Ici, il faut distinguer deux cas :
    - on décrit une table existante. Il faut alors chercher le nom de la contrainte dans les propriétés de la table (phpMyAdmin ou pgAdmin) ;
    - on décrit une table que l'on va créer. Alors on met le nom que l'on veut ;
- lignes 23-25 : on crée une personne **[personne1]** et on affiche son dictionnaire **[__dict__]**. On va avoir ici :

```
personne1={'_BaseEntity__id': 67, '_Personne__prénom': 'x', '_Personne__nom': 'y', '_Personne__âge': 10}
```

- lignes 27-33 : on fait un mapping, ç-à-d qu'on crée une correspondance entre la classe **[Personne]** et la table **[personnes]**. C'est essentiellement une correspondance **[propriétés de la classe ←→ colonnes de la table]**. La fonction **[mapper]** accepte ici trois paramètres :
  - ligne 28 : le 1er paramètre est le nom de la classe pour laquelle on fait le mapping ;
  - ligne 28 : le second paramètre est la table à laquelle elle va être associée. Celle-ci est l'objet **[Table]** créé ligne 16 ;
  - ligne 28 : le 3ième paramètre est ici un paramètre nommé **[properties]**. C'est un dictionnaire dans lequel les clés sont les propriétés de la classe mappée et les valeurs les colonnes de la table mappée. Pour désigner la colonne X de la table **[personnes_table]**, on écrit **[personnes_table.c.X]** ;
- lignes 35-36 : on réaffiche la personne **[personne1]** une fois le mapping fait. On constate qu'elle n'a pas changé :

```
personne1={'_BaseEntity__id': 67, '_Personne__prénom': 'x', '_Personne__nom': 'y', '_Personne__âge': 10}
```

- lignes 37-39 : on crée une nouvelle personne **[personne2]** et on l'affiche. On a alors l'affichage suivant :

```
personne2={'_sa_instance_state': <sqlalchemy.orm.state.InstanceState object at 0x00000259A6747FA0>,
'id': 68, 'prénom': 'x1', 'nom': 'y1', 'âge': 11}
```

  On constate que le dictionnaire **[__dict__]** a été profondément modifié :

  - une nouvelle propriété **[_sa_instance_state]** apparaît. On voit que c'est un objet de l'ORM **[sqlalchemy]** ;
  - les autres propriétés ont été débarrassées de leur préfixe qui indiquait à quelle classe elles appartenaient ;

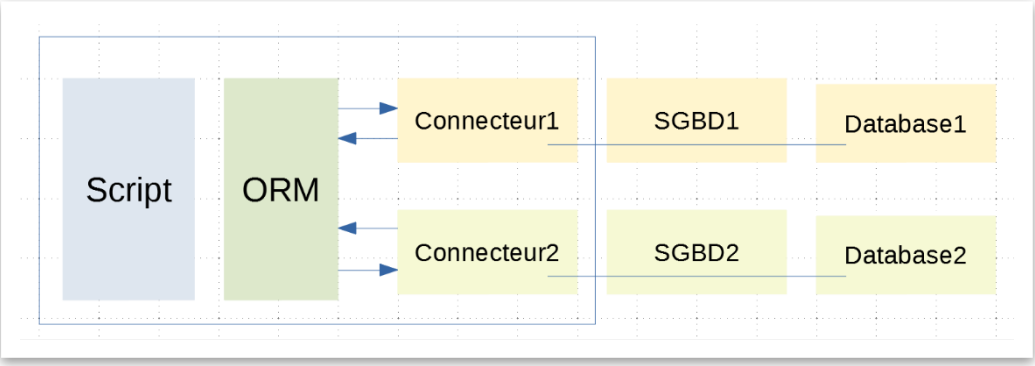
  On peut donc conclure que l'opération de mapping des lignes 27-33 a modifié la classe **[Personne]**.

Lorsqu'on voudra afficher l'état d'un objet **[Personne]**, on ne voudra pas en général de la propriété **[_sa_instance_state]**. Elle n'est là en effet que pour la cuisine interne de **[sqlalchemy]** et en général elle ne nous intéresse pas. C'est pourquoi on a écrit dans le script **[config]** :

```
1.      # configuration des classes
2.      from Personne import Personne
3.      Personne.excluded_keys = ['_sa_instance_state']
```

19.2.3 Le script [main]

Le script **[main]** va manipuler la table **[personnes]** de la base MySQL **[dbpersonnes]** en s'interfaçant avec **[sqlalchemy]**. Pour comprendre la suite, il faut se rappeler l'architecture utilisée ici :



Si **[Database1]** est la base **[dbpersonnes]**, on voit que la liaison entre le script et cette base passe par deux entités :

- le connecteur Python au SGBD MySQL ;
- le SGBD MySQL ;

Le script **[main]** va dialoguer avec l'ORM qui va ensuite dialoguer avec le connecteur Python. L'ORM dialogue avec ce connecteur avec les outils décrits dans les paragraphes |MySQL| et |PostgreSQL| notamment en émettant des ordres SQL. Le script **[main]** ne va pas utiliser d'ordres SQL. Il va s'appuyer sur l'API (Application Programming Interface) de l'ORM, faite de classes et d'interfaces.

Le script **[main]** est le suivant :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # imports
7.  from sqlalchemy import create_engine, Table, Column, Integer, String, MetaData, UniqueConstraint
8.  from sqlalchemy.exc import IntegrityError, InterfaceError
9.  from sqlalchemy.orm import mapper, sessionmaker
10.
11. from Personne import Personne
12.
13. # chaîne de connexion à une base de données MySQL
14. engine = create_engine("mysql+mysqlconnector://admpersonnes:nobody@localhost/dbpersonnes")
15.
16. # metadata
17. metadata = MetaData()
```

```
18.
19. # la table
20. personnes_table = Table("personnes", metadata,
21.                         Column('id', Integer, primary_key=True),
22.                         Column('prenom', String(30), nullable=False),
23.                         Column("nom", String(30), nullable=False),
24.                         Column("age", Integer, nullable=False),
25.                         UniqueConstraint('nom', 'prenom', name='uix_1')
26.                         )
27.
28. # le mapping
29. mapper(Personne, personnes_table, properties={
30.     'id': personnes_table.c.id,
31.     'prénom': personnes_table.c.prenom,
32.     'nom': personnes_table.c.nom,
33.     'âge': personnes_table.c.age
34. })
35.
36. # la session factory
37. Session = sessionmaker()
38. Session.configure(bind=engine)
39.
40. session = None
41. try:
42.     # une session
43.     session = Session()
44.
45.     # suppression de la table [personnes]
46.     session.execute("drop table if exists personnes")
47.
48.     # recréation de la table à partir du mapping
49.     metadata.create_all(engine)
50.
51.     # une insertion
52.     session.add(Personne().fromdict({"id": 67, "prénom": "x", "nom": "y", "âge": 10}))
53.     # session.commit()
54.
55.     # une requête
56.     personnes = session.query(Personne).all()
57.
58.     # affichage
59.     print("Liste des personnes ---------")
60.     for personne in personnes:
61.         print(personne)
62.
63.     # deux autres insertions dont la seconde échoue à cause de l'unicité (prénom,nom)
64.     session.add(Personne().fromdict({"id": 68, "prénom": "x1", "nom": "y1", "âge": 10}))
65.     session.add(Personne().fromdict({"id": 69, "prénom": "x1", "nom": "y1", "âge": 10}))
66.
67.     # une requête
68.     personnes = session.query(Personne).all()
69.
70.     # affichage
71.     print("Liste des personnes ---------")
72.     for personne in personnes:
73.         print(personne)
74.
75.     # validation de la session
76.     session.commit()
77.
78. except (InterfaceError, IntegrityError) as erreur:
79.     # affichage
80.     print(f"L'erreur suivante s'est produite : {erreur}")
81.     # annulation de la dernière session
82.     if session:
83.         print("rollback...")
84.         session.rollback()
85. finally:
86.     # on libère les ressources de la session
87.     if session:
88.         session.close()
```

Commentaires

- lignes 1-4 : l'application est configurée ;
- lignes 7-9 : on importe toute une série de classes et d'interfaces de la bibliothèque **[sqlalchemy]** ;

- ligne 11 : la classe **[Personne]** est importée ;
- ligne 14 : la chaîne de connexion à la base de données. Elle précise :
  - le SGBD utilisé (mysql) ;
  - le connecteur Python utilisé (mysql.connector sans le .) ;
  - l'utilisateur qui se connecte (admpersonnes) ;
  - son mot de passe (nobody) ;
  - la machine sur laquelle se trouve le SGBD (localhost=machine sur laquelle se trouve le script qui s'exécute) ;
  - le nom de la base de données (dbpersonnes) ;

  Avec ces informations, **[sqlalchemy]** peut se connecter à la base de données. A noter que le connecteur Python utilisé doit être déjà installé. **[sqlalchemy]** ne le fait pas.
- lignes 19-26 : description de la table **[personnes]** ;
- lignes 28-34 : mapping entre la classe **[Personne]** et la table **[personnes]** ;
- lignes 36-38 : la plupart des opérations **[sqlalchemy]** se font dans une session. La notion de session **[sqlalchemy]** est proche de celle de transaction SQL. Les sessions sont crées à partir de la classe **[Session]** rendue par la fonction **[sessionmaker]** de la ligne 37 ;
- ligne 38 : la classe **[Session]** est associée à la base **[dbpersonnes]** via la chaîne de connexion de la ligne 14 ;
- ligne 43 : on crée une session. Comme il a été dit, on peut rapprocher une session d'une transaction ;
- lignes 45-46 : la méthode **[Session.execute]** permet d'exécuter un ordre SQL. Ce n'est pas quelque chose de courant puisqu'on a dit que l'ORM permettait d'éviter le langage SQL ;
- lignes 48-49 : la méthode **[metadata.create_all]** permet de créer toutes les tables utilisant l'instance **[MetaData]** de la ligne 17. Nous n'en avons qu'une : la table **[personnes]** définie lignes 20-26. **[sqlalchemy]** va utiliser l'information de ces lignes pour créer la table. On a là un premier intérêt de l'ORM : il cache les spécificités des SGBD. En effet, l'ordre SQL **[create]** peut être très différent d'un SGBD à l'autre à cause des types donnés aux colonnes. Il n'y a pas eu d'uniformisation SQL des types de données. Ainsi l'ordre **[create]** varie d'un SGBD à l'autre. Ici, grâce à **[sqlalchemy]** :
  - nous décrivons de façon unique la table que nous désirons ;
  - **[sqlalchemy]** se débrouille pour générer le **[create]** qui va bien pour le SGBD qu'il a en face de lui ;
- ligne 52 : on ajoute un objet **[Personne]** à la session. Cela ne l'ajoute pas automatiquement en base de données. En effet, un ORM suit ses propres règles pour se synchroniser avec la base de données. Il va toujours chercher à optimiser le nombre de requêtes qu'il fait. Prenons un exemple. Le script ajoute (add) deux personnes (personne1, personne2) dans la session puis fait ensuite une requête : il veut voir toutes les personnes présentes dans la table. **[sqlalchemy]** peut procéder ainsi :
  - l'ajout de **[personne1]** peut se faire en mémoire. Il n'y a pas besoin pour l'instant de le mettre en base de données ;
  - idem pour **[personne2]** ;
  - vient ensuite la requête de type **[select]**. Il faut alors récupérer toutes les lignes de la table **[personnes]**. **[sqlalchemy]** va alors mettre **[personne1, personne2]** en base puis faire la requête ;

  [sqlalchemy] va ainsi faire des optimisations transparentes pour le développeur.
- ligne 56 : pour faire une requête de type **[select]** (je veux voir …), on utilise la méthode **[Session.query]**. Le paramètre de la méthode **[query]** est la classe mappée avec la table interrogée. Cette méthode rend un type **[Query]**. La méthode **[Query.all]** demande tous les objets **[Personne]** de la session. On lui ramène toutes les lignes de la table **[personnes]**, chacune sous la forme d'un objet **[Personne]**. Pour faire cela, **[sqlalchemy]** utilise le mapping qui a été fait entre la classe **[Personne]** et la table **[personnes]**. Le résultat de la ligne 56 est une liste d'objets **[Personne]** ;
- lignes 58-61 : on affiche les éléments de la liste **[personnes]**. Parce que la classe **[Personne]** dérive de la classe **[BaseEntity]**, la méthode **[Personne.__str__]** utilisée ici implicitement dans la ligne 61, est en fait la méthode **[BaseEntity.__str__]** qui rend la chaîne jSON de l'objet appelant. Cette chaîne est la chaîne jSON du dictionnaire **[Personne.asdict]** (cf. |BaseEntity|). Nous avons dit qu'après le mapping, on allait trouver la propriété **[_sa_instance_state]** dans chaque objet **[Personne]**. Or la valeur de cette propriété n'est pas un type **[BaseEntity]**. Il faut donc l'exclure du dictionnaire de la classe **[Personne]** sinon l'affichage 'plante'. C'est ce qui a été fait dans le script **[config]** ;
- lignes 63-65 : on ajoute deux autres personnes qui ont les mêmes nom et prénom. Or on a une contrainte d'unicité sur l'union de ces deux colonnes. Une erreur devrait donc se produire. C'est ce qu'on cherche à voir ;
- lignes 67-68 : on demande de nouveau la liste de toutes les personnes de la base ;
- lignes 70-73 : et on les affiche ;
- lignes 75-76 : la session est validée 'commitée'. Comme son nom l'indique, la transaction sous-jacente va être validée ;
- on va voir à l'exécution que les lignes 67-76 ne vont pas être exécutées à cause de l'exception produite par la ligne 65. On va alors aller aux lignes 78-84 pour gérer l'exception ;
- ligne 78 : l'exception **[InterfaceError]** se produit si **[sqlalchemy]** n'arrive pas à se connecter à la base de données **[dbpersonnes]**. L'exception **[IntegrityError]** se produit ligne 65 ;
- ligne 80 : on affiche l'erreur ;
- lignes 82-84 : si la session existe, on l'annule. Cela revient à annuler la transaction sous-jacente ;
- lignes 85-88 : dans tous les cas, erreur ou pas, la session est fermée pour libérer des ressources ;

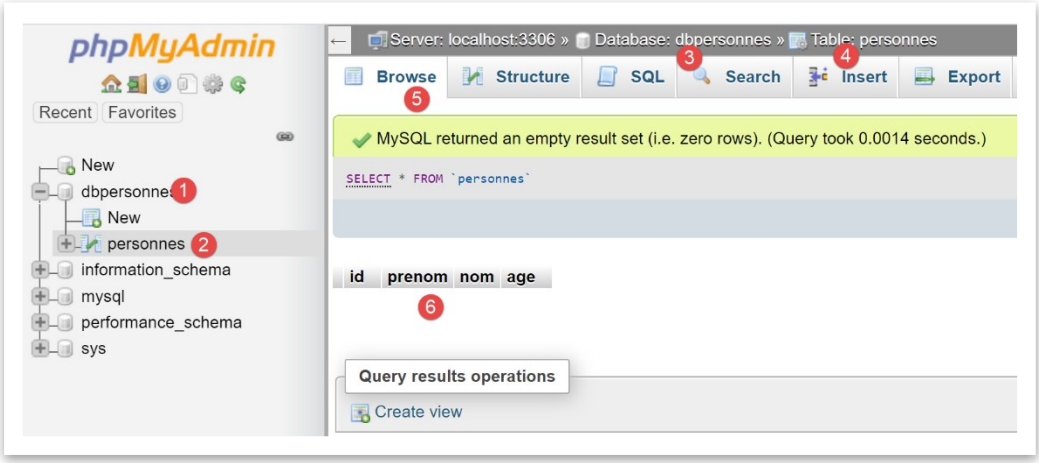
Les résultats de l'exécution sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/sqlalchemy/01/main.py
2.  Liste des personnes ---------
```

```
3.  {"nom": "y", "prénom": "x", "id": 67, "âge": 10}
4.  L'erreur suivante s'est produite : (raised as a result of Query-invoked autoflush; consider using a
    session.no_autoflush block if this flush is occurring prematurely)
5.  (mysql.connector.errors.IntegrityError) 1062 (23000): Duplicate entry 'y1-x1' for key 'uix_1'
6.  [SQL: INSERT INTO personnes (id, prenom, nom, age) VALUES (%(id)s, %(prenom)s, %(nom)s, %(age)s)]
7.  [parameters: ({'id': 68, 'prenom': 'x1', 'nom': 'y1', 'age': 10}, {'id': 69, 'prenom': 'x1', 'nom': 'y1',
    'age': 10})]
8.  (Background on this error at: http://sqlalche.me/e/13/gkpj)
9.  rollback...
10.
11. Process finished with exit code 0
```

- lignes 2-3 : la liste des personnes après la 1ère insertion ;
- ligne 5 : l'exception **[IntegrityError]** qui s'est produite lorsqu'on a ajouté deux personnes ayant les mêmes nom et prénom ;
- lignes 6-7 : on notera l'ordre SQL qui a échoué. C'est un ordre INSERT paramétré : **[sqlalchemy]** a inséré les deux personnes avec un unique INSERT. On voit là qu'il a essayé d'optimiser les ordres SQL émis ;

Maintenant allons-voir, avec phpMyAdmin, le contenu de la table **[personnes]** :

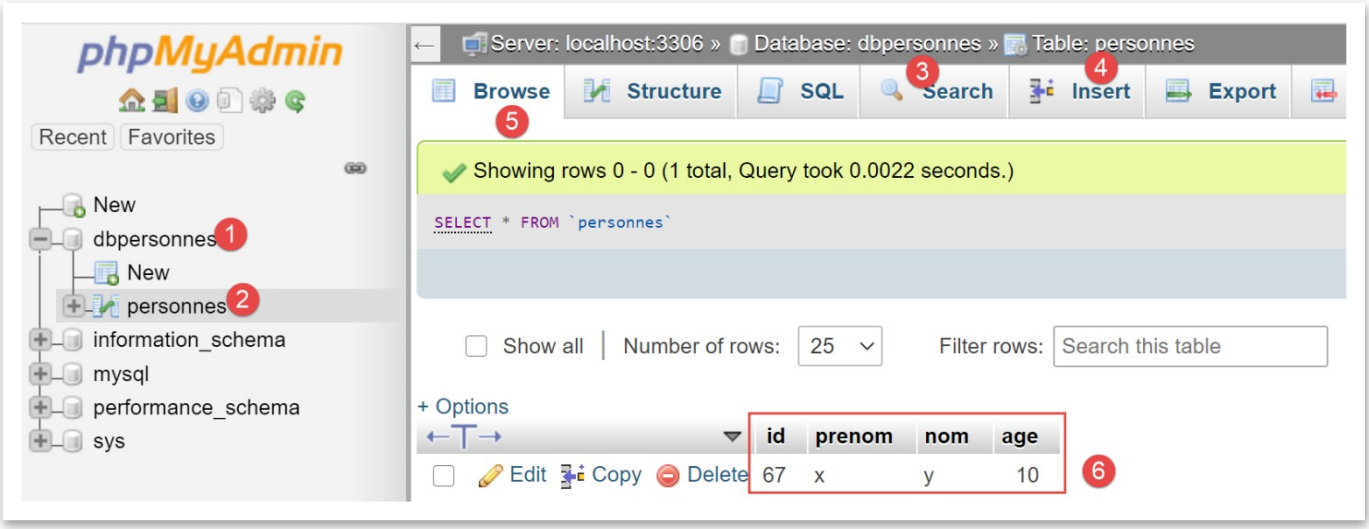


On voit en **[6]** que la table est vide. Il n'y a même pas la 1ère personne que le script avait mis dans la session. Ceci parce que celle-ci se déroulait dans une transaction et que celle-ci a été défaite dans la clause **[except]** du script **[main]**.

Procédons maintenant à la modification suivante dans **[main]** :

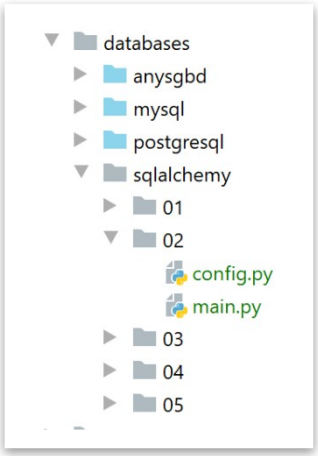
```
1.      # une insertion
2.      session.add(Personne().fromdict({"id": 67, "prénom": "x", "nom": "y", "âge": 10}))
3.      # session.commit()
```

Après avoir ajouté une personne ligne 2, nous décommentons la ligne 3. L'opération **[session.commit]** va valider la transaction sous-jacente et une nouvelle transaction va démarrer. Après exécution, le contenu de la table **[personnes]** est le suivant :



On voit en **[6]** que la 1ère insertion a été conservée. Cela vient du fait qu'elle a été faite au sein d'une transaction 1 et que l'erreur qui a suivi a été faite au sein d'une transaction 2.

19.3 Scripts 02 : les mappings de [sqlalchemy]



Les scripts 02 sont une variante des scripts 01. On essaie de faire le maximum de configurations dans **[config.py]**. On y configure maintenant l'environnement **[sqlalchemy]** de l'application :

```
1.  def configure():
2.      # chemin absolu référence des chemins relatifs de la configuration
3.      root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
4.      # chemins absolus des dépendances
5.      absolute_dependencies = [
6.          # BaseEntity, MyException, Personne, Utils
7.          f"{root_dir}/classes/02/entities",
8.      ]
9.
10.     # on fixe le syspath
11.     from myutils import set_syspath
12.     set_syspath(absolute_dependencies)
13.
14.     # imports
15.     from sqlalchemy import create_engine, Table, Column, Integer, String, MetaData, UniqueConstraint
16.     from sqlalchemy.orm import mapper, sessionmaker
17.
18.     # lien vers une base de données MySQL
19.     engine = create_engine("mysql+mysqlconnector://admpersonnes:nobody@localhost/dbpersonnes")
20.
21.     # metadata
22.     metadata = MetaData()
23.
24.     # la table
25.     personnes_table = Table("personnes", metadata,
26.                             Column('id', Integer, primary_key=True),
27.                             Column('prenom', String(30), nullable=False),
28.                             Column("nom", String(30), nullable=False),
29.                             Column("age", Integer, nullable=False),
30.                             UniqueConstraint('nom', 'prenom', name='uix_1')
31.                             )
32.
33.     # le mapping
34.     from Personne import Personne
35.
36.     mapper(Personne, personnes_table, properties={
37.         'id': personnes_table.c.id,
38.         'prénom': personnes_table.c.prenom,
39.         'nom': personnes_table.c.nom,
40.         'âge': personnes_table.c.age
41.     })
42.
43.     # la session factory
44.     Session = sessionmaker()
45.     Session.configure(bind=engine)
46.
47.     # on met ces informations dans la config
```

```
48.     config = {}
49.     config["Session"] = Session
50.     config["metadata"] = metadata
51.     config["engine"] = engine
52.     config["personnes_table"] = personnes_table
53.
54.     # configuration des classes
55.     from Personne import Personne
56.     Personne.excluded_keys = ['_sa_instance_state']
57.
58.     # on rend la config
59.     return config
```

Commentaires

- lignes 2-12 : configuration du Python Path ;
- lignes 14-45 : on configure l'environnement **[sqlalchemy]** ;
- lignes 47-52 : l'environnement **[sqlalchemy]** est mis dans le dictionnaire de la configuration ;
- lignes 54-56 : on configure la classe **[Personne]** ;

Avec cette configuration le script **[main]** devient le suivant :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # le syspath est configuré - on fait les imports
7.  from sqlalchemy.exc import IntegrityError, DatabaseError, InterfaceError
8.  from sqlalchemy.orm.exc import FlushError
9.
10. from Personne import Personne
11.
12. session = None
13. try:
14.     # une session
15.     session = config["Session"]()
16.
17.     # suppression de la table [personnes]
18.     session.execute("drop table if exists personnes")
19.
20.     # recréation de la table à partir du mapping
21.     config["metadata"].create_all(config["engine"])
22.
23.     # deux insertions
24.     session.add(Personne().fromdict({"prénom": "x", "nom": "y", "âge": 10}))
25.     personne = Personne().fromdict({"prénom": "x1", "nom": "y1", "âge": 7})
26.     session.add(personne)
27.
28.     # validation des deux insertions
29.     session.commit()
30.
31.     # une requête
32.     personnes = session.query(Personne).all()
33.
34.     # affichage
35.     print("Liste des personnes-----------")
36.     for personne in personnes:
37.         print(personne)
38.
39.     # deux autres insertions dont la seconde échoue
40.     session.add(Personne().fromdict({"prénom": "x2", "nom": "y2", "âge": 10}))
41.     session.add(Personne().fromdict({"prénom": "x2", "nom": "y2", "âge": 10}))
42.
43.     # une requête
44.     personnes = session.query(Personne).all()
45.
46.     # affichage
47.     print("Liste des personnes-----------")
48.     for personne in personnes:
49.         print(personne)
50.
51.     # validation de la session
52.     session.commit()
53.
```

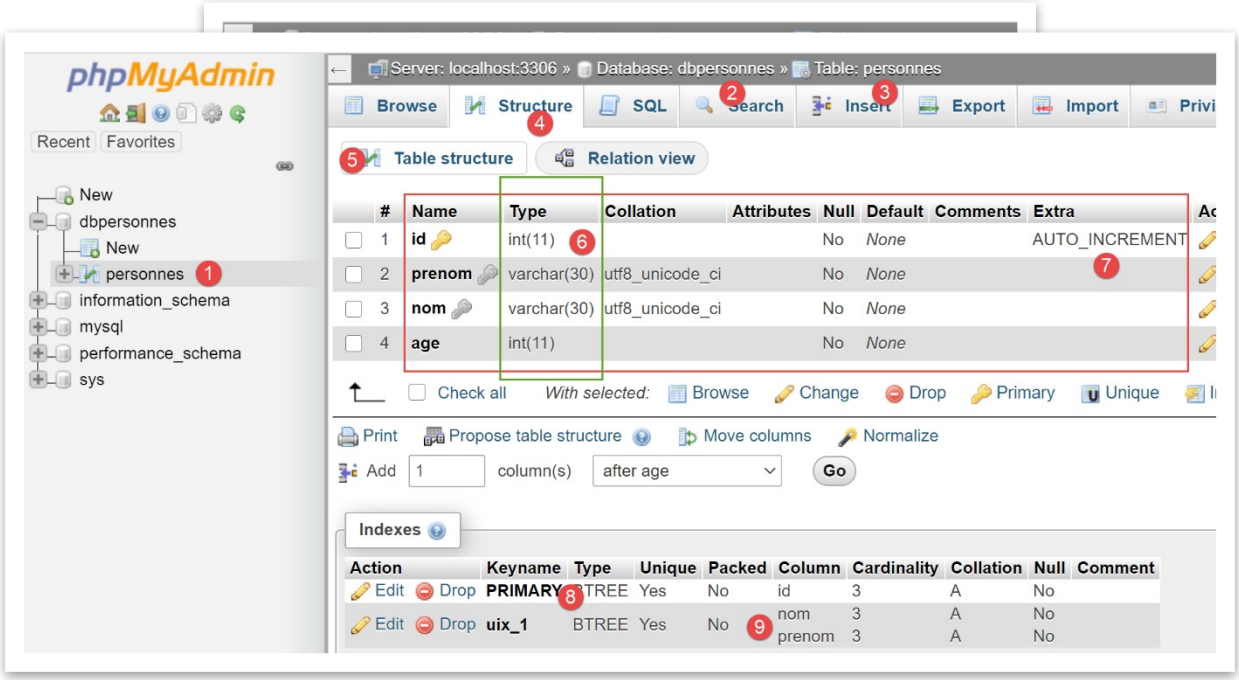
```
except (FlushError, DatabaseError, InterfaceError, IntegrityError) as erreur:
    # affichage
    print(f"L'erreur suivante s'est produite : {erreur}")
    # annulation de la dernière session
    if session:
        print("rollback...")
        session.rollback()
finally:
    # affichage
    print("Travail terminé...")
    # on libère les ressources de la session
    if session:
        session.close()
```

Les résultats de l'exécution sont les suivants :

```
C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/sqlalchemy/02/main.py
Liste des personnes----------
{"âge": 10, "nom": "y", "prénom": "x", "id": 1}
{"âge": 7, "nom": "y1", "prénom": "x1", "id": 2}
L'erreur suivante s'est produite : (raised as a result of Query-invoked autoflush; consider using a
session.no_autoflush block if this flush is occurring prematurely)
(mysql.connector.errors.IntegrityError) 1062 (23000): Duplicate entry 'y2-x2' for key 'uix_1'
[SQL: INSERT INTO personnes (prenom, nom, age) VALUES (%(prenom)s, %(nom)s, %(age)s)]
[parameters: {'prenom': 'x2', 'nom': 'y2', 'age': 10}]
(Background on this error at: http://sqlalche.me/e/13/gkpj)
rollback...
Travail terminé...

Process finished with exit code 0
```

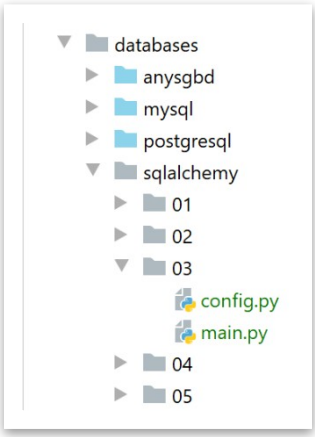
Dans phpMyAdmin, la table **[personnes]** est devenue la suivante :



Maintenant, regardons la table **[personnes]** générée par **[sqlalchemy]** :

- en **[6]**, les types utilisés pour les différentes colonnes ;
- en **[7]**, on voit que la colonne **[id]** a l'attribut **[AUTO_INCREMENT]**. Cela signifie que lors de l'insertion d'une ligne dans la table, si cette ligne n'a pas de valeur pour la colonne **[id]**, celle-ci sera générée par MySQL de façon incrémentale : 1, 2, 3, … Cette propriété nous évite de nous préoccuper de la valeur de la clé primaire lorsque nous faisons une insertion dans la table : nous laissons MySQL la générer ;
- en **[8]**, on voit que la colonne **[id]** est clé primaire ;
- en **[9]**, on retrouve la contrainte d'unicité sur les champs **[nom, prenom]** ;

19.4 Scripts 03 : manipulation des entités de la session [sqlalchemy]



Le fichier de configuration **[config]** est le même que dans l'exemple précédent. Dans le script **[main]** on fait les opérations classiques **[INSERT, UPDATE, DELETE, SELECT]** sur la table **[personnes]** à l'aide des méthodes de **[sqlalchemy]** :

```
1.  # on configure l'application
2.  import config
3.
4.  config = config.configure()
5.
6.  # imports
7.  from sqlalchemy import func
8.  from sqlalchemy.exc import IntegrityError, DatabaseError, InterfaceError
9.  from sqlalchemy.orm.session import Session
10. from Personne import Personne
11.
12. # affiche le contenu de la table [personnes]
13. def affiche_table(session: Session):
14.     print("----------------")
15.     # une requête
16.     personnes = session.query(Personne).all()
17.     # affichage
18.     affiche_personnes(personnes)
19.
20. # affiche une liste de personnes
21. def affiche_personnes(personnes: list):
22.     print("----------------")
23.     # affichage
24.     for personne in personnes:
25.         print(personne)
26.
27.
28. # main --------------------------
29. session = None
30. try:
31.     # une session
32.     session = config["Session"]()
33.
34.     # suppression de la table [personnes]
35.     # checkfirst=True : vérifie d'abord que la table existe
36.     config["personnes_table"].drop(config["engine"], checkfirst=True)
37.
38.     # recréation de la table à partir du mapping
39.     config["metadata"].create_all(config["engine"])
40.
41.     # des insertions
42.     session.add(Personne().fromdict({"prénom": "Pierre", "nom": "Nicazou", "âge": 35}))
43.     session.add(Personne().fromdict({"prénom": "Géraldine", "nom": "Colou", "âge": 26}))
44.     session.add(Personne().fromdict({"prénom": "Paulette", "nom": "Girondé", "âge": 56}))
45.
46.     # on affiche le contenu de la session
47.     affiche_table(session)
48.
49.     # liste des personnes par ordre alphabétique des noms et à nom égal par ordre alphabétique des prénoms
50.     personnes = session.query(Personne).order_by(Personne.nom.desc(), Personne.prénom.desc())
51.
```

```
52.     # affichage
53.     affiche_personnes(personnes)
54.
55.     # liste des personnes ayant un âge dans l'intervalle [20,40] par ordre décroissant de l'âge
56.     # puis à âge égal par ordre alphabétique des noms et à nom égal par ordre alphabétique des prénoms
57.     personnes = session.query(Personne). \
58.         filter(Personne.âge >= 20, Personne.âge <= 40). \
59.         order_by(Personne.âge.desc(), Personne.nom.asc(), Personne.prénom.asc())
60.
61.     # affichage
62.     affiche_personnes(personnes)
63.
64.     # insertion de mme Bruneau
65.     bruneau = Personne().fromdict({"prénom": "Josette", "nom": "Bruneau", "âge": 46})
66.     session.add(bruneau)
67.     # modification de son âge
68.     bruneau.âge = 47
69.
70.     # liste des personnes ayant Bruneau pour nom
71.     personne = session.query(Personne).filter(func.lower(Personne.nom) == "bruneau").first()
72.
73.     # affichage
74.     affiche_personnes([personne])
75.
76.     # suppression de Mme Bruneau
77.     session.delete(personne)
78.
79.     # liste des personnes ayant Bruneau pour nom
80.     personnes = session.query(Personne).filter(func.lower(Personne.nom) == "bruneau")
81.
82.     # affichage
83.     affiche_personnes(personnes)
84.
85.     # validation de la session
86.     session.commit()
87.
88. except (DatabaseError, InterfaceError, IntegrityError) as erreur:
89.     # affichage
90.     print(f"L'erreur suivante s'est produite : {erreur}")
91.     # annulation de la dernière session
92.     if session:
93.         session.rollback()
94.
95. finally:
96.     # affichage
97.     print("Travail terminé...")
98.     # on libère les ressources de la session
99.     if session:
100.        session.close()
```

Commentaires

- lignes 20-25 : la fonction **[affiche_personnes]** affiche les éléments d'une liste de personnes ;
- lignes 12-18 : la fonction **[affiche_table]** affiche le contenu de la table **[personnes]** ;
- lignes 34-36 : on supprime la table **[personnes]**. Contrairement aux versions précédentes, on n'utilise pas un ordre SQL mais une méthode de **[sqlalchemy]** :
  - config**["personnes_table"]** est l'objet **[Table]** décrivant la table **[personnes]** ;
  - config**["engine"]** est la chaîne de connexion à la base de données **[dbpersonnes]** ;
  - le paramètre nommé **[checkfirst=True]** demande à ce que l'opération ne se fasse que si la table **[personnes]** existe ;
- lignes 38-39 : la table **[personnes]** est recréée ;
- lignes 41-44 : trois personnes sont mises dans la session. On rappelle qu'elles ne sont pas forcément insérées immédiatement dans la table **[personnes]**. Cela dépend de la stratégie de **[sqlalchemy]** qui vise la performance ;
- lignes 46-47 : le contenu de la table **[personnes]** es affiché. Si les insertions des trois personnes n'avaient pas encore été faites elles le sont maintenant à cause de cette demande ;
- lignes 49-50 : un exemple d'utilisation de la méthode **[order_by]** qui permet de présenter les résultats d'une requête dans un certain ordre. La syntaxe **[order_by(critère1, critère2)]** affiche les résultats d'abord selon le critère **[critère1]** et lorsque des lignes présentent la même valeur de **[critère1]**, elles sont alors triées selon le critère **[critère2]**. On peut mettre plusieurs critères ainsi ;
- lignes 55-59 : introduisent la notion de filtre avec la méthode **[filter]**. La notation **[filter(critère1, critère2)]** fait un ET logique (AND) entre les critère utilisés ;
- lignes 64-67 : une nouvelle personne est mise en session ;

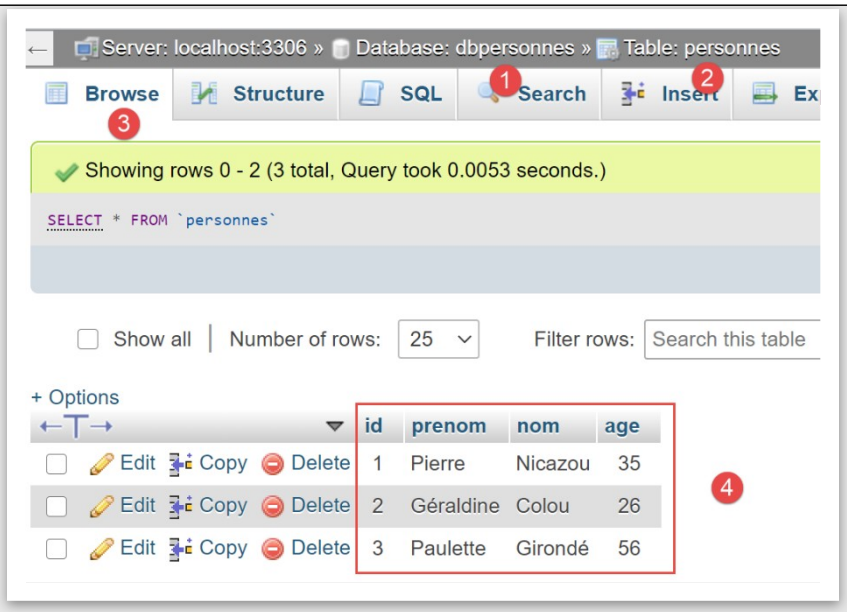
- lignes 70-71 : un autre exemple de requête filtrée. La fonction **[func.lower(param)]** rend **[param]** en minuscules. Il existe ainsi d'autres fonctions disponibles notées **[func.xx]**. Dans l'expression de la ligne 71 :
  - **[session.query.filter]** rend une liste d'objets **[Personne]** ;
  - **[session.query.filter.first]** rend le 1er élément de cette liste ;
- ligne 77 : on supprime un élément de la session ;
- ligne 86 : la session est validée ;

Les résultats de l'exécution sont les suivants :

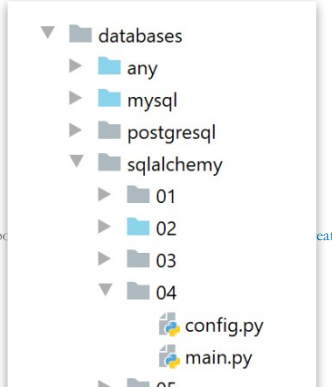
```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/sqlalchemy/03/main.py
2.  ----------------
3.  ----------------
4.  {"âge": 35, "nom": "Nicazou", "prénom": "Pierre", "id": 1}
5.  {"âge": 26, "nom": "Colou", "prénom": "Géraldine", "id": 2}
6.  {"âge": 56, "nom": "Girondé", "prénom": "Paulette", "id": 3}
7.  ----------------
8.  {"âge": 35, "nom": "Nicazou", "prénom": "Pierre", "id": 1}
9.  {"âge": 56, "nom": "Girondé", "prénom": "Paulette", "id": 3}
10. {"âge": 26, "nom": "Colou", "prénom": "Géraldine", "id": 2}
11. ----------------
12. {"âge": 35, "nom": "Nicazou", "prénom": "Pierre", "id": 1}
13. {"âge": 26, "nom": "Colou", "prénom": "Géraldine", "id": 2}
14. ----------------
15. {"prénom": "Josette", "nom": "Bruneau", "âge": 47, "id": 4}
16. ----------------
17. Travail terminé...
18.
19. Process finished with exit code 0
```

- lignes 4-6 : le contenu de la session ;
- lignes 8-10 : le contenu de la session dans l'ordre décroissant des noms ;
- lignes 12-13 : le contenu de la session pour les personnes dont l'âge est dans l'intervalle **[20, 40]** ;
- ligne 15 : la personne de nom « bruneau » ;

Dans phpMyAdmin, le contenu de la table **[personnes]** à la fin de l'exécution est le suivant :



19.5 Scripts 04 : utilisation d'une base [PostgreSQL]



Le dossier **[04]** est une copie du dossier **[03]**. On change une unique chose, la chaîne de connexion dans le fichier **[config]** :

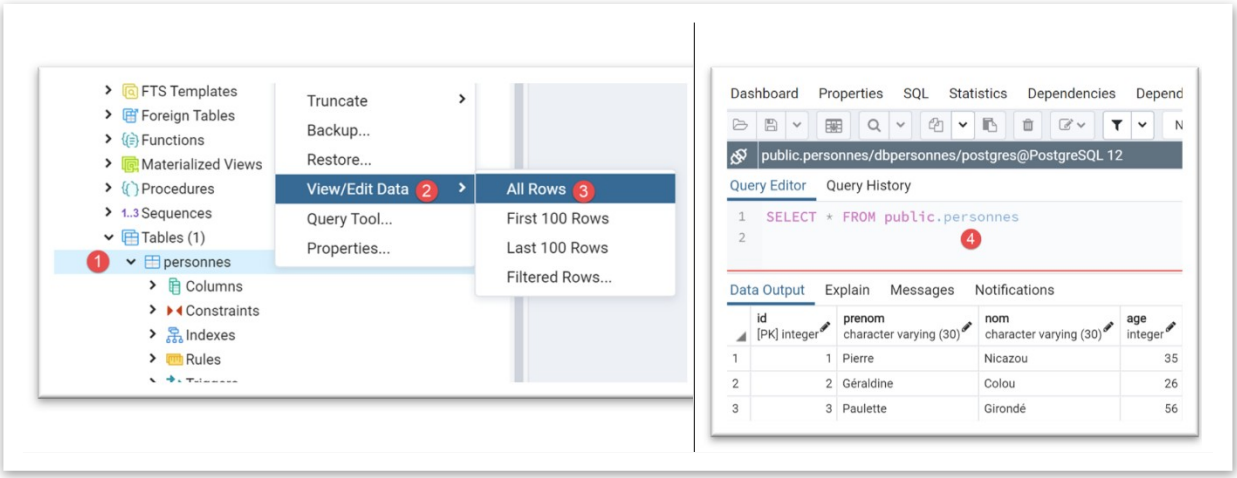
```
1.      # lien vers une base de données PostgreSQL
2.      engine = create_engine("postgresql+psycopg2://admpersonnes:nobody@localhost/dbpersonnes")
```

Désormais cette chaîne de connexion désigne la base **[dbpersonnes]** d'un SGBD **[PostgreSQL]**. On notera l'utilisation du connecteur **[psycopg2]**. Il faut que celui-ci soit installé.

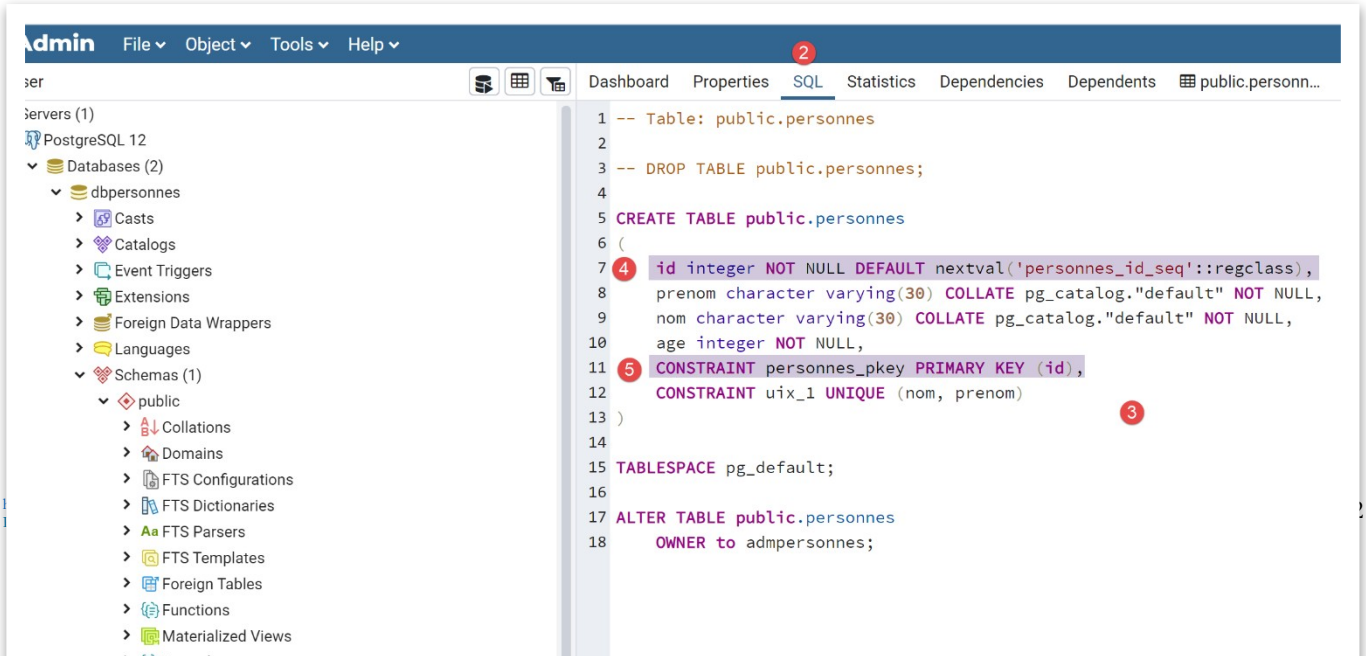
L'exécution du script **[main]** donne les résultats suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/sqlalchemy/04/main.py
2.  ----------------
3.  ----------------
4.  {"nom": "Nicazou", "prénom": "Pierre", "id": 1, "âge": 35}
5.  {"nom": "Colou", "prénom": "Géraldine", "id": 2, "âge": 26}
6.  {"nom": "Girondé", "prénom": "Paulette", "id": 3, "âge": 56}
7.  ----------------
8.  {"nom": "Nicazou", "prénom": "Pierre", "id": 1, "âge": 35}
9.  {"nom": "Girondé", "prénom": "Paulette", "id": 3, "âge": 56}
10. {"nom": "Colou", "prénom": "Géraldine", "id": 2, "âge": 26}
11. ----------------
12. {"nom": "Nicazou", "prénom": "Pierre", "id": 1, "âge": 35}
13. {"nom": "Colou", "prénom": "Géraldine", "id": 2, "âge": 26}
14. ----------------
15. {"prénom": "Josette", "nom": "Bruneau", "âge": 47, "id": 4}
16. ----------------
17. Travail terminé...
18.
19. Process finished with exit code 0
```

Avec l'outil **[pgAdmin]** (cf. paragraphe |pgAdmin|), la table **[personnes]** est dans l'état suivant :

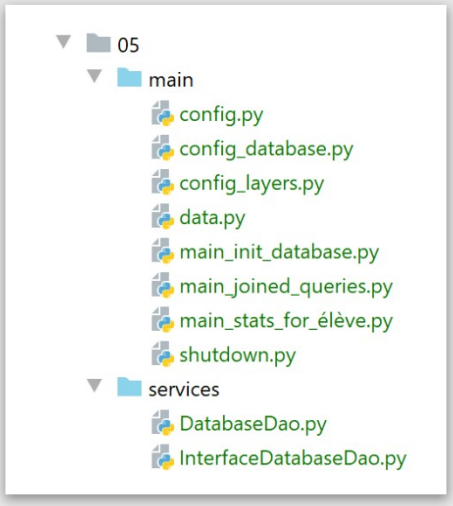


La table **[personnes]** a été générée avec le code SQL suivant :



- en **[4-5]**, on voit que la colonne **[id]** est clé primaire. On voit également qu'elle a une valeur par défaut **[mot clé DEFAULT]** qui fait que si on insère une ligne sans clé primaire, celle-ci sera générée par le SGBD. C'est un fonctionnement fréquent : on laisse le SGBD générer les clés primaires ;

Cette version 05 des scripts **[sqlalchemy]** montre bien la facilité de passer d'un SGBD à l'autre : il a suffi de changer la chaîne de connexion dans un script de configuration. Rien d'autre n'a changé. Si on compare les types des colonnes **[id, nom, prenom, age]** ci-dessus avec ceux de la table MySQL de l'exemple |02|, on voit qu'ils sont différents. **[sqlalchemy]** les adapte au SGBD utilisé. Cette facilité de s'adapter à un nouveau SGBD est une raison suffisante pour adopter **[sqlalchemy]** ou un autre ORM.

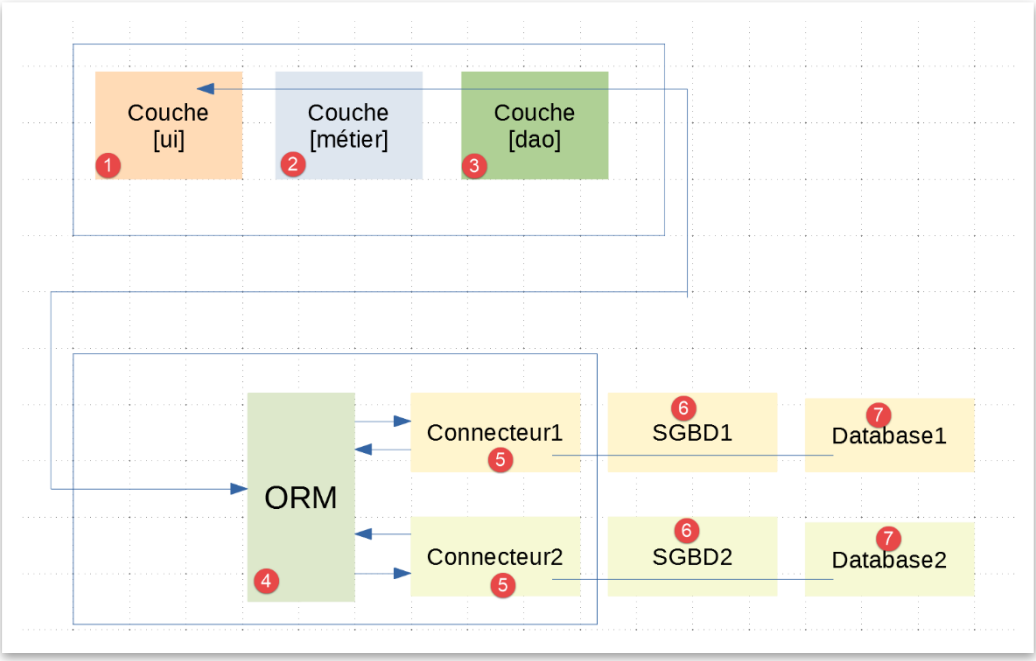


19.6 Scripts 05 : exemple complet

L'exemple étudié est une reprise de celui étudié au paragraphe |troiscouches-v01|. Cet exemple présentait une architecture à trois couches **[ui, métier, dao]** qui manipulait des entités **[Classe, Elève, Matière, Note]**. Les entités étaient codées en dur dans une couche **[dao]**. Nous les mettons maintenant dans une base de données. Nous utiliserons deux SGBD : MySQL et PostgreSQL.

19.6.1 L'architecture de l'application

L'architecture de l'application sera la suivante :

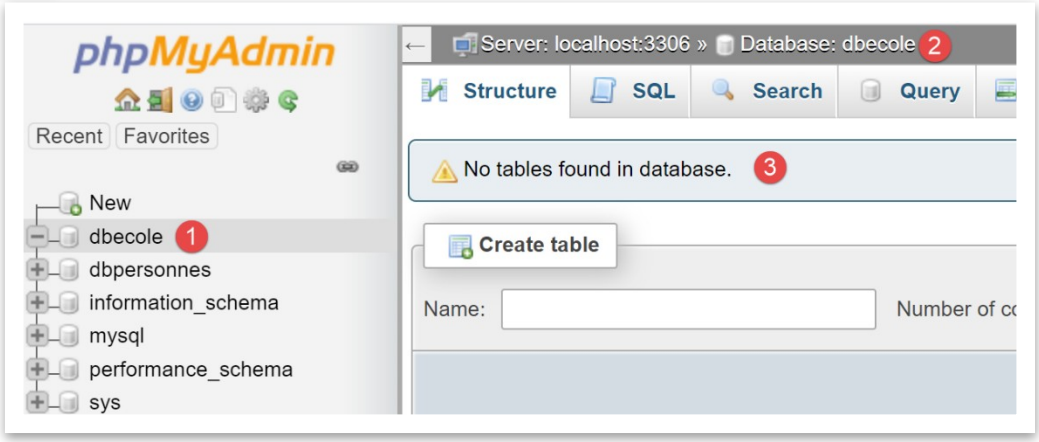


- en **[1-3]**, on trouve les couches **[ui, métier, dao]** déjà présentes dans l'exemple |troiscouches-v01|. La couche **[dao]** communique désormais avec la couche **[ORM]** ;

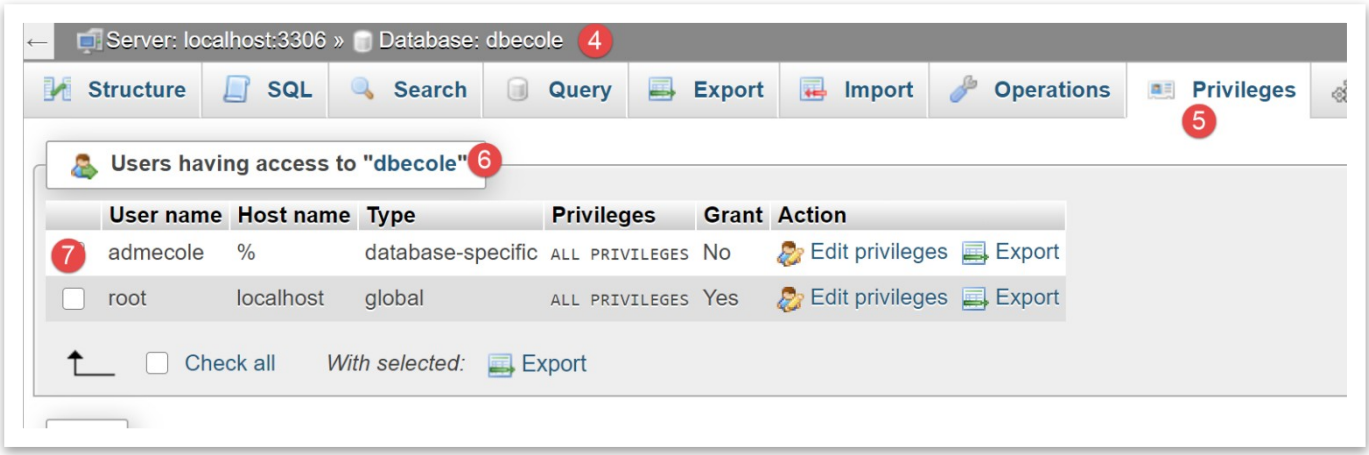
- les couches **[1-5]** sont implémentées par du code Python ;



19.6.2

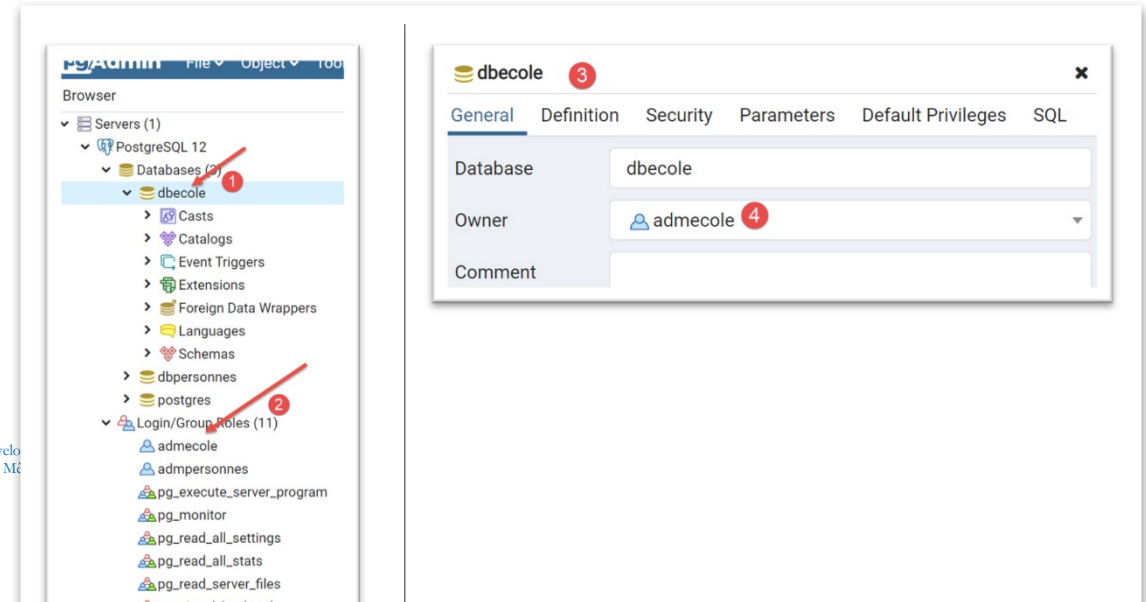


Nous construisons une base MySQL nommée **[dbecole]** propriété de l'utilisateur **[admecole]** ayant le mot de passe **[mdpecole]**. Pour cela nous suivons la procédure décrite au paragraphe |création d'une base de données| :



- en **[1]**, la base **[dbecole]** sans tables **[3]** ;
- en **[7]**, l'utilisateur **[admecole]** a tous les privilèges sur cette base de données ;

On fait de même avec le SGBD PostgreSQL. Nous construisons une base nommée **[dbecole]** propriété de l'utilisateur **[admecole]** ayant le mot de passe **[mdpecole]**. Pour cela nous suivons la procédure décrite au paragraphe |création d'une base de données| :



- en **[1]**, la base **[dbecole]** ;
- en **[2]**, l'utilisateur **[admecole]** ;
- en **[3-4]**, la base **[dbecole]** est la propriété de l'utilisateur **[admecole]** ;

19.6.3 Les entités manipulées par l'application

Dans l'application |troiscouches v01|, les entités manipulées étaient les suivantes (cf. |entités|). Ce sont ces entités qui vont être stockées dans les bases de données précédentes. Nous ne dupliquerons pas ces entités dans la nouvelle application. Nous irons les chercher là où elles sont déjà définies.

La classe **[Classe]** :

```
1.  # imports
2.  from BaseEntity import BaseEntity
3.  from MyException import MyException
4.  from Utils import Utils
5.
6.
7.  class Classe(BaseEntity):
8.      # attributs exclus de l'état de la classe
9.      excluded_keys = []
10.
11.     # propriétés de la classe
12.     @staticmethod
13.     def get_allowed_keys() -> list:
14.         # id : identifiant de la classe
15.         # nom : nom de la classe
16.         return BaseEntity.get_allowed_keys() + ["nom"]
17.
18.     # getter
19.     @property
20.     def nom(self: object) -> str:
21.         return self.__nom
22.
23.     #  setters
24.     @nom.setter
25.     def nom(self: object, nom: str):
26.         # nom doit être une chaîne de caractères non vide
27.         if Utils.is_string_ok(nom):
28.             self.__nom = nom
29.         else:
30.             raise MyException(11, f"Le nom de la classe {self.id} doit être une chaîne de caractères non vide")
```

La classe **[Elève]** :

```
1.  # imports
2.  from BaseEntity import BaseEntity
3.  from Classe import Classe
4.  from MyException import MyException
5.
6.  from Utils import Utils
7.
8.
9.  class Elève(BaseEntity):
10.     # attributs exclus de l'état de la classe
11.     excluded_keys = []
12.
13.     # propriétés de la classe
14.     @staticmethod
15.     def get_allowed_keys() -> list:
16.         # id : identifiant de l'élève
17.         # nom : nom de l'élève
18.         # prénom : prénom de l'élève
19.         # classe : classe de l'élève
20.         return BaseEntity.get_allowed_keys() + ["nom", "prénom", "classe"]
21.
22.     # getters
23.     @property
24.     def nom(self: object) -> str:
25.         return self.__nom
26.
27.     @property
28.     def prénom(self: object) -> str:
29.         return self.__prénom
30.
31.     @property
32.     def classe(self: object) -> Classe:
33.         return self.__classe
34.
35.     #  setters
36.     @nom.setter
37.     def nom(self: object, nom: str) -> str:
```

```
        # nom doit être une chaîne de caractères non vide
        if Utils.is_string_ok(nom):
            self.__nom = nom
        else:
            raise MyException(41, f"Le nom de l'élève {self.id} doit être une chaîne de caractères non vide")

    @prénom.setter
    def prénom(self: object, prénom: str) -> str:
        # prénom doit être une chaîne de caractères non vide
        if Utils.is_string_ok(prénom):
            self.__prénom = prénom
        else:
            raise MyException(42, f"Le prénom de l'élève {self.id} doit être une chaîne de caractères non vide")

    @classe.setter
    def classe(self: object, value):
        try:
            # on attend un type Classe
            if isinstance(value, Classe):
                self.__classe = value
            # ou un type dict
            elif isinstance(value,dict):
                self.__classe=Classe().fromdict(value)
            # ou un type json
            elif isinstance(value,str):
                self.__classe = Classe().fromjson(value)
        except BaseException as erreur:
            raise MyException(43, f"L'attribut [{value}] de l'élève {self.id} doit être de type Classe ou dict ou json.
Erreur : {erreur}")
```

La classe **[Matière]** :

```
# imports
from BaseEntity import BaseEntity
from MyException import MyException
from Utils import Utils


class Matière(BaseEntity):
    # attributs exclus de l'état de la classe
    excluded_keys = []

    # propriétés de la classe
    @staticmethod
    def get_allowed_keys() -> list:
        # id : identifiant de la matière
        # nom : nom de la matière
        # coefficient : coefficient de la matière
        return BaseEntity.get_allowed_keys() + ["nom", "coefficient"]

    # getter
    @property
    def nom(self: object) -> str:
        return self.__nom

    @property
    def coefficient(self: object) -> float:
        return self.__coefficient

    #  setters
    @nom.setter
    def nom(self: object, nom: str):
        # nom doit être une chaîne de caractères non vide
        if Utils.is_string_ok(nom):
            self.__nom = nom
        else:
            raise MyException(21, f"Le nom de la matière {self.id} doit être une chaîne de caractères non vide")

    @coefficient.setter
    def coefficient(self, coefficient: float):
        # le coefficient doit être un réel >=0
        erreur = False
        if isinstance(coefficient, (int, float)):
            if coefficient >= 0:
                self.__coefficient = coefficient
            else:
                erreur = True
        else:
            erreur = True
        # erreur ?
        if erreur:
            raise MyException(22, f"Le coefficient de la matière {self.nom} doit être un réel >=0")
```

La classe **[Note]** :

```python
# imports
from BaseEntity import BaseEntity
from Elève import Elève
from Matière import Matière
from MyException import MyException


class Note(BaseEntity):
    # attributs exclus de l'état de la classe
    excluded_keys = []

    # propriétés de la classe
    @staticmethod
    def get_allowed_keys() -> list:
        # id : identifiant de la note
        # valeur : la note elle-même
        # élève : élève (de type Elève) concerné par la note
        # matière : matière (de type Matière) concernée par la note
        # l'objet Note est donc la note d'un élève dans une matière
        return BaseEntity.get_allowed_keys() + ["valeur", "élève", "matière"]

    # getters
    @property
    def valeur(self: object) -> float:
        return self.__valeur

    @property
    def élève(self: object) -> Elève:
        return self.__élève

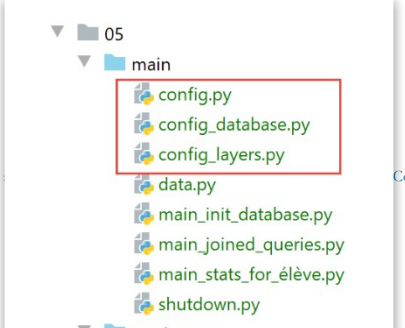
    @property
    def matière(self: object) -> Matière:
        return self.__matière

    # getters
    @valeur.setter
    def valeur(self: object, valeur: float):
        # la note doit être un réel entre 0 et 20
        if isinstance(valeur, (int, float)) and 0 <= valeur <= 20:
            self.__valeur = valeur
        else:
            raise MyException(31,
                f"L'attribut {valeur} de la note {self.id} doit être un nombre dans l'intervalle [0,20]")

    @élève.setter
    def élève(self: object, value):
        try:
            # on attend un type Elève
            if isinstance(value, Elève):
                self.__élève = value
            # ou un type dict
            elif isinstance(value, dict):
                self.__élève = Elève().fromdict(value)
            # ou un type json
            elif isinstance(value, str):
                self.__élève = Elève().fromjson(value)
        except BaseException as erreur:
            raise MyException(32,
                f"L'attribut [{value}] de la note {self.id} doit être de type Elève ou dict ou json. Erreur : {erreur}")

    @matière.setter
    def matière(self: object, value):
        try:
            # on attend un type Matière
            if isinstance(value, Matière):
                self.__matière = value
            # ou un type dict
            elif isinstance(value, dict):
                self.__matière = Matière().fromdict(value)
            # ou un type json
            elif isinstance(value, str):
                self.__matière = Matière().fromjson(value)
        except BaseException as erreur:
            raise MyException(33,
                f"L'attribut [{value}] de la note {self.id} doit être de type Matière ou dict ou json. Erreur :
{erreur}")
```

19.6.4 Configuration



La configuration a été éclatée sur plusieurs fichiers :

- la configuration générale dans **[config.py]** : elle établit le Python Path de l'application et instancie les couches de l'architecture ;
- la configuration de **[sqlalchemy]** dans **[config_database]** : elle fait les mappings Classes / Tables ;
- les couches de l'application sont configurées dans **[config_layers]** ;

Le fichier **[config]** est le suivant :

```
1.  def configure(config: dict) -> dict:
2.      import os
3.
4.      # étape 1 ---
5.      # on établit le Python Path de l'application
6.      # chemin absolu du dossier de ce script
7.      script_dir = os.path.dirname(os.path.abspath(__file__))
8.
9.      # chemin absolu référence des chemins relatifs de la configuration
10.     root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
11.
12.     # chemins absolus des dépendances
13.     absolute_dependencies = [
14.         # BaseEntity, MyException
15.         f"{root_dir}/classes/02/entities",
16.         # projet troiscouches v01
17.         f"{root_dir}/troiscouches/v01/interfaces",
18.         f"{root_dir}/troiscouches/v01/services",
19.         f"{root_dir}/troiscouches/v01/entities",
20.         # dossiers du présent projet
21.         script_dir,
22.         f"{script_dir}/../services",
23.     ]
24.
25.     # mise à jour du syspath
26.     from myutils import set_syspath
27.     set_syspath(absolute_dependencies)
28.
29.     # étape 2 ------
30.     # configuration base de données
31.     import config_database
32.     config = config_database.configure(config)
33.
34.     # étape 3 ------
35.     # instanciation des couches de l'application
36.     import config_layers
37.     config = config_layers.configure(config)
38.
39.     # on rend la config
40.     return config
```

- lignes 4-27 : construction du Python Path de l'application ;
- lignes 29-32 : configuration **[sqlalchemy]** ;
- lignes 34-37 : configuration des couches de l'application ;

Le fichier **[config_database]** est le suivant :

```
1.  def configure(config: dict) -> dict:
2.      # config['sgbd'] est le nom du SGBD utilisé
3.      # mysql : MySQL
4.      # pgres : PostgreSQL
5.
6.      # configuration sqlalchemy
7.      from sqlalchemy import Table, Column, Integer, MetaData, String, Float, ForeignKey, create_engine
8.
9.      from sqlalchemy.orm import mapper, relationship, sessionmaker
10.
11.     # chaînes de connexion aux bases de données exploitées
12.     engines = {
13.         'mysql': "mysql+mysqlconnector://admecole:mdpecole@localhost/dbecole",
14.         'pgres': "postgresql+psycopg2://admecole:mdpecole@localhost/dbecole"
15.     }
16.     # chaîne de connexion à la base de données exploitée
17.     engine = create_engine(engines[config['sgbd']])
18.
```

```
    # metadata
    metadata = MetaData()

    # les tables de la base
    tables = {}
    # les classes mappées
    from Classe import Classe
    from Elève import Elève
    from Note import Note
    from Matière import Matière

    # la table des classes
    tables['classes'] = classes_table = \
        Table("classes", metadata,
              Column('id', Integer, primary_key=True),
              Column('nom', String(30), nullable=False),
              )

    mapper(Classe, tables['classes'], properties={
        'id': classes_table.c.id,
        'nom': classes_table.c.nom
    })

    # la table des élèves
    tables['élèves'] = élèves_table = \
        Table("élèves", metadata,
              Column('id', Integer, primary_key=True),
              Column('nom', String(30), nullable=False),
              Column('prénom', String(30), nullable=False),
              # un élève appartient à une classe
              Column('classe_id', Integer, ForeignKey('classes.id')),
              )
    # mapping
    mapper(Elève, tables['élèves'], properties={
        'id': élèves_table.c.id,
        'nom': élèves_table.c.nom,
        'prénom': élèves_table.c.prénom,
        'classe': relationship(Classe, backref="élèves", lazy="select")
    })

    # la table des matières
    tables['matières'] = matières_table = \
        Table("matières", metadata,
              Column('id', Integer, primary_key=True),
              Column('nom', String(30), nullable=False),
              Column('coefficient', Float, nullable=False)
              )
    # mapping
    mapper(Matière, tables['matières'], properties={
        'id': matières_table.c.id,
        'nom': matières_table.c.nom,
        "coefficient": matières_table.c.coefficient
    })

    # la table des notes
    tables['notes'] = notes_table = \
        Table("notes", metadata,
              Column('id', Integer, primary_key=True),
              Column('valeur', Float, nullable=False),
              # une note est celle d'un élève
              Column('élève_id', Integer, ForeignKey('élèves.id')),
              # une note est celle d'une matière
              Column('matière_id', Integer, ForeignKey('matières.id')),
              )

    # mapping
    mapper(Note, tables['notes'], properties={
        'id': notes_table.c.id,
        'valeur': notes_table.c.valeur,
        'élève': relationship(Elève, backref="notes", lazy="select"),
        'matière': relationship(Matière, backref="notes", lazy="select")
    })

    # configuration des entités [BaseEntity]
    Elève.excluded_keys = ['_sa_instance_state', 'notes', 'classe']
    Classe.excluded_keys = ['_sa_instance_state', 'élèves']
    Matière.excluded_keys = ['_sa_instance_state', 'notes']
```

```
96.     Note.excluded_keys = ['_sa_instance_state', 'matière', 'élève']
97.
98.     # la session factory
99.     Session = sessionmaker()
100.    Session.configure(bind=engine)
101.
102.    # une session
103.    session = Session()
104.
105.    # on enregistre certaines informations dans le dictionnaire de la configuration
106.    config['database'] = {"engine": engine, "metadata": metadata, "tables": tables, "session": session}
107.
108.    # on rend la config
109.    return config
```

Commentaires

- lignes 1-4 : la fonction **[configure]** reçoit un dictionnaire en paramètre. Seule la clé **[sgbd]** est exploitée. Elle vaut **[mysql]** si la base est une base MySQL, **[pgres]** si la base est une base PostgreSQL ;
- lignes 6-9 : imports d'éléments de **[sqlalchemy]**. Le script **[config_database]** fait les mappings entre les tables de la base **[dbecole]** et les entités **[Classes, Elève, Matière, Note]**. Dans la table, les données de l'entité sont encapsulées dans une ligne. Dans le code Python, elles sont encapsulées dans un objet. D'où le nom ORM (Object Relational Mapper) : l'ORM fait un mapping (une liaison) entre les lignes d'une base de données relationnelle et des objets. Dans cette application, nous avons quatre entités **[Classe, Elève, Matière, Note]** qui seront reliées à quatre tables **[classes, élèves, matières, notes]**. Notez que les noms des tables peuvent comporter des caractères accentués ;
- lignes 11-17 : la chaîne de connexion à la base de données exploitée. Celle-ci dépend de l'élément config**['sgbd']** ;
- lignes 24-28 : les entités de l'application qui vont faire l'objet d'un mapping **[sqlalchemy]**. Lorsque ces lignes seront exécutées, le Python Path aura déjà été établi par le script **[config]** ;
- lignes 30-40 : le mapping entre l'entité **[Classe]** et la table **[classes]** ;
- lignes 30-35 : la table **[classes]** est définie avec la classe **[Table]** de **[sqlalchemy]**. Nous indiquons que cette table a deux colonnes :
  - la colonne **[id]** qui est clé primaire et qui est le n° de la classe, ligne 33 ;
  - la colonne **[nom]** qui contient le nom de la classe, ligne 34 ;
- lignes 31-32 : notez que la syntaxe x=y=z est légale en Python : la valeur de z est affectée à y puis la valeur de y à x ;
- lignes 37-40 : on liste les correspondances entre les colonnes de la table **[classes]** et les propriétés de l'entité **[Classe]** ;
- lignes 42-57 : le mapping entre l'entité **[Elève]** et la table **[élèves]** ;
- lignes 51-57 : la table **[élèves]** est définie avec la classe **[Table]** de **[sqlalchemy]**. Nous indiquons que cette table a quatre colonnes :
  - la colonne **[id]** qui est clé primaire et qui est le n° de l'élève, ligne 45 ;
  - la colonne **[nom]** qui contient le nom de l'élève, ligne 46 ;
  - la colonne **[prénom]** qui contient le prénom de l'élève, ligne 47. Notez qu'un nom de colonne peut comporter des caractères accentués ;
  - ligne 49, la colonne **[classe_id]** qui contiendra le n° de la classe à laquelle appartient l'élève. On appelle cela une clé étrangère. **[élèves.classe_id]** est une clé étrangère (ForeignKey) sur la colonne **[classes.id]**. Cela signifie que la valeur de **[élèves.classe_id]** doit exister dans la colonne **[classes.id]** ;
- lignes 51-57 : on liste les correspondances entre les colonnes de la table **[élèves]** et les propriétés de l'entité **[Elève]** :
  - les lignes 53-55 sont simples à comprendre ;
  - la ligne 56 est plus difficile : elle définit la valeur de la propriété **[Elève.classe]** comme étant calculée par la relation (relationship) de clé étrangère qui relie les tables **[élèves]** et **[classes]**. Les paramètres de la fonction **[relationship]** sont les suivants :
    - **[Classe]** : c'est le nom de l'entité avec laquelle l'entité **[Elève]** a une relation de clé étrangère. Celle-ci doit se matérialiser dans la table **[élèves]** par la présence d'une clé étrangère sur la table **[classes]**. Nous savons que celle-ci existe ;
    - **[backref="élèves"]** : le nom d'une propriété qui sera **ajoutée** à l'entité **[Classe]**. **[Classe.élèves]** sera la liste de tous les élèves de la classe. **Cette propriété ne doit pas déjà exister**. Si elle existe déjà, il faut simplement choisir un autre nom ici pour **[backref]**. Le développeur n'a pas à gérer cette propriété. C'est **[sqlalchemy]** qui le fera. Il doit simplement savoir qu'elle existe, ajoutée par **[sqlalchemy]**, et qu'il peut l'utiliser dans son code ;
    - **[lazy='select']** : cela signifie que l'ORM ne doit pas chercher à donner immédiatement une valeur à la propriété **[Elève.classe]**. Il ne doit chercher sa valeur que lorsque le code la demande explicitement. Ainsi :
      - si le code demande la liste de tous les élèves, ceux-ci seront ramenés mais leur propriété **[classe]** ne sera pas calculée ;
      - un peu plus tard, le code s'intéresse à un élève **[e]** précis et référence sa classe **[e.classe]**. Cette référence va alors forcer **[sqlalchemy]** à faire une requête en base pour ramener la classe de l'élève, ceci de façon transparente pour le développeur ;
      - aussi mettre **[lazy='select']** vise à éviter les requêtes inutiles en base ;
  - ligne 56 : lorsque l'ORM récupère une ligne de la table **[élèves]**, il récupère les informations **[id, nom, prénom, classe_id]**. A partir de là, il doit construire un objet Elève(id, nom, prénom, classe). Pour les propriétés **[id, nom,**

prénom] ça ne pose pas de difficultés. Pour la propriété **[classe]**, c'est plus compliqué. Sa valeur est une référence d'objet de type **[Classe]**. Or l'ORM ne dispose que d'une information **[élèves.classe_id]**. Comme **[élèves.classe_id]** est une clé étrangère sur la colonne **[classes.id]**, on lui dit ici d'utiliser cette relation pour récupérer dans la table **[classes]** la ligne de n° id=**[élèves.classe_id]** (elle existe forcément) et de créer à partir de cette ligne l'objet **[Classe]** attendu par la propriété **[Elève.classe]** ;

- lignes 59-71 : le mapping entre l'entité **[Matière]** et la table **[matières]** ;
- lignes 59-65 : définition de la table **[sqlalchemy]** nommée **[matières]** ;
- lignes 66-71 : on liste les correspondances entre les colonnes de la table **[matières]** et les propriétés de l'entité **[Matière]**. Il n'y a pas de difficultés ici ;
- lignes 73-90 : le mapping entre l'entité **[Note]** et la table **[notes]** ;
- lignes 73-82 : définition de la table **[sqlalchemy]** nommée **[notes]**. Elle a deux clés étrangères :
  - ligne 79, la colonne **[notes.élève_id]** prend ses valeurs dans la colonne **[élèves.id]**]. Cette clé étrangère matérialise le fait qu'une note est la note d'un élève précis ;
  - ligne 81, la colonne **[notes.matière_id]** prend ses valeurs dans la colonne **[matières.id]**. Cette clé étrangère matérialise le fait qu'une note est une note dans une matière précise ;
- lignes 84-90 : le mapping entre l'entité **[Note]** et la table **[notes]** :
  - ligne 88 : la propriété **[Note.élève]** doit avoir pour valeur une instance de type **[Elève]**. L'ORM n'a pour information dans la ligne de la table **[notes]** que la colonne **[notes.élève_id]** qui référence la colonne **[élèves.id]**. On dit ici d'utiliser cette relation de clé étrangère pour retrouver l'intance **[Elève]** dont on a la note. Par ailleurs **[relationship(Elève, backref="notes", …)]** va créer la nouvelle propriété **[Elève.notes]** qui sera la liste des notes de l'élève. Cette propriété ne doit pas déjà exister dans la classe **[Elève]** ;
  - ligne 89 : la propriété **[Note.matière]** doit avoir pour valeur une instance de type **[Matière]**. L'ORM n'a pour information dans la ligne de la table **[notes]** que la colonne **[notes.matière_id]** qui référence la colonne **[matières.id]**. On dit ici d'utiliser cette relation de clé étrangère pour retrouver l'intance **[Matière]** dont on a la note. Par ailleurs **[relationship(Matière, backref="notes", …)]** va créer la nouvelle propriété **[Matière.notes]** qui sera la liste des notes dans la matière. Cette propriété ne doit pas déjà exister dans la classe **[Matière]** ;
- lignes 92-96 : on définit pour chaque entité dérivée de **[BaseEntity]** la liste des propriétés à exclure du dictionnaire des propriétés de l'entité (BaseEntity.asdict). Nous avons vu que **[sqlalchemy]** ajoutait la propriété **[_sa_instance_state]** à toutes entités mappées. On ne la veut pas dans le dictionnaire des propriétés. Par ailleurs on a vu que les mappings précédents avaient ajouté de nouvelles propriétés aux entités :
  - **[Elève.notes]** : toutes les notes de l'élève ;
  - **[Classe.élèves]** : tous les élèves de la classe ;
  - **[Matière.notes]** : toutes les notes de la matière ;

  En général, on ve veut pas ces propriétés ajoutées dans l'état de l'entité. En effet, calculer leur valeur a un coût SQL et cette valeur est souvent inutile. Ainsi si on récupère l'élève de nom 'X' :
  - l'ORM va ramener une entité **[Elève(id, nom, prénom, classe, notes)]**. A cause de **[lazy='select']**, les propriétés **[classe, notes]** liées à des clés étrangères de la base n'auront pas été calculées ;
  - maintenant si j'affiche la chaîne jSON de cet élève, on sait que ce sera la chaîne jSON du dictionnaire **[asdict]** de l'entité. Si les propriétés **[classe]** et **[notes]** sont dedans, **[sqlalchemy]** va être obligé de faire des requêtes SQL pour calculer leurs valeurs. C'est coûteux. Si on peut éviter ces requêtes, c'est préférable ;
  - ici, nous avons exclu toutes les propriétés liées à une clé étrangère ;
- lignes 98-100 : instanciation et configuration d'une **[Session factory]** (factory=usine de production). L'objet **[Session]** sert à créer des sessions **[sqlalchemy]** adossées à des transactions ;
- lignes 102-103 : création d'une session sqlalchemy] ;
- ligne 106 : certains éléments de la configuration **[sqlalchemy]** sont mis dans le dictionnaire global de la configuration de l'application ;
- ligne 109 : on rend ce dictionnaire ;

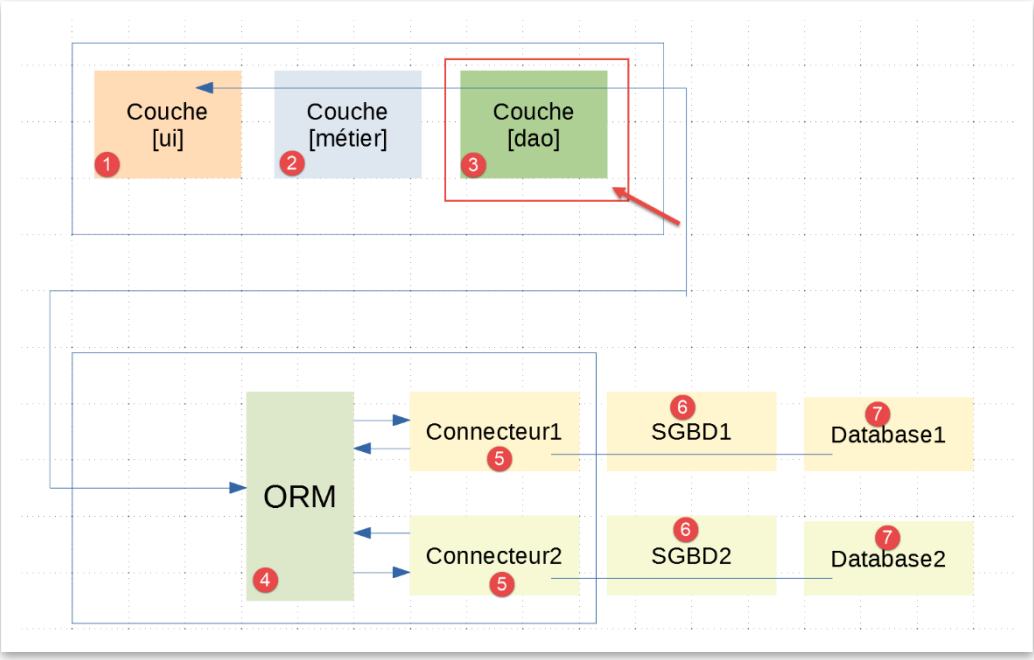
Le fichier **[config_layers]** configure les couches de l'application :

```
1.  def configure(config: dict) -> dict:
2.      # instanciation couche [dao]
3.      from DatabaseDao import DatabaseDao
4.      dao = DatabaseDao(config)
5.
6.      # instanciation de la couche [métier]
7.      from Métier import Métier
8.      métier = Métier(dao)
9.
10.     # instanciation de la couche [ui]
11.     from Console import Console
12.     ui = Console(métier)
13.
14.     # on met les couches dans la config
15.     config['dao'] = dao
16.     config['métier'] = métier
17.     config['ui'] = ui
```

```
18.
19.     # on rend la config
20.     return config
```

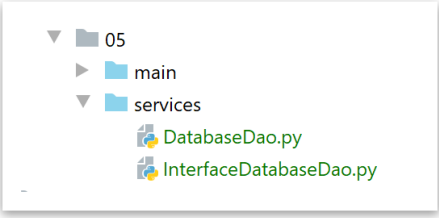
- ligne 1 : la fonction **[configure]** reçoit le dictionnaire de la configuration globale de l'application ;
- lignes 2-12 : les couches de l'application sont instanciées ;
- lignes 15-17 : les références des couches sont mises dans la configuration globale ;
- ligne 20 : on rend la nouvelle configuration ;

19.6.5 La couche [dao] - 1



Il faut comprendre ici que la couche **[dao] [3]** communique avec l'ORM **[sqlalchemy] [4]** configuré comme il a été décrit dans le paragraphe précédent. Des trois couches **[ui, métier, dao]** de l'application |troiscouches v01|, seule la couche **[dao]** doit être réécrite. Les couches **[ui, métier]** sont conservées.

L'implémentation de la couche **[dao]** a été placée dans le dossier **[services]** :



[InterfaceDatabaseDao] est l'interface de la couche **[dao]** :

```
1.  from abc import ABC, abstractmethod
2.
3.  from InterfaceDao import InterfaceDao
4.
5.
6.  class InterfaceDatabaseDao(InterfaceDao, ABC):
7.
8.      # initialisation de la base de données
9.      @abstractmethod
10.     def init_database(self, data: dict):
11.         pass
```

- ligne 6 : l'interface **[InterfaceDatabaseDao]** dérive à la fois de la classe **[ABC]** pour être une classe abstraite et de l'interface **[InterfaceDao]** du projet |troiscouches v01| ;
- lignes 8-11 : on ajoute la méthode **[init_database]** aux méthodes héritées de **[InterfaceDao]**. Elle aura pour rôle d'initialiser la base de données avec les données du dictionnaire **[data]** qu'on lui passe en paramètre ligne 10 ;

Rappelons que l'interface **[InterfaceDao]** était la suivante :

```
1.  # imports
2.  from abc import ABC, abstractmethod
3.
4.  # interface Dao
5.  from Elève import Elève
6.
7.
8.  class InterfaceDao(ABC):
9.      # liste des classes
10.     @abstractmethod
11.     def get_classes(self: object) -> list:
12.         pass
13.
14.     # liste des élèves
15.     @abstractmethod
16.     def get_élèves(self: object) -> list:
17.         pass
18.
19.     # liste des matières
20.     @abstractmethod
21.     def get_matières(self: object) -> list:
22.         pass
23.
24.     # liste des notes
25.     @abstractmethod
26.     def get_notes(self: object) -> list:
27.         pass
28.
29.     # liste des notes d'un élève
30.     @abstractmethod
31.     def get_notes_for_élève_by_id(self: object, élève_id: int) -> list:
32.         pass
33.
34.     # chercher un élève par son id
35.     @abstractmethod
36.     def get_élève_by_id(self: object, élève_id: int) -> Elève:
37.         pass
```

L'implémentation de la couche **[dao]** est la suivante :

```
1.  from sqlalchemy.exc import DatabaseError, IntegrityError, InterfaceError
2.
3.  from Classe import Classe
4.  from Elève import Elève
5.  from InterfaceDatabaseDao import InterfaceDatabaseDao
6.  from Matière import Matière
7.  from MyException import MyException
8.  from Note import Note
9.
10.
11. class DatabaseDao(InterfaceDatabaseDao):
12.
13.     def __init__(self, config: dict):
14.         # database = {"engine": engine, "metadata": metadata, "tables": tables, "session": session}
15.         self.database = config['database']
16.         self.session = self.database['session']
17.
18.     def init_database(self, data: dict):
19.         …
20. …
```

- ligne 11 : la classe **[DatabaseDao]** implémente l'interface **[InterfaceDatabaseDao]** ;
- lignes 13-16 : le constructeur de la classe. Il reçoit en paramètre, le dictionnaire de la configuration de l'application ;
- ligne 15 : on mémorise la configuration **[sqlalchemy]** ;
- ligne 16 : on mémorise la session **[sqlalchemy]** au travers de laquelle on va manipuler la base de données ;
- ligne 18 : la méthode **[init_database]** initialise la base de données avec le dictionnaire **[data]** ;

Le dictionnaire **[data]** est implémenté par le script **[data.py]** suivant :

```
1.  def configure():
2.      from Classe import Classe
3.      from Elève import Elève
4.      from Matière import Matière
5.      from Note import Note
6.
7.      # on instancie les classes
8.      classe1 = Classe().fromdict({"id": 1, "nom": "classe1"})
9.      classe2 = Classe().fromdict({"id": 2, "nom": "classe2"})
10.     classes = [classe1, classe2]
11.     # les matières
12.     matière1 = Matière().fromdict({"id": 1, "nom": "matière1", "coefficient": 1})
13.     matière2 = Matière().fromdict({"id": 2, "nom": "matière2", "coefficient": 2})
14.     matières = [matière1, matière2]
15.     # les élèves
16.     élève11 = Elève().fromdict({"id": 11, "nom": "nom1", "prénom": "prénom1", "classe": classe1})
17.     élève21 = Elève().fromdict({"id": 21, "nom": "nom2", "prénom": "prénom2", "classe": classe1})
18.     élève32 = Elève().fromdict({"id": 32, "nom": "nom3", "prénom": "prénom3", "classe": classe2})
19.     élève42 = Elève().fromdict({"id": 42, "nom": "nom4", "prénom": "prénom4", "classe": classe2})
20.     élèves = [élève11, élève21, élève32, élève42]
21.     # les notes des élèves dans les différentes matières
22.     note1 = Note().fromdict({"id": 1, "valeur": 10, "élève": élève11, "matière": matière1})
23.     note2 = Note().fromdict({"id": 2, "valeur": 12, "élève": élève21, "matière": matière1})
24.     note3 = Note().fromdict({"id": 3, "valeur": 14, "élève": élève32, "matière": matière1})
25.     note4 = Note().fromdict({"id": 4, "valeur": 16, "élève": élève42, "matière": matière1})
26.     note5 = Note().fromdict({"id": 5, "valeur": 6, "élève": élève11, "matière": matière2})
27.     note6 = Note().fromdict({"id": 6, "valeur": 8, "élève": élève21, "matière": matière2})
28.     note7 = Note().fromdict({"id": 7, "valeur": 10, "élève": élève32, "matière": matière2})
29.     note8 = Note().fromdict({"id": 8, "valeur": 12, "élève": élève42, "matière": matière2})
30.     notes = [note1, note2, note3, note4, note5, note6, note7, note8]
31.     # on regroupe l'ensemble
32.     data = {"élèves": élèves, "classes": classes, "matières": matières, "notes": notes}
33.     # on rend les données
34.     return data
```

- ligne 34 : le dictionnaire qui sera passé à la méthode **[init_database]**. Ce dictionnaire est composé des clés suivantes (ligne 32) :
  - **[élèves]** : la liste des élèves ;
  - **[classes]** : la liste des classes ;
  - **[matières]** : la liste des matières ;
  - **[notes]** : la liste des notes de tous les élèves dans toutes les matières ;

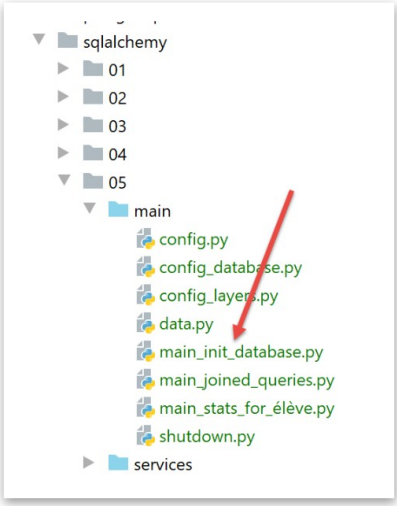
Revenons à la méthode **[init_database]** :

```
1.  def init_database(self, data: dict):
2.          # config de la bd
3.          database = self.database
4.          engine = database['engine']
5.          metadata = database['metadata']
6.          tables = database['tables']
7.
8.          try:
9.              # suppression des tables existantes
10.             # checkfirst=True : vérifie d'abord que la table existe
11.             tables["notes"].drop(engine, checkfirst=True)
12.             tables["matières"].drop(engine, checkfirst=True)
13.             tables["élèves"].drop(engine, checkfirst=True)
14.             tables["classes"].drop(engine, checkfirst=True)
15.
16.             # recréation des tables à partir du mapping
17.             metadata.create_all(engine)
18.
19.             # remplissage des tables
20.             session = self.session
21.
22.             # classes
23.             classes = data["classes"]
24.             for classe in classes:
25.                 session.add(classe)
26.
27.             # matières
28.             matières = data["matières"]
29.             for matière in matières:
```

```
30.                 session.add(matière)
31.
32.             # élèves
33.             élèves = data["élèves"]
34.             for élève in élèves:
35.                 session.add(élève)
36.
37.             # notes
38.             notes = data["notes"]
39.             for note in notes:
40.                 session.add(note)
41.
42.             # commit
43.             session.commit()
44.         except (DatabaseError, InterfaceError, IntegrityError) as erreur:
45.             # annulation de la session
46.             if session:
47.                 session.rollback()
48.             # on remonte l'exception
49.             raise MyException(23, f"{erreur}")
```

- lignes 3-6 : on récupère des informations dans la configuration de la base de données ;
- lignes 9-14 : nous avons vu que la configuration **[sqlalchemy]** avait mappé quatre entités sur quatre tables **[élèves, matières, classes, notes]**. On commence par supprimer ces tables si elles existent ;
- lignes 16-17 : on recrée les quatre tables qu'on vient de supprimer ;
- lignes 22-25 : on met toutes les classes dans la session ;
- lignes 27-30 : on met toutes les matières dans la session ;
- lignes 32-35 : on met tous les élèves dans la session ;
- lignes 37-40 : on met toutes les notes dans la session ;
- pour faire ces ajouts, on a suivi un ordre. On a commencé par les entités n'ayant pas de relations avec d'autres entités pour terminer par celles qui en avaient. Ainsi lorsqu'on ajoute les élèves dans la session, les classes que ceux-ci référencent sont déjà en session ;
- ligne 43 : la session **[sqlalchemy]** est validée. Après cette opération, on est sûrs que toutes les données en session ont été synchronisées avec la base de données. En clair, elles sont arrivées dans les tables. Ceci a pu se faire grâce aux mappings qui ont été faits dans la configuration de **[sqlalchemy]**. **[sqlalchemy]** sait comment chaque entité doit être stockée dans les tables. **[sqlalchemy]** a également généré les clés étrangères que peuvent posséder les tables ;
- lignes 44-49 : si on rencontre un problème, la session **[sqlalchemy]** est annulée et ligne 49, une exception est levée ;

19.6.6 Initialisation de la base de données



Le script **[main_init_database]** initialise la base de données avec le contenu du script **[data.py]**. Son code est le suivant :

```
1.  # on attend un paramètre mysql ou pgres
2.  import sys
3.
4.  syntaxe = f"{sys.argv[0]} mysql / pgres"
5.  erreur = len(sys.argv) != 2
6.  if not erreur:
7.      sgbd = sys.argv[1].lower()
8.      erreur = sgbd != "mysql" and sgbd != "pgres"
```

```
9.  if erreur:
10.     print(f"syntaxe : {syntaxe}")
11.     sys.exit()
12.
13. # on configure l'application
14. import config
15. config = config.configure({'sgbd': sgbd})
16.
17. # le syspath est configuré - on peut faire les imports
18. from MyException import MyException
19.
20. # on récupère les données à mettre en base
21. import data
22. data = data.configure()
23.
24. # on récupère la couche [dao]
25. dao = config["dao"]
26.
27. # ---------- main
28. try:
29.     # création et initialisation des tables de la bd
30.     dao.init_database(data)
31. except MyException as ex:
32.     # on affiche l'erreur
33.     print(f"L'erreur suivante s'est produite : {ex}")
34. finally:
35.     # libération des ressources mobilisées par l'application
36.     import shutdown
37.     shutdown.execute(config)
38. # fin
39. print("Travail terminé...")
```

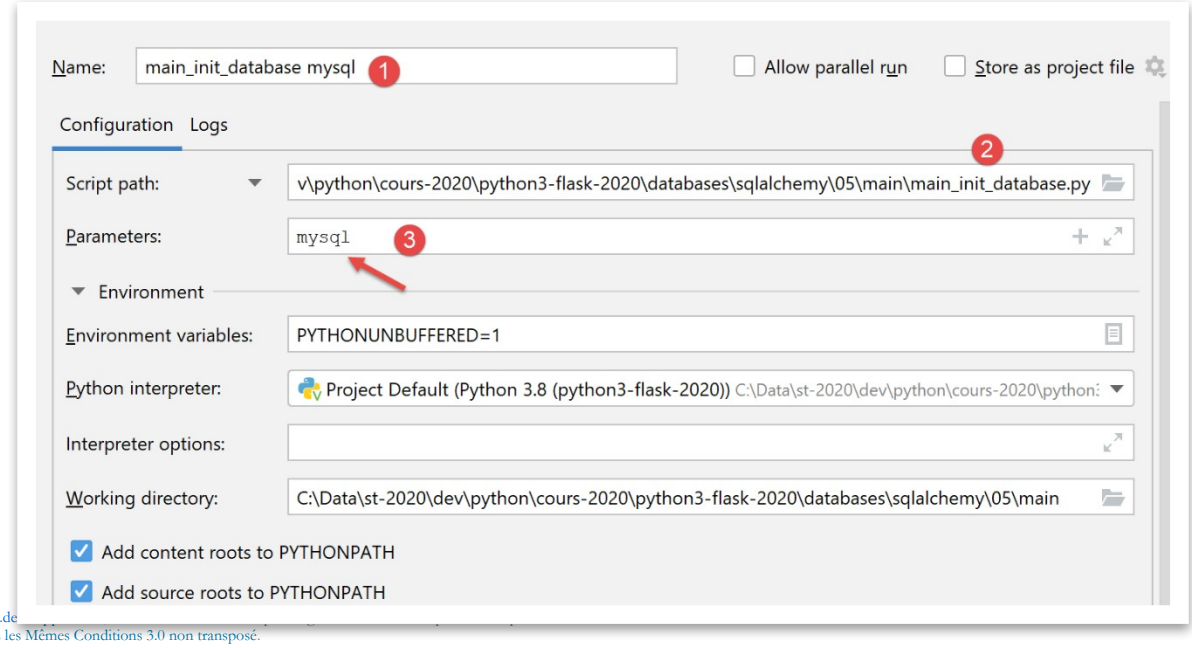
- lignes 1-11 : le script attend un paramètre **[mysql]** ou **[pgres]** selon qu'on veut initialiser une base MySQL ou PostgreSQL ;
- lignes 13-15 : l'application est configurée pour le SGBD passé en paramètre ;
- lignes 20-22 : on récupère les données à mettre en base ;
- ligne 25 : la couche **[dao]** a déjà été instanciée et est accessible dans la configuration de l'application ;
- ligne 30 : la base de données est initialisée ;
- lignes 34-37 : erreur ou pas, on libère les ressources de l'application à l'aide du module **[shutdown]** ;

Le module **[shutdown.py]** est le suivant :

```
1.  def execute(config: dict):
2.      # on libère les ressources mobilisées par l'application
3.      sqlalchemy_session = config['database']['session']
4.      if sqlalchemy_session:
5.          sqlalchemy_session.close()
```

La fonction **[shutdown.execute]** ferme la session **[sqlalchemy]** utilisée pour initialiser la base de données.

Nous créons une 1ère configuration d'exécution (cf. |configuration d'exécution|) pour exécuter **[main_init_database]** avec le SGBD MySQL :

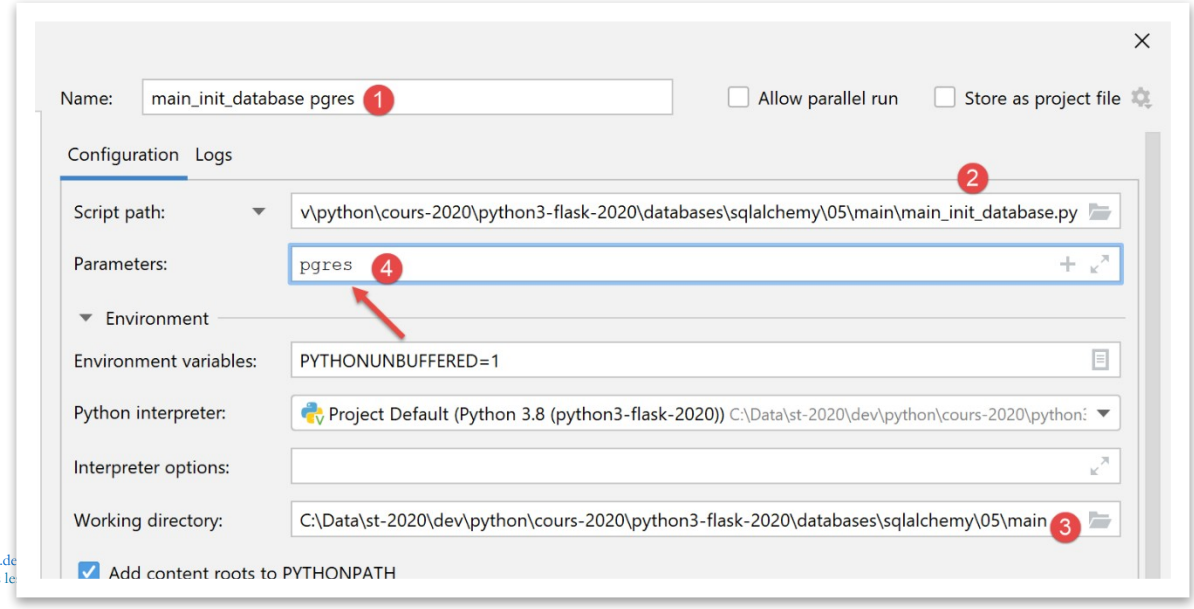


Partage dans les Mêmes Conditions 3.0 non transposé.

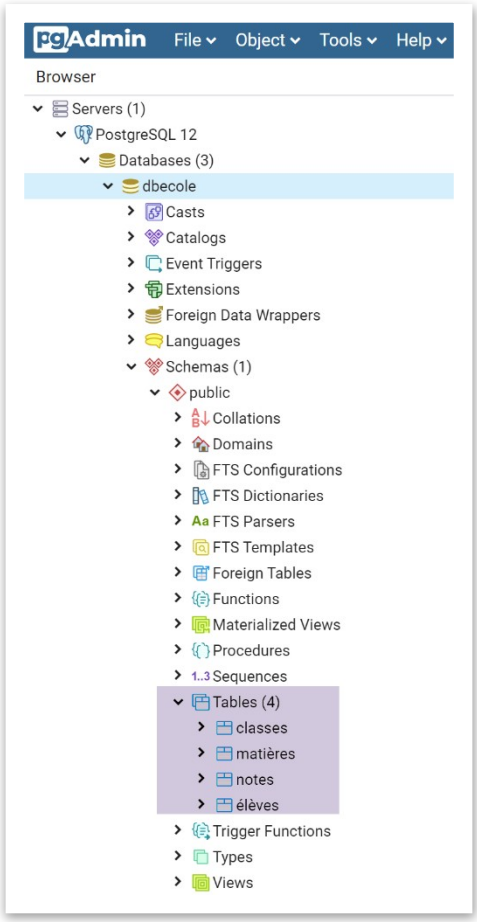
Les résultats de l'exécution de cette configuration sont les suivants dans phpMyAdmin :

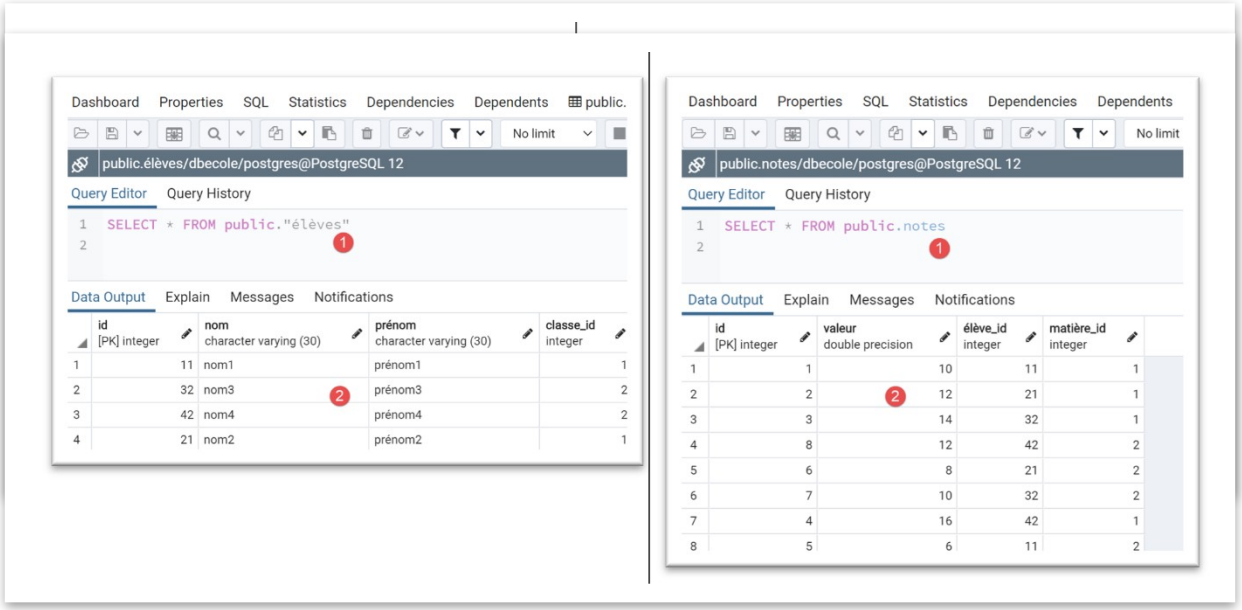


Pour le SGBD **[PostgreSQL]**, nous utilisons la configuration d'exécution suivante :



A l'exécution, les résultats dans **[pgAdmin]** sont les suivants :





On notera la facilité avec laquelle on a pu changer de SGBD.

19.6.7 La couche [dao] – 2

Nous revenons sur la classe **[DatabaseDao]** qui implémente la couche **[dao]**. Nous n'avons montré pour l'instant que l'implémentation de la méthode **[init_database]**. Nous montrons maintenant l'implémentation des autres méthodes :

```python
from sqlalchemy.exc import DatabaseError, IntegrityError, InterfaceError

from Classe import Classe
from Elève import Elève
from InterfaceDatabaseDao import InterfaceDatabaseDao
from Matière import Matière
from MyException import MyException
from Note import Note


class DatabaseDao(InterfaceDatabaseDao):

    def __init__(self, config: dict):
        # database = {"engine": engine, "metadata": metadata, "tables": tables, "session": session}
        self.database = config['database']
        self.session = self.database['session']

    def init_database(self, data: dict):
        …

    # liste de toutes les classes
    def get_classes(self: object) -> list:
        # requête
        return self.session.query(Classe).all()

    # liste de tous les élèves
    def get_élèves(self: object) -> list:
        # requête
        return self.session.query(Elève).all()

    # liste de toutes les matières
    def get_matières(self: object) -> list:
        # requête
        return self.session.query(Matière).all()

    # la liste des notes de tous les élèves
    def get_notes(self: object) -> list:
        # requête
        return self.session.query(Note).all()

    # la liste des notes d'un élève particulier
    def get_notes_for_élève_by_id(self: object, élève_id: int) -> list:
        # on cherche l'élève - une exception est lancée s'il n'existe pas
        # on la laisse remonter
        élève = self.get_élève_by_id(élève_id)
        # on récupère ses notes (lazy loading)
        notes = élève.notes
        # on rend un dictionnaire
        return {"élève": élève, "notes": notes}

    # un élève repéré par son n°
    def get_élève_by_id(self, élève_id: int) -> Elève:
        # on cherche l'élève
        élèves = self.session.query(Elève).filter(Elève.id == élève_id).all()
        # a-t-on trouvé ?
        if élèves:
            return élèves[0]
        else:
            raise MyException(11, f"L'élève d'identifiant {élève_id} n'existe pas")

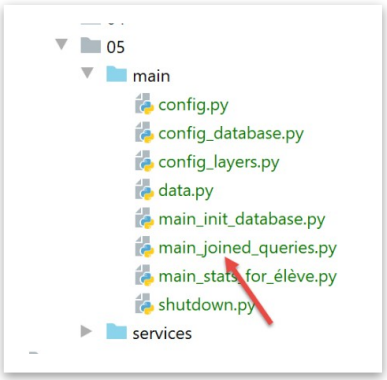
    # un élève repéré par son nom
    def get_élève_by_name(self, élève_name: str) -> Elève:
        # on cherche l'élève
        élèves = self.session.query(Elève).filter(Elève.nom == élève_name).all()
        # a-t-on trouvé ?
        if élèves:
            return élèves[0]
        else:
            raise MyException(12, f"L'élève de nom {élève_name} n'existe pas")

    # une classe repérée par son n°
    def get_classe_by_id(self, classe_id: int) -> Classe:
        # on cherche la classe
        classes = self.session.query(Classe).filter(Classe.id == classe_id).all()
        # a-t-on trouvé ?
        if classes:
```

```
77.             return classes[0]
78.         else:
79.             raise MyException(13, f"La classe d'identifiant {classe_id} n'existe pas")
80.
81.     # une classe repérée par son nom
82.     def get_classe_by_name(self, classe_name: str) -> Classe:
83.         # on cherche l'classe
84.         classes = self.session.query(Classe).filter(Classe.nom == classe_name).all()
85.         # a-t-on trouvé ?
86.         if classes:
87.             return classes[0]
88.         else:
89.             raise MyException(14, f"La classe de nom {classe_name} n'existe pas")
90.
91.     # une matière repérée par son n°
92.     def get_matière_by_id(self, matière_id: int) -> Matière:
93.         # on cherche l'matière
94.         matières = self.session.query(Matière).filter(Matière.id == matière_id).all()
95.         # a-t-on trouvé ?
96.         if matières:
97.             return matières[0]
98.         else:
99.             raise MyException(11, f"La matière d'identifiant {matière_id} n'existe pas")
100.
101.    # une matière repérée par son nom
102.    def get_matière_by_name(self, matière_name: str) -> Matière:
103.        # on cherche la matière
104.        matières = self.session.query(Matière).filter(Matière.nom == matière_name).all()
105.        # a-t-on trouvé ?
106.        if matières:
107.            return matières[0]
108.        else:
109.            raise MyException(15, f"La matière de nom {matière_name} n'existe pas")
```

- lignes 21-24 : la méthode **[get_classes]** doit rendre la liste des classes de l'école. Ligne 20, nous utilisons une requête déjà rencontrée ;
- lignes 26-39 : trois autres méthodes similaires pour obtenir les listes des élèves, des matières et des notes ;
- lignes 51-59 : la méthode **[get_élève_by_id]** doit rendre un élève identifié par son n°. Elle lance une exception si celui-ci n'existe pas ;
  - ligne 54 : on utilise une requêtre filtrée. On obtient une liste vide ou avec un élément ;
  - ligne 57 : si la liste récupérée n'est pas vide on rend le 1er élément de la liste ;
  - sinon ligne 59, on lance une exception ;
- lignes 41-49 : la méthode **[get_notes_for_élève_by_id]** doit rendre les notes d'un élève identifié par son n° :
  - ligne 45, on utilise la méthode **[get_élève_by_id]** pour obtenir l'entité Elève de l'élève ;
  - ligne 47, on utilise la propriété **[Elève.notes]** créée par le mapping entre l'entité **[Note]** et la table **[notes]** (cf. paragraphe |configuration sqlalchemy|) et qui représente les notes de l'élève ;
  - ligne 49 : on rend un dictionnaire ;
- lignes 61-109 : une série de méthodes analogues permettant de :
  - retrouver un élève par son nom, lignes 61-69 ;
  - retrouver une classe, lignes 71-89 ;
  - retrouver une matière, lignes 91-109 ;

19.6.8 Le script [main_joined_queries]



Le script **[main_joined_queries]** s'appelle ainsi parce qu'il vise à mettre en lumière les requêtes faites implicitement par **[sqlalchemy]** pour récupérer des informations appartenant à plusieurs tables. Ces requêtes cachées au programmeur sont faites à chaque fois qu'une propriété d'une entité a été associée à la fonction **[relationship]** dans le mapping de l'entité. Par exemple :

```
1.      # mapping
2.      mapper(Note, tables['notes'], properties={
3.          'id': notes_table.c.id,
4.          'valeur': notes_table.c.valeur,
5.          'élève': relationship(Elève, backref="notes", lazy="select"),
6.          'matière': relationship(Matière, backref="notes", lazy="select")
7.      })
```

Ci-dessus, le mapping entre l'entité **[Note]** et la table **[notes]** :

- ligne 5, lorsque la propriété **[élève]** d'une entité **[Note]** est demandée pour la 1ère fois, elle sera cherchée dans la table **[élèves]** via une requête SQL. Tant que cette propriété n'a pas été demandée, elle reste indéfinie (lazy load). Une fois qu'elle a été obtenue, sa valeur reste en mémoire de l'ORM. Lorsqu'elle sera référencée une deuxième fois, l'ORM délivrera immédiatement sa valeur sans passer par une nouvelle requête SQL. Tout cela est transparent pour le développeur ;
- idem pour la propriété inverse **[Elève.notes]** (backref), ligne 5 ;
- idem pour la propriété **[Note.matière]** et sa propriété inverse **[Matière.notes]** (backref), ligne 6 ;

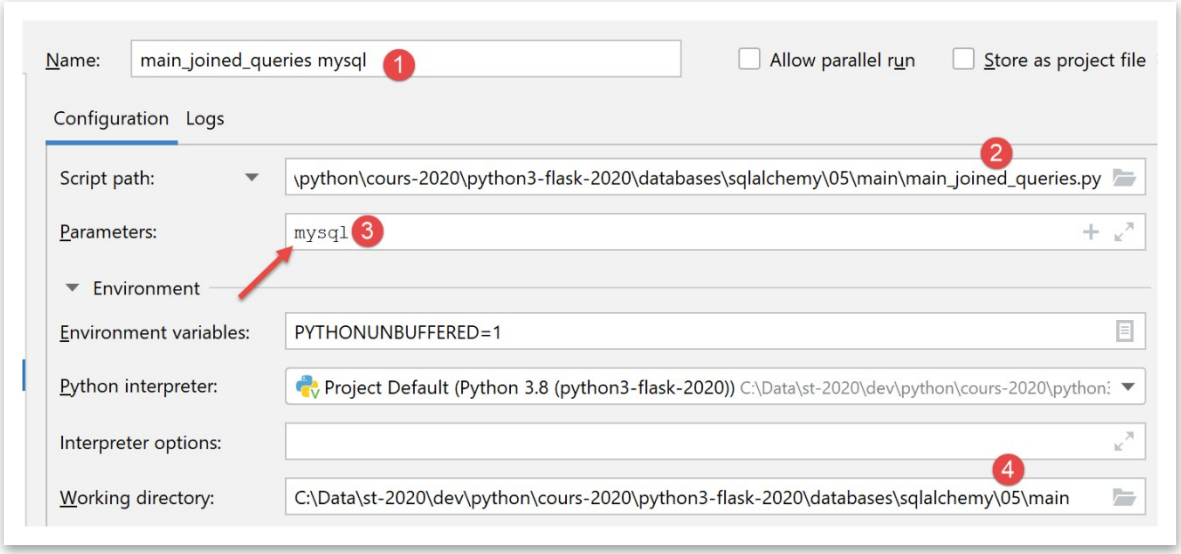
Le script **[main_joined_queries]** est le suivant :

```
1.  # on attend un paramètre mysql ou pgres
2.  import sys
3.
4.  syntaxe = f"{sys.argv[0]} mysql / pgres"
5.  erreur = len(sys.argv) != 2
6.  if not erreur:
7.      sgbd = sys.argv[1].lower()
8.      erreur = sgbd != "mysql" and sgbd != "pgres"
9.  if erreur:
10.     print(f"syntaxe : {syntaxe}")
11.     sys.exit()
12.
13. # on configure l'application
14. import config
15. config = config.configure({"sgbd": sgbd})
16.
17. # le syspath est configuré - on peut faire els imports
18. from MyException import MyException
19.
20. # la couche [dao]
21. dao = config["dao"]
22. try:
23.     # élève by id
24.     print("élève id=11 -----------")
25.     élève = dao.get_élève_by_id(11)
26.     print(f"élève={élève}")
27.     # la classe de l'élève (lazy loading)
28.     classe = élève.classe
29.     print(f"classe de l'élève : {classe}")
30.     # les élèves de la même classe (lazy loading)
31.     print("élèves dans la même classe :")
32.     for élève in classe.élèves:
33.         print(f"élève={élève}")
34.
35.     # un élève par son nom
36.     print("élève nom='nom2' -----------")
37.     print(f"élève={dao.get_élève_by_name('nom2')}")
38.     # sa classe (lazy loading)
39.     print(f"classe de l'élève : {élève.classe}")
40.
41.     # notes d'un élève
42.     print("notes de l'élève id=11 -----------")
43.     # d'abord l'élève
44.     élève = dao.get_élève_by_id(11)
45.     # puis ses notes (lazy loading)
46.     for note in élève.notes:
47.         # la note
48.         print(f"note={note}, "
49.               # la matière de la note (lazy loading)
50.               f"matière={note.matière}")
51.
```

```
52.     # les élèves d'une classe
53.     print("élèves de la classe nom='classe1' -----------")
54.     # d'abord la classe
55.     classe = dao.get_classe_by_name('classe1')
56.     # puis les élèves (lazy loading)
57.     for élève in classe.élèves:
58.         print(élève)
59.
60.     # même chose pour [classe2]
61.     print("élèves de la classe de nom 'classe2' -----------")
62.     classe = dao.get_classe_by_name('classe2')
63.     for élève in classe.élèves:
64.         print(élève)
65.
66.     # les notes dans une matière
67.     print("matière de nom='matière1' -----------")
68.     # d'abord la matière
69.     matière = dao.get_matière_by_name('matière1')
70.     print(f"matière={matière}")
71.     # puis les notes dans cette matière (lazy loading)
72.     print("Notes dans la matière : ")
73.     for note in matière.notes:
74.         print(note)
75.
76.     # même chose pour matière2
77.     print("matière de nom='matière2' -----------")
78.     matière = dao.get_matière_by_name('matière2')
79.     print(f"matière={matière}")
80.     print("Notes dans la matière : ")
81.     for note in matière.notes:
82.         print(f"note={note}")
83. except MyException as ex1:
84.     # on affiche l'erreur
85.     print(f"L'erreur 1 suivante s'est produite : {ex1}")
86. except BaseException as ex2:
87.     # on affiche l'erreur
88.     print(f"L'erreur 2 suivante s'est produite : {ex2}")
89. finally:
90.     # on libère les ressources
91.     import shutdown
92.     shutdown.execute(config)
```

Les commentaires suffisent à comprendre le code.

On crée une configuration d'exécution pour MySQL :



Les résultats de l'exécution sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/sqlalchemy/05/main/
    main_joined_queries.py mysql
2.  élève id=11 -----------
3.  élève={"classe_id": 1, "nom": "nom1", "prénom": "prénom1", "id": 11}
4.  classe de l'élève : {"nom": "classe1", "id": 1}
```

```
5.  élèves dans la même classe :
6.  élève={"classe_id": 1, "nom": "nom1", "prénom": "prénom1", "id": 11}
7.  élève={"classe_id": 1, "nom": "nom2", "prénom": "prénom2", "id": 21}
8.  élève nom='nom2' -----------
9.  élève={"classe_id": 1, "nom": "nom2", "prénom": "prénom2", "id": 21}
10. classe de l'élève : {"nom": "classe1", "id": 1}
11. notes de l'élève id=11 -----------
12. note={"matière_id": 1, "valeur": 10.0, "élève_id": 11, "id": 1}, matière={"coefficient": 1.0, "nom":
    "matière1", "id": 1}
13. note={"matière_id": 2, "valeur": 6.0, "élève_id": 11, "id": 5}, matière={"coefficient": 2.0, "nom":
    "matière2", "id": 2}
14. élèves de la classe nom='classe1' -----------
15. {"classe_id": 1, "nom": "nom1", "prénom": "prénom1", "id": 11}
16. {"classe_id": 1, "nom": "nom2", "prénom": "prénom2", "id": 21}
17. élèves de la classe de nom 'classe2' -----------
18. {"classe_id": 2, "nom": "nom3", "prénom": "prénom3", "id": 32}
19. {"classe_id": 2, "nom": "nom4", "prénom": "prénom4", "id": 42}
20. matière de nom='matière1' -----------
21. matière={"coefficient": 1.0, "nom": "matière1", "id": 1}
22. Notes dans la matière :
23. {"matière_id": 1, "valeur": 10.0, "élève_id": 11, "id": 1}
24. {"matière_id": 1, "valeur": 12.0, "élève_id": 21, "id": 2}
25. {"matière_id": 1, "valeur": 14.0, "élève_id": 32, "id": 3}
26. {"matière_id": 1, "valeur": 16.0, "élève_id": 42, "id": 4}
27. matière de nom='matière2' -----------
28. matière={"coefficient": 2.0, "nom": "matière2", "id": 2}
29. Notes dans la matière :
30. note={"matière_id": 2, "valeur": 6.0, "élève_id": 11, "id": 5}
31. note={"matière_id": 2, "valeur": 8.0, "élève_id": 21, "id": 6}
32. note={"matière_id": 2, "valeur": 10.0, "élève_id": 32, "id": 7}
33. note={"matière_id": 2, "valeur": 12.0, "élève_id": 42, "id": 8}
34.
35. Process finished with exit code 0
```

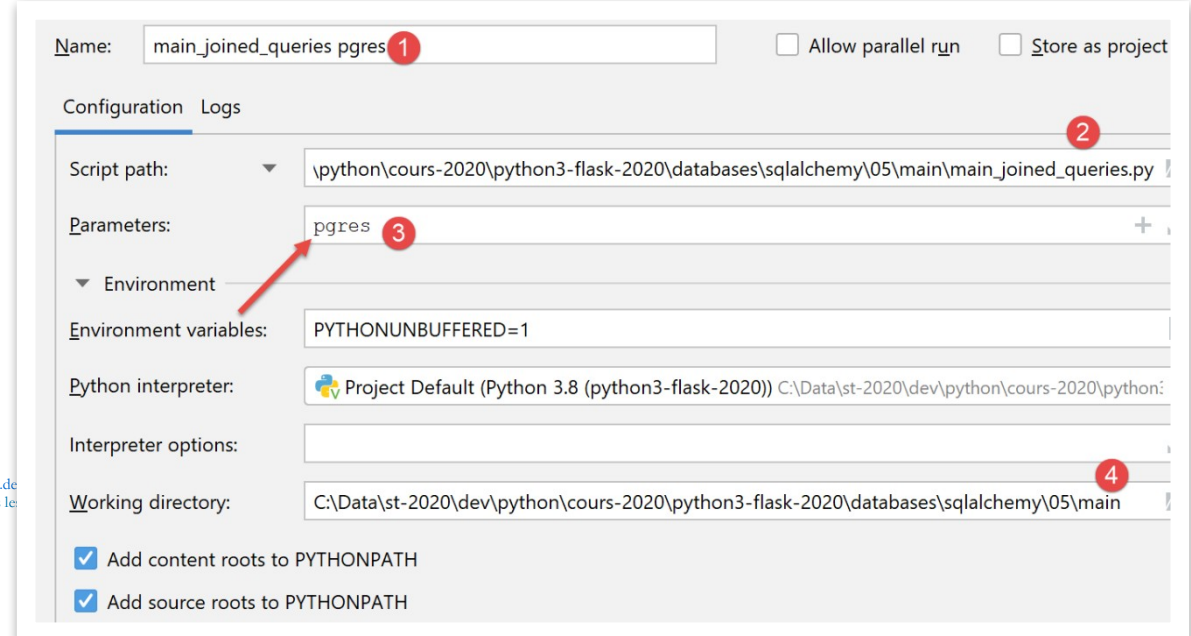
Pour comprendre ces résultats, il faut se rappeler qu'on a exclu certaines propriétés du dictionnaire des entités (cf |configuration|) :

```
1.      # configuration des entités [BaseEntity]
2.      Elève.excluded_keys = ['_sa_instance_state', 'notes', 'classe']
3.      Classe.excluded_keys = ['_sa_instance_state', 'élèves']
4.      Matière.excluded_keys = ['_sa_instance_state', 'notes']
5.      Note.excluded_keys = ['_sa_instance_state', 'matière', 'élève']
```

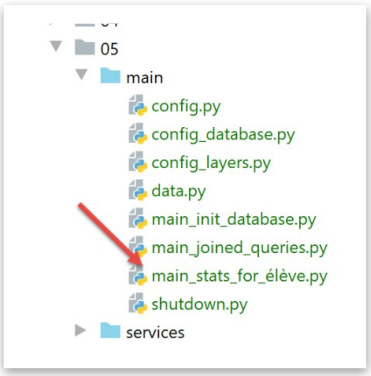
Ainsi, lorsqu'on écrit **[print(f"élève={élève}")]** ligne 26 du code, la ligne 1 ci-dessus nous dit que les propriétés **['_sa_instance_state', 'notes', 'classe']** ne seront pas affichées. C'est ce qu'on voit ligne 3 des résultats. Toutes les autres propriétés sont affichées. Ainsi, toujours ligne 3, on découvre une nouvelle propriété **[classe_id]** qui n'existait initialement pas dans l'entité **[Elève]**. Cette propriété correspond directement à la colonne **[classe_id]** de la table **[élèves]**. Ainsi **[sqlalchemy]** a jouté les propriétés suivantes à l'entité **[Elève]** : **[classe_id, _sa_instance_state, notes]**. Il faut en avoir conscience, notamment parce qu'elles ne doivent pas déjà exister dans l'entité mappée.

Les propriétés exclues du dictionnaire des entités sont importantes. Si par exemple, on n'exclut pas les propriétés **[notes, élève]** de l'entité **[Elève]** alors l'opération **[print(f"élève={élève}")]** va les afficher et va donc, comme il vient d'être expliqué, provoquer des requêtes SQL implicites (lazy loading) pour récupérer les valeurs de ces propriétés. Si, comme ici, c'est une liste d'élèves qui est affichée, les opérations SQL implicites sont faites pour chaque élève. Cela peut être d'une part inutile et d'autre part sûrement coûteux en temps d'exécution.

Pour exécuter le script avec une base PostgreSQL, on crée la configuration d'exécution suivante :

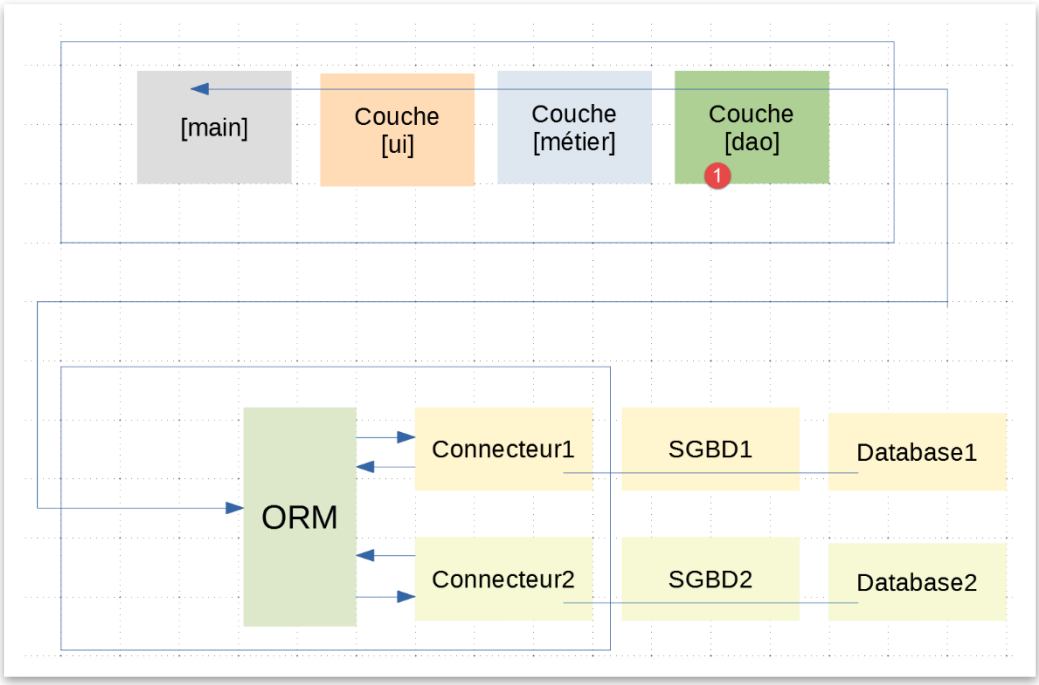


L'exécution donne les mêmes résultats qu'avec MySQL.



19.6.9 Le script `[main_stats_for_élève]`

Le script **`[main_stats_for_élève]`** est celui déjà utilisé dans l'application |troiscouches v01]. Il s'appelait alors **`[main]`**. C'est une application console permettant d'obtenir certains indicateurs sur les notes d'un élève : **`[moyenne pondérée, min, max, liste]`**. Il s'insère dans l'architecture suivante :



Dans cette architecture en couches, seule la couche **`[dao]`** a été changée entre l'application |troiscouches v01| et celle-ci. Comme la nouvelle couche **`[dao]`** respecte l'interface **`[InterfaceDao]`** de l'ancienne couche **`[dao]`**, les couches **`[ui, métier]`** n'ont pas à être changées. On peut donc continuer à utiliser celles définies dans l'application |troiscouches v01|.

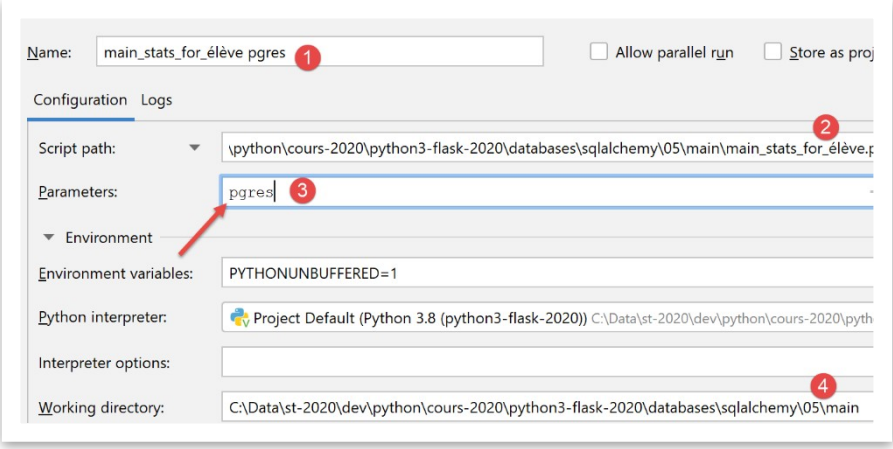
Le script **`[main_stats_for_élève]`** implémente la couche **`[main]`** du schéma ci-dessus de la façon suivante :

```
1.  # on attend un paramètre mysql ou pgres
2.  import sys
3.
4.  syntaxe = f"{sys.argv[0]} mysql / pgres"
5.  erreur = len(sys.argv) != 2
6.  if not erreur:
7.      sgbd = sys.argv[1].lower()
8.      erreur = sgbd != "mysql" and sgbd != "pgres"
9.  if erreur:
10.     print(f"syntaxe : {syntaxe}")
11.     sys.exit()
12.
```

```
13. # on configure l'application
14. import config
15. config = config.configure({'sgbd': sgbd})
16.
17. # le syspath est configuré - on peut faire les imports
18. from MyException import MyException
19.
20. # la couche [ui]
21. ui = config["ui"]
22. try:
23.     # exécution couche [ui]
24.     ui.run()
25. except MyException as ex1:
26.     # on affiche l'erreur
27.     print(f"L'erreur 1 suivante s'est produite : {ex1}")
28. except BaseException as ex2:
29.     # on affiche l'erreur
30.     print(f"L'erreur 2 suivante s'est produite : {ex2}")
31. finally:
32.     # on libère les ressources
33.     import shutdown
34.     shutdown.execute(config)
```

- ligne 20 : on récupère une référence sur la couche **[ui]** dans la configuration de l'application ;
- ligne 24 : on lance le dialogue avec l'utilisateur à l'aide de l'unique méthode de la couche **[ui]** ;

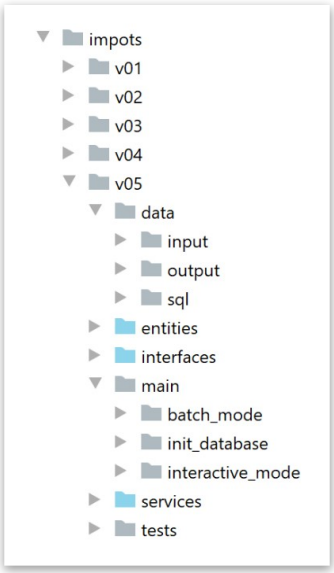
Une configuration d'exécution pour PostgreSQL serait la suivante :



Voici un exemple d'exécution avec cette configuration :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/databases/sqlalchemy/05/main/
    main_stats_for_élève.py pgres
2.  Numéro de l'élève (>=1 et * pour arrêter) : 11
3.  Elève={"prénom": "prénom1", "id": 11, "classe_id": 1, "nom": "nom1"}, notes=[10.0 6.0], max=10.0, min=6.0,
    moyenne pondérée=7.33
4.  Numéro de l'élève (>=1 et * pour arrêter) : 1
5.  L'erreur suivante s'est produite : MyException[11, L'élève d'identifiant 1 n'existe pas]
6.  Numéro de l'élève (>=1 et * pour arrêter) : *
7.
8.  Process finished with exit code 0
```

20 Exercice d'application : version 5

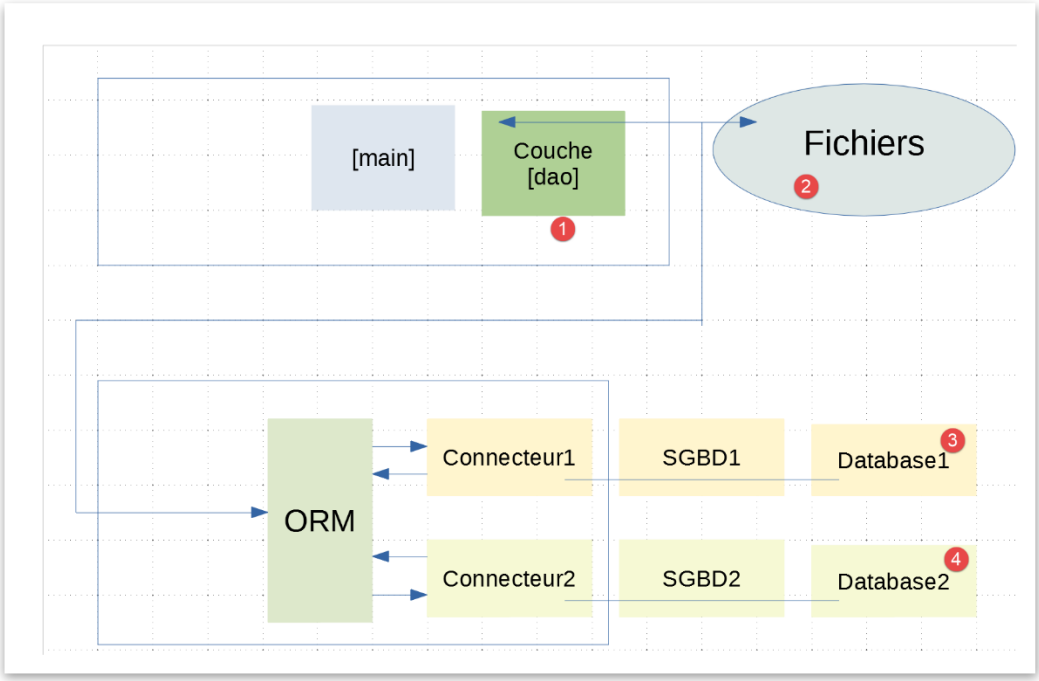


Nous allons développer trois applications :

- l'application 1 initialisera la base de données qui va venir remplacer le fichier `[admindata.json]` de la version 4 ;
- l'application 2 fera le calcul des impôts en mode batch ;
- l'application 3 fera le calcul des impôts en mode interactif ;

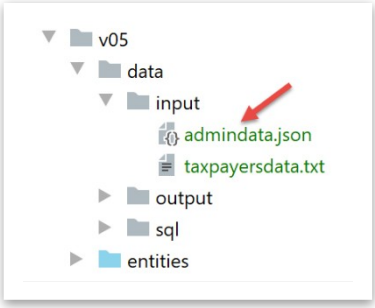
20.1 Application 1 : initialisation de la base de données

L'application 1 aura l'architecture suivante :



C'est une évolution de l'architecture de la version 4 (paragraphe | Version 4 |) : les données fiscales seront trouvées dans une base de données au lieu d'être dans un fichier jSON. La couche `[dao]` va évoluer pour implémenter ce changement.

20.1.1 Le fichier `[admindata.json]`



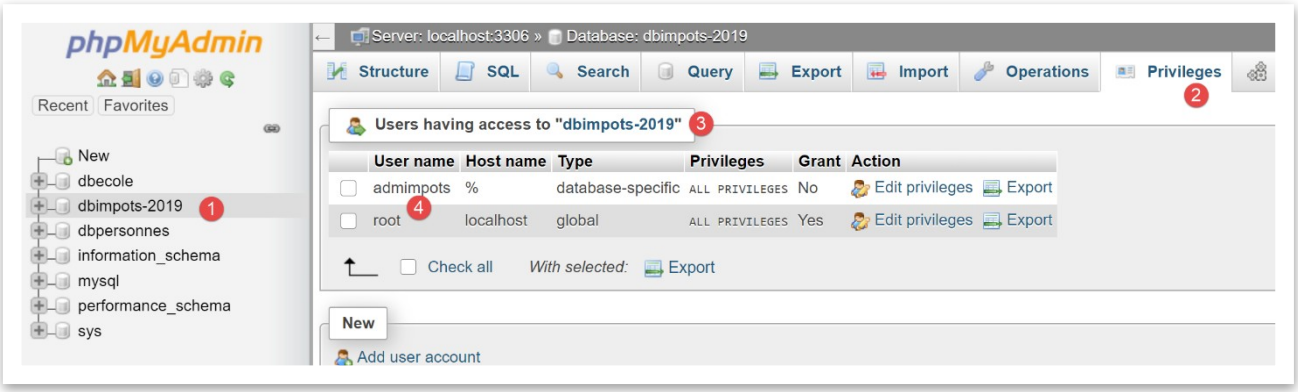
Le fichier **`[admindata.json]`** est celui qu'il était dans la version 4 :

```
{
    "limites": [9964, 27519, 73779, 156244, 0],
    "coeffr": [0, 0.14, 0.3, 0.41, 0.45],
    "coeffn": [0, 1394.96, 5798, 13913.69, 20163.45],
    "plafond_qf_demi_part": 1551,
    "plafond_revenus_celibataire_pour_reduction": 21037,
    "plafond_revenus_couple_pour_reduction": 42074,
    "valeur_reduc_demi_part": 3797,
    "plafond_decote_celibataire": 1196,
    "plafond_decote_couple": 1970,
    "plafond_impot_couple_pour_decote": 2627,
    "plafond_impot_celibataire_pour_decote": 1595,
    "abattement_dixpourcent_max": 12502,
    "abattement_dixpourcent_min": 437
}
```

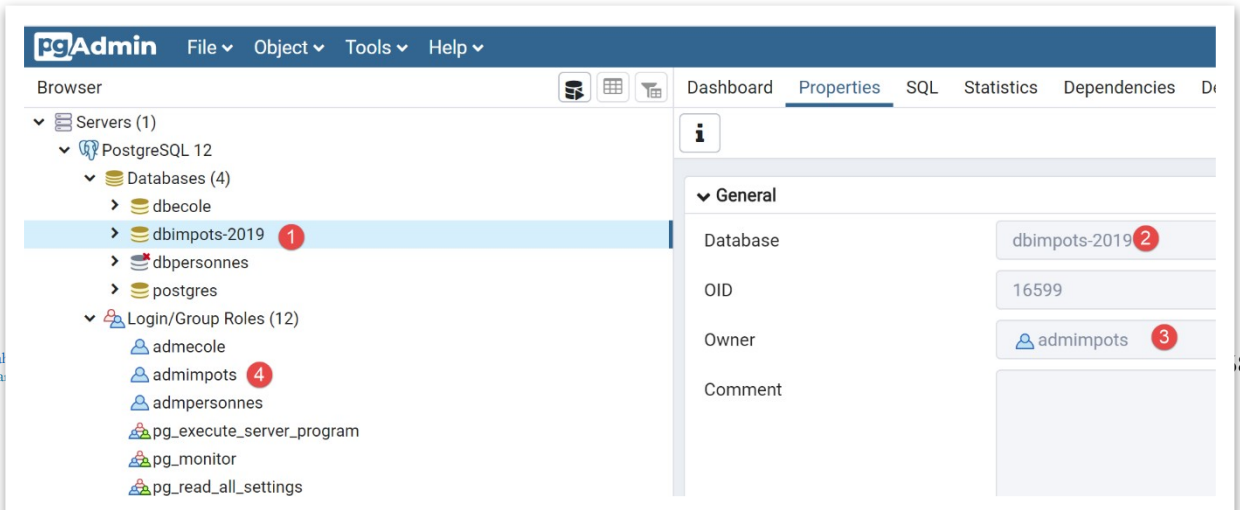
Nous allons utiliser comme colonnes de la base de données les clés de ce dictionnaire.

20.1.2 Création des bases de données

Comme il a été montré au paragraphe |création d'une base de données MySQL|, nous créons une base de données MySQL nommée **`[dbimpots-2019]`** propriété de l'utilisateur **`[admimpots]`** de mot de passe **`[mdpimpots]`**. Dans **`[phpMyAdmin]`** cela donne la chose suivante :



De même, comme il a été montré au paragraphe |création d'une base de données PostgreSQL|, nous créons une base de données PostgreSQL nommée **`[dbimpots-2019]`** propriété de l'utilisateur **`[admimpots]`** de mot de passe **`[mdpimpots]`**. Dans **`[pgAdmin]`** cela donne la chose suivante :



Les bases sont créées mais pour l'instant elles n'ont aucune table. Celles-ci vont être construites par l'ORM **[sqlalchemy]**.

20.1.3 Les entitées mappées par [sqlalchemy]

Nous allons créer deux tables pour encapsuler les données de **[admindata.json]** :

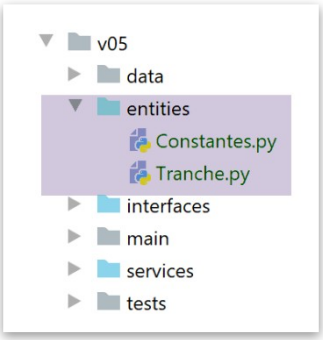
Définie par **[sqlalchemy]** la table **[tbtranches]** rassemblera les données des tableaux **[limites, coeffr, coeffn]** du dictionnaire **[admindata.json]** :

```
1.      # la table des tranches de l'impôt
2.      tranches_table = Table("tbtranches", metadata,
3.                             Column('id', Integer, primary_key=True),
4.                             Column('limite', Float, nullable=False),
5.                             Column('coeffr', Float, nullable=False),
6.                             Column('coeffn', Float, nullable=False)
7.                             )
```

Définie par **[sqlalchemy]** la table **[tbconstantes]** rassemblera les constantes du dictionnaire **[admindata.json]** :

```
1.      # la table des constantes
2.      constantes_table = Table("tbconstantes", metadata,
3.                               Column('id', Integer, primary_key=True),
4.                               Column('plafond_qf_demi_part', Float, nullable=False),
5.                               Column('plafond_revenus_celibataire_pour_reduction', Float, nullable=False),
6.                               Column('plafond_revenus_couple_pour_reduction', Float, nullable=False),
7.                               Column('valeur_reduc_demi_part', Float, nullable=False),
8.                               Column('plafond_decote_celibataire', Float, nullable=False),
9.                               Column('plafond_decote_couple', Float, nullable=False),
10.                              Column('plafond_impot_celibataire_pour_decote', Float, nullable=False),
11.                              Column('plafond_impot_couple_pour_decote', Float, nullable=False),
12.                              Column('abattement_dixpourcent_max', Float, nullable=False),
13.                              Column('abattement_dixpourcent_min', Float, nullable=False)
14.                              )
```

Les entités qui seront mappées avec ces deux tables seront les suivantes :



L'entité **[Constantes]** encapsule les constantes du dictionnaire **[admindata.json]** :

```
1.  from BaseEntity import BaseEntity
2.
3.
4.  # classe conteneur des données de l'administration fiscale
5.  class Constantes(BaseEntity):
6.      # clés exclues de l'état de la classe
7.      excluded_keys = ["_sa_instance_state"]
8.
9.      # clés autorisées
10.     @staticmethod
11.     def get_allowed_keys() -> list:
12.         return ["id",
13.                 "plafond_qf_demi_part",
14.                 "plafond_revenus_celibataire_pour_reduction",
15.                 "plafond_revenus_couple_pour_reduction",
16.                 "valeur_reduc_demi_part",
17.                 "plafond_decote_celibataire",
18.                 "plafond_decote_couple",
19.                 "plafond_decote_couple",
20.                 "plafond_impot_celibataire_pour_decote",
21.                 "plafond_impot_couple_pour_decote",
22.                 "abattement_dixpourcent_max",
```

```
23.                 "abattement_dixpourcent_min"]
```

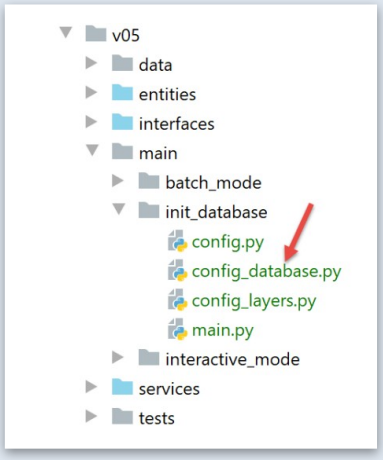
- ligne 5 : la classe **[Constantes]** étend la classe **[BaseEntity]** ;
- ligne 7 : par mapping **[sqlalchemy]**, la classe **[Constante]** va recevoir la propriété **[_sa_instance_state]**. Nous l'excluons du dictionnaire **[asdict]** de l'entité ;
- lignes 11-23 : les propriétés de l'entité. On a repris les noms utilisés dans le dictionnaire **[admindata.json]** pour faciliter l'écriture du code ;

L'entité **[Tranche]** encapsule une ligne des trois tableaux **[limites, coeffr, coeffn]** du dictionnaire **[admindata.json]** :

```
1.  from BaseEntity import BaseEntity
2.
3.
4.  # classe conteneur des données de l'administration fiscale
5.  class Tranche(BaseEntity):
6.      # clés exclues de l'état de la classe
7.      excluded_keys = ["_sa_instance_state"]
8.
9.      # clés autorisées
10.     @staticmethod
11.     def get_allowed_keys() -> list:
12.         return ["id", "limite", "coeffr", "coeffn"]
```

- ligne 5 : la classe **[Tranche]** étend la classe **[BaseEntity]** ;
- ligne 7 : on exclut des propriétés du dictionnaire **[asdict]** de l'entité, la propriété **[_sa_instance_state]** ajoutée par **[sqlalchemy]** ;
- lignes 10-12 : les propriétés de la classe ;

Le mapping entre les entités **[Constantes, Tranche]** et les tables **[constantes, tranches]** sera le suivant :

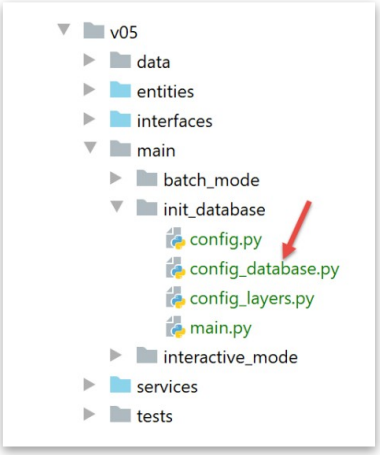



```
1.  …
2.      # la table des constantes
3.      constantes_table = Table("tbconstantes", metadata,
4.                               Column('id', Integer, primary_key=True),
5.                               Column('plafond_qf_demi_part', Float, nullable=False),
6.                               Column('plafond_revenus_celibataire_pour_reduction', Float, nullable=False),
7.                               Column('plafond_revenus_couple_pour_reduction', Float, nullable=False),
8.                               Column('valeur_reduc_demi_part', Float, nullable=False),
9.                               Column('plafond_decote_celibataire', Float, nullable=False),
10.                              Column('plafond_decote_couple', Float, nullable=False),
11.                              Column('plafond_impot_celibataire_pour_decote', Float, nullable=False),
12.                              Column('plafond_impot_couple_pour_decote', Float, nullable=False),
13.                              Column('abattement_dixpourcent_max', Float, nullable=False),
14.                              Column('abattement_dixpourcent_min', Float, nullable=False)
15.                              )
16.
17.     # la table des tranches de l'impôt
18.     tranches_table = Table("tbtranches", metadata,
19.                            Column('id', Integer, primary_key=True),
20.                            Column('limite', Float, nullable=False),
21.                            Column('coeffr', Float, nullable=False),
22.                            Column('coeffn', Float, nullable=False)
23.                            )
24.     # les mappings
25.     from Tranche import Tranche
26.     mapper(Tranche, tranches_table)
```

```
27.
28.     from Constantes import Constantes
29.     mapper(Constantes, constantes_table)
```

- les mappings sont faits aux lignes 24-29. On y a omis de faire les correspondances entre propriétés des entités mappées et tables de la base de données. C'est possible lorsque les noms des colonnes des tables sont les mêmes que ceux des propriétés auxquelles elles doivent être associées. Pour cette raison, nous avons repris dans les tables les noms des propriétés des entités mappées. Cela facilite l'écriture du code et sa compréhension ;

20.1.4 Le fichier de configuration de `[sqlalchemy]`



Nous venons de détailler une partie de la configuration de **`[sqlalchemy]`**. Le fichier **`[config_database]`** dans sa totalité est le suivant :

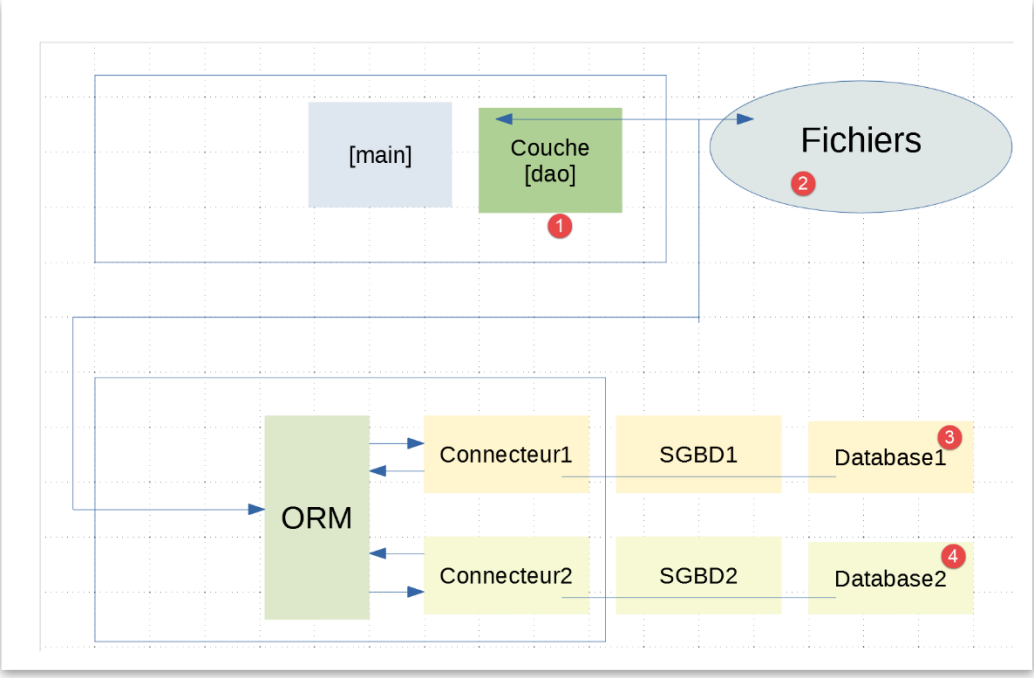
```
1.  def configure(config: dict) -> dict:
2.      # configuration sqlalchemy
3.      from sqlalchemy import create_engine, Table, Column, Integer, MetaData, Float
4.      from sqlalchemy.orm import mapper, sessionmaker
5.
6.      # chaînes de connexion aux bases de données exploitées
7.      connection_strings = {
8.          'mysql': "mysql+mysqlconnector://admimpots:mdpimpots@localhost/dbimpots-2019",
9.          'pgres': "postgresql+psycopg2://admimpots:mdpimpots@localhost/dbimpots-2019"
10.     }
11.     # chaîne de connexion à la base de données exploitée
12.     engine = create_engine(connection_strings[config['sgbd']])
13.
14.     # metadata
15.     metadata = MetaData()
16.
17.     # la table des constantes
18.     constantes_table = Table("tbconstantes", metadata,
19.                              Column('id', Integer, primary_key=True),
20.                              Column('plafond_qf_demi_part', Float, nullable=False),
21.                              Column('plafond_revenus_celibataire_pour_reduction', Float, nullable=False),
22.                              Column('plafond_revenus_couple_pour_reduction', Float, nullable=False),
23.                              Column('valeur_reduc_demi_part', Float, nullable=False),
24.                              Column('plafond_decote_celibataire', Float, nullable=False),
25.                              Column('plafond_decote_couple', Float, nullable=False),
26.                              Column('plafond_impot_celibataire_pour_decote', Float, nullable=False),
27.                              Column('plafond_impot_couple_pour_decote', Float, nullable=False),
28.                              Column('abattement_dixpourcent_max', Float, nullable=False),
29.                              Column('abattement_dixpourcent_min', Float, nullable=False)
30.                              )
31.
32.     # la table des tranches de l'impôt
33.     tranches_table = Table("tbtranches", metadata,
34.                            Column('id', Integer, primary_key=True),
35.                            Column('limite', Float, nullable=False),
36.                            Column('coeffr', Float, nullable=False),
37.                            Column('coeffn', Float, nullable=False)
38.                            )
39.     # les mappings
40.     from Tranche import Tranche
41.     mapper(Tranche, tranches_table)
```

```
42.
43.     from Constantes import Constantes
44.     mapper(Constantes, constantes_table)
45.
46.     # la session factory
47.     session_factory = sessionmaker()
48.     session_factory.configure(bind=engine)
49.
50.     # une session
51.     session = session_factory()
52.
53.     # on enregistre certaines informations
54.     config['database'] = {"engine": engine, "metadata": metadata, "tranches_table": tranches_table,
55.                           "constantes_table": constantes_table, "session": session}
56.
57.     # résultat
58.     return config
```

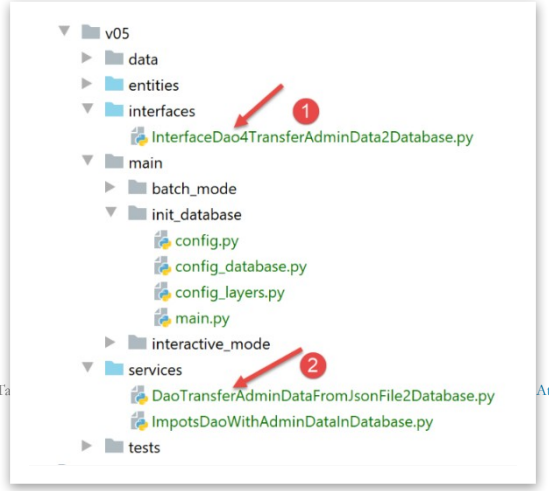
- ligne 1 : la fonction **[configure]** reçoit en paramètre un dictionnaire dont la clé **[sgbd]** lui dit quel SGBD utiliser : MySQL (mysql) ou PostgreSQL (pgres) ;
- lignes 6-12 : on sélectionne la base de données demandée par la configuration ;
- lignes 14-44 : mappings entités / tables. Ces mappings sont simples car il n'existe aucun lien entre les tables **[tranches]** et **[constantes]**. Elles sont indépendantes. Il n'y a donc pas de clé étrangère de l'une sur l'autre à gérer ;
- lignes 46-51 : on crée la session **[session]** de travail de l'application ;
- lignes 53-58 : les informations utiles sont mises dans le dictionnaire de la configuration et celui-ci est retourné ;

20.1.5 La couche [dao]

Revenons à l'architecture de l'application 1 à construire :



La couche **[dao]** **[1]** doit lire le fichier **[admindata.json]** **[2]** et transférer son contenu dans une des bases **[3, 4]** ;



La couche **[dao]** présente l'interface **[1]** et est implémentée par la classe **[2]**.

L'interface **[InterfaceDao4TransferAdminData2Database]** est la suivante :

```
1.  # imports
2.  from abc import ABC, abstractmethod
3.
4.
5.  # interface InterfaceImpôtsUI
6.  class InterfaceDao4TransferAdminData2Database(ABC):
7.      # transfert des données fiscales dans une base de données
8.      @abstractmethod
9.      def transfer_admindata_in_database(self:object):
10.         pass
```

- lignes 8-10 : l'interface ne présente qu'une méthode **[transfer_admindata_in_database]** sans paramètres. Comme cette méthode a besoin de paramètres (quel fichier ?, quelle base de données ?), cela signifie que ceux-ci seront passés au constructeur des classes implémentant cette interface ;

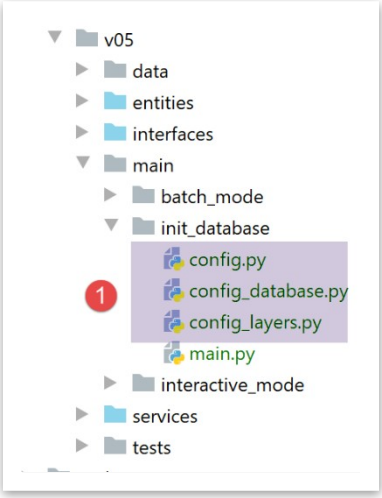
La classe **[DaoTransferAdminDataFromJsonFile2Database]** implémente l'interface **[InterfaceDao4TransferAdminData2Database]** de la façon suivante :

```
1.  # imports
2.  import codecs
3.  import json
4.
5.  from sqlalchemy.exc import DatabaseError, IntegrityError, InterfaceError
6.
7.  from Constantes import Constantes
8.  from ImpôtsError import ImpôtsError
9.  from InterfaceDao4TransferAdminData2Database import InterfaceDao4TransferAdminData2Database
10. from Tranche import Tranche
11.
12.
13. class DaoTransferAdminDataFromJsonFile2Database(InterfaceDao4TransferAdminData2Database):
14.
15.     # constructeur
16.     def __init__(self, config: dict):
17.         self.config = config
18.
19.     # transfert
20.     def transfer_admindata_in_database(self) -> None:
21.         # initialisations
22.         session = None
23.         config = self.config
24.
25.         try:
26.             # on récupère les données de l'administration fiscale
27.             with codecs.open(config["admindataFilename"], "r", "utf8") as fd:
28.                 # transfert du contenu dans un dictionnaire
29.                 admindata = json.load(fd)
30.
31.             # on récupère la configuration de la base de données
32.             database = config["database"]
33.
34.             # suppression des deux tables de la base de données
35.             # checkfirst=True : vérifie d'abord que la table existe
36.             database["tranches_table"].drop(database["engine"], checkfirst=True)
37.             database["constantes_table"].drop(database["engine"], checkfirst=True)
38.
39.             # recréation des tables à partir des mappings
40.             database["metadata"].create_all(database["engine"])
41.
42.             # la session [sqlalchemy] courante
43.             session = database["session"]
44.
45.             # on remplit la table des tranches de l'impôt
46.             limites = admindata["limites"]
47.             coeffr = admindata["coeffr"]
48.             coeffn = admindata["coeffn"]
49.             for i in range(len(limites)):
50.                 session.add(Tranche().fromdict(
51.                     {"limite": limites[i], "coeffr": coeffr[i], "coeffn": coeffn[i]}))
52.             # on remplit la table des constantes
53.             session.add(Constantes().fromdict({
54.                 'plafond_qf_demi_part': admindata["plafond_qf_demi_part"],
55.                 'plafond_revenus_celibataire_pour_reduction':
    admindata["plafond_revenus_celibataire_pour_reduction"],
56.                 'plafond_revenus_couple_pour_reduction': admindata["plafond_revenus_couple_pour_reduction"],
57.                 'valeur_reduc_demi_part': admindata["valeur_reduc_demi_part"],
58.                 'plafond_decote_celibataire': admindata["plafond_decote_celibataire"],
59.                 'plafond_decote_couple': admindata["plafond_decote_couple"],
```

```
60.                 'plafond_impot_celibataire_pour_decote': admindata["plafond_impot_celibataire_pour_decote"],
61.                 'plafond_impot_couple_pour_decote': admindata["plafond_impot_couple_pour_decote"],
62.                 'abattement_dixpourcent_max': admindata["abattement_dixpourcent_max"],
63.                 'abattement_dixpourcent_min': admindata["abattement_dixpourcent_min"]
64.             }))
65.
66.             # validation de la session [sqlalchemy]
67.             session.commit()
68.         except (IntegrityError, DatabaseError, InterfaceError) as erreur:
69.             # on relance l'exception sous une autre forme
70.             raise ImpôtsError(17, f"{erreur}")
71.         finally:
72.             # on libère les ressources de la session
73.             if session:
74.                 session.close()
```

- ligne 13 : la classe **[DaoTransferAdminDataFromJsonFile2Database]** implémente l'interface **[InterfaceDao4TransferAdminData2Database]** ;
- lignes 15-17 : le constructeur de la classe reçoit en paramètre le dictionnaire de la configuration. Les clés suivantes vont être utilisées :
  - **[admindataFilename]** (ligne 27) : le nom du fichier jSON contenant les données de l'administration fiscale à transférer en base ;
  - **[database]** ligne 32 : le configuration **[sqlalchemy]** de l'application ;
- lignes 34-37 : suppression des tables **[constantes]** et **[tranches]** si elles existent ;
- lignes 39-40 : recréation des deux tables ;
- ligne 43 : on récupère la session **[sqlalchemy]** présente dans la configuration ;
- lignes 45-51 : les tableaux **[limites, coeffr, coeffn]** du dictionnaire **[admindata]** sont mis dans la session. Pour cela on met dans la session des instances de l'entité **[Tranche]** ;
- lignes 52-64 : une instance de l'entité **[Constantes]** est mise en session ;
- lignes 66-67 : la session est validée. Si les données de la session n'étaient pas encore en base, elles y sont mises à ce moment là ;
- lignes 68-70 : gestion d'une éventuelle erreur ;
- lignes 71-74 : la session est fermée. On peut le faire car la couche **[dao]** n'est utilisée qu'une fois ;

20.1.6 Configuration de l'application



L'application est configurée par trois fichiers **[1]** :

- **[config]** est le fichier de configuration générale. C'est lui qui configure l'application **[main]**. Il se fait aider par les deux autres fichiers :
  - **[config_database]** que nous avons étudié et qui configure l'ORM **[sqlalchemy]** ;
  - **[config_layers]** qui configure les couches de l'application ;

Le fichier **[config]** est le suivant :

```
1.  def configure(config: dict) -> dict:
2.      # [config] a la clé [sgbd] qui vaut:
3.      # [mysql] pour gérer une base MySQL
4.      # [pgres] pour gérer une base PostgreSQL
5.
```

```
6.      import os
7.
8.      # étape 1 ---
9.      # on établit le Python Path de l'application
10.
11.     # chemin absolu du dossier de ce script
12.     script_dir = os.path.dirname(os.path.abspath(__file__))
13.
14.     # root_dir (à changer éventuellement)
15.     root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
16.
17.     # chemins absolus des dépendances
18.     absolute_dependencies = [
19.         # InterfaceImpôtsDao, InterfaceImpôtsMétier, InterfaceImpôtsUi
20.         f"{root_dir}/impots/v04/interfaces",
21.         # AbstractImpôtsDao, ImpôtsConsole, ImpôtsMétier
22.         f"{root_dir}/impots/v04/services",
23.         # AdminData, ImpôtsError, TaxPayer
24.         f"{root_dir}/impots/v04/entities",
25.         # BaseEntity, MyException
26.         f"{root_dir}/classes/02/entities",
27.         # dossiers locaux
28.         f"{script_dir}",
29.         f"{script_dir}/../../interfaces",
30.         f"{script_dir}/../../services",
31.         f"{script_dir}/../../entities",
32.     ]
33.
34.     # on fixe le syspath
35.     from myutils import set_syspath
36.     set_syspath(absolute_dependencies)
37.
38.     # étape 2 ------
39.     # on complète la configuration de l'application
40.     config.update({
41.         # chemins absolus des fichiers de données
42.         "admindataFilename": f"{script_dir}/../../data/input/admindata.json"
43.     })
44.
45.     # étape 3 ------
46.     # configuration base de données
47.     import config_database
48.     config = config_database.configure(config)
49.
50.     # étape 4 ------
51.     # instanciation des couches de l'application
52.     import config_layers
53.     config = config_layers.configure(config)
54.
55.     # on rend la config
56.     return config
```

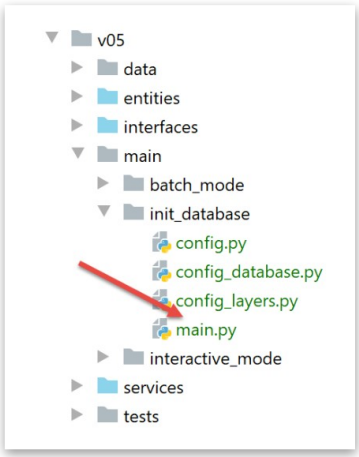
- lignes 8-36 : on construit le Python Path de l'application ;
- lignes 38-43 : on met dans la configuration le chemin du fichier **[admindata.json]** ;
- lignes 45-48 : configuration **[sqlalchemy]** ;
- lignes 50-53 : instanciation des couches de l'application ;
- ligne 56 : on rend la configuration générale ;

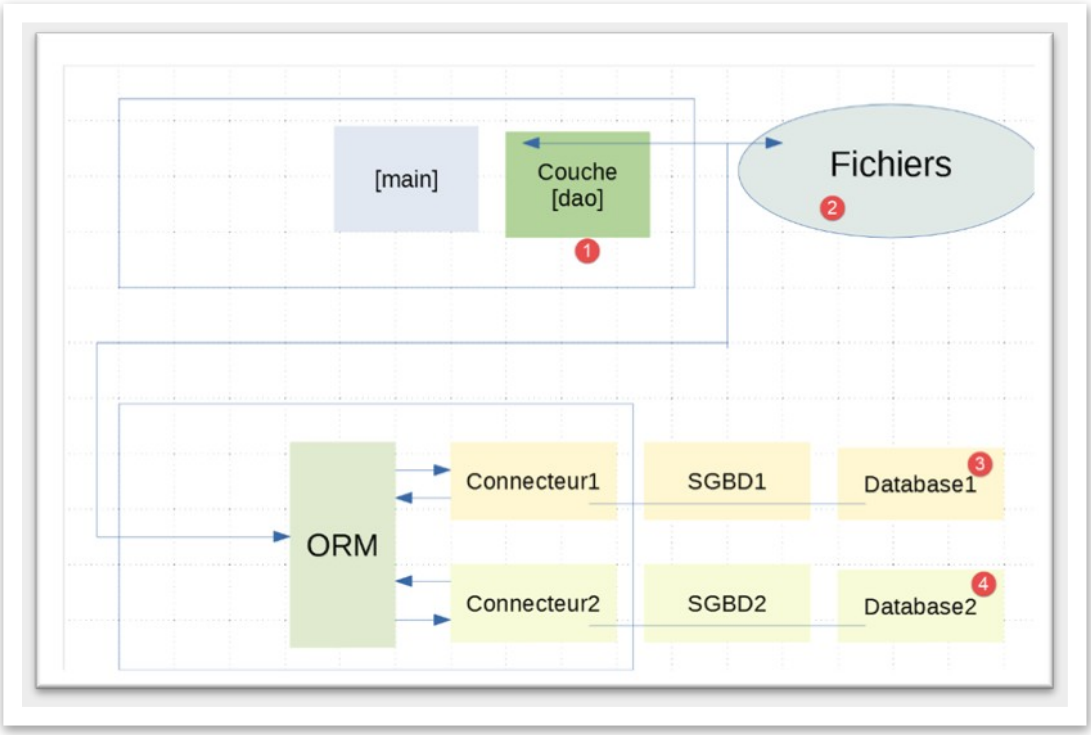
Le fichier **[config_layers]** est le suivant :

```
1.  def configure(config: dict) -> dict:
2.      # instanciation couche [dao]
3.      from DaoTransferAdminDataFromJsonFile2Database import DaoTransferAdminDataFromJsonFile2Database
4.      config['dao'] = DaoTransferAdminDataFromJsonFile2Database(config)
5.
6.      # on rend la config
7.      return config
```

- lignes 3-4 : instanciation de la couche **[dao]**. On a vu que le constructeur de la classe **[DaoTransferAdminDataFromJsonFile2Database]** attendait en paramètre le dictionnaire de la configuration générale de l'application ;
- ligne 4 : la référence sur la couche **[dao]** est mise dans la configiration ;
- ligne 7 : on rend la configuration ;

20.1.7 Le script `[main]` de l'application





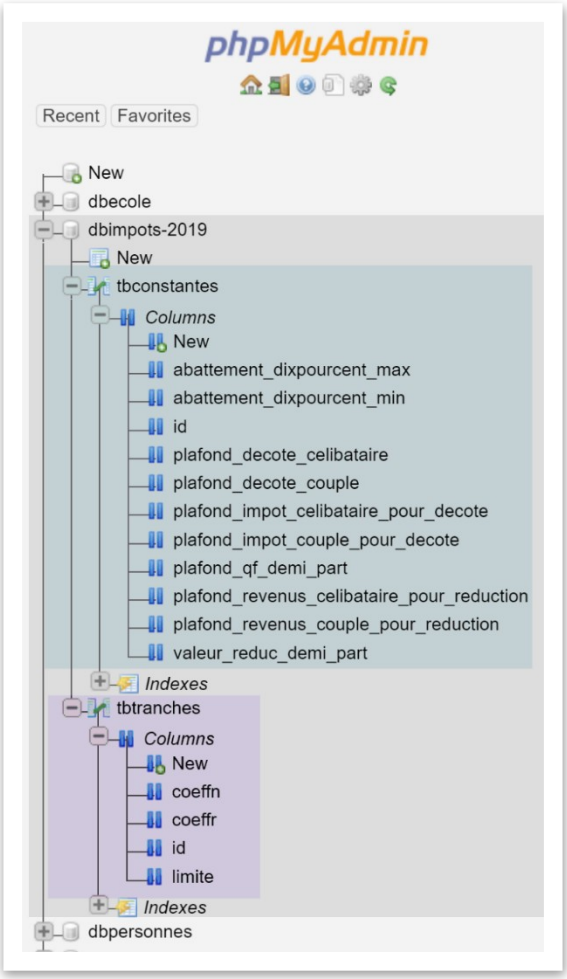
Le script principal **`[main]`** est le suivant :

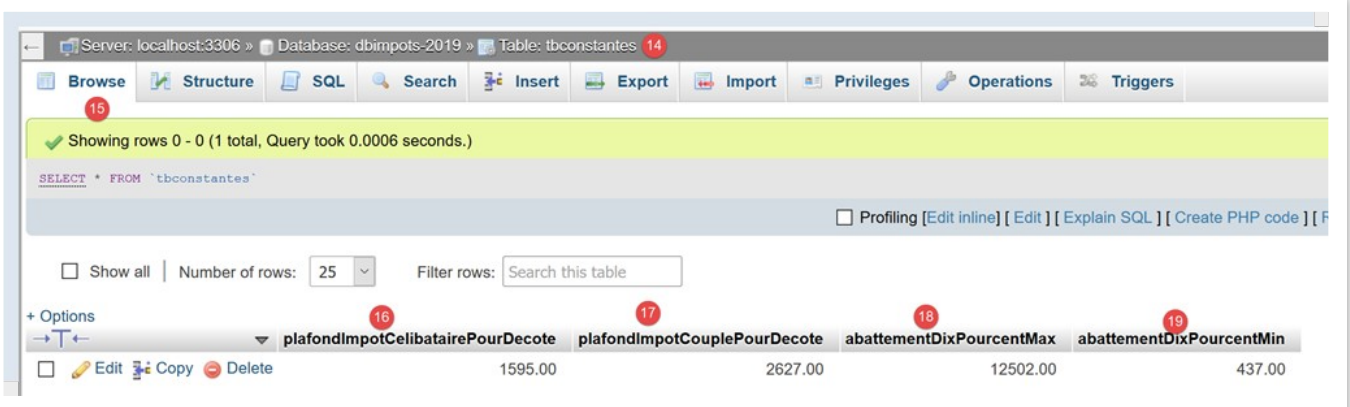
```
1.  # on attend un paramètre mysql ou pgres
2.  import sys
3.  syntaxe = f"{sys.argv[0]} mysql / pgres"
4.  erreur = len(sys.argv) != 2
5.  if not erreur:
6.      sgbd = sys.argv[1].lower()
7.      erreur = sgbd != "mysql" and sgbd != "pgres"
8.  if erreur:
9.      print(f"syntaxe : {syntaxe}")
10.     sys.exit()
11.
12. # on configure l'application
13. import config
14. config = config.configure({'sgbd': sgbd})
15.
16. # le syspath est établi - on peut faire les imports
17. from ImpôtsError import ImpôtsError
18.
```

```
19. # on récupère la couche [dao]
20. dao = config["dao"]
21.
22. # code
23. try:
24.     # transfert des données dans la base
25.     dao.transfer_admindata_in_database()
26. except ImpôtsError as ex1:
27.     # on affiche l'erreur
28.     print(f"L'erreur 1 suivante s'est produite : {ex1}")
29. except BaseException as ex2:
30.     # on affiche l'erreur
31.     print(f"L'erreur 2 suivante s'est produite : {ex2}")
32. finally:
33.     # fin
34.     print("Terminé...")
```

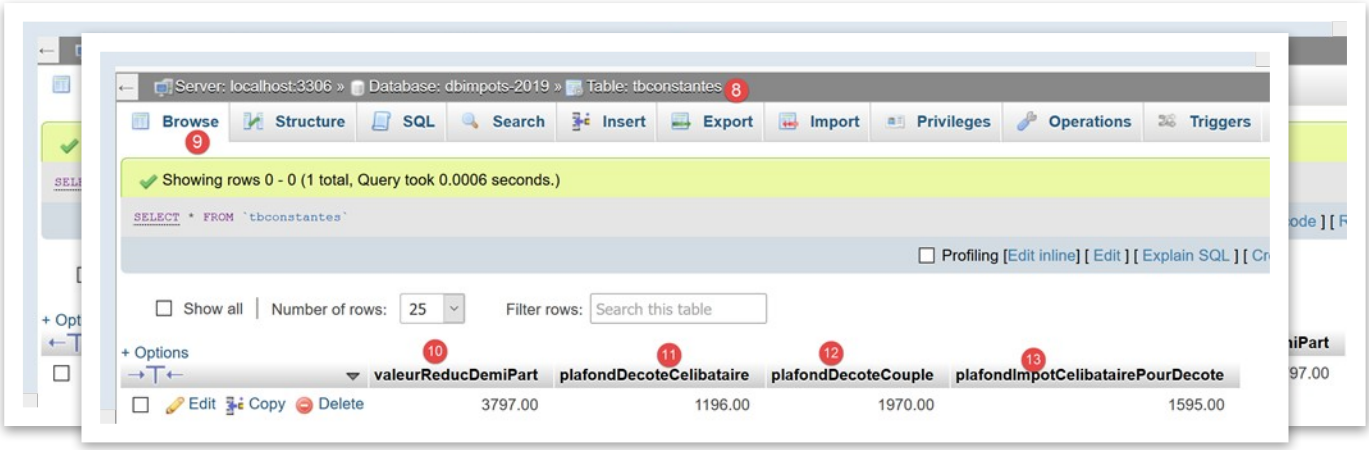
- lignes 1-10 : on attend un paramètre. On vérifie qu'il est là et correct ;
- lignes 12-14 : on configure l'application (générale, sqlalchemy, couches) en passant en paramètre le type de SGBD choisi ;
- lignes 19-20 : on va avoir besoin de la couche **[dao]**. On la récupère ;
- ligne 25 : on fait le transfert en base. Toutes les informations nécessaires à la méthode **[transfer_admindata_in_database]** sont disponibles dans les propriétés de la couche **[dao]** de la ligne 20. C'est là qu'elle ira les chercher ;

Après exécution avec la base MySQL, celle-ci contient les éléments suivants (phpMyAdmin) :

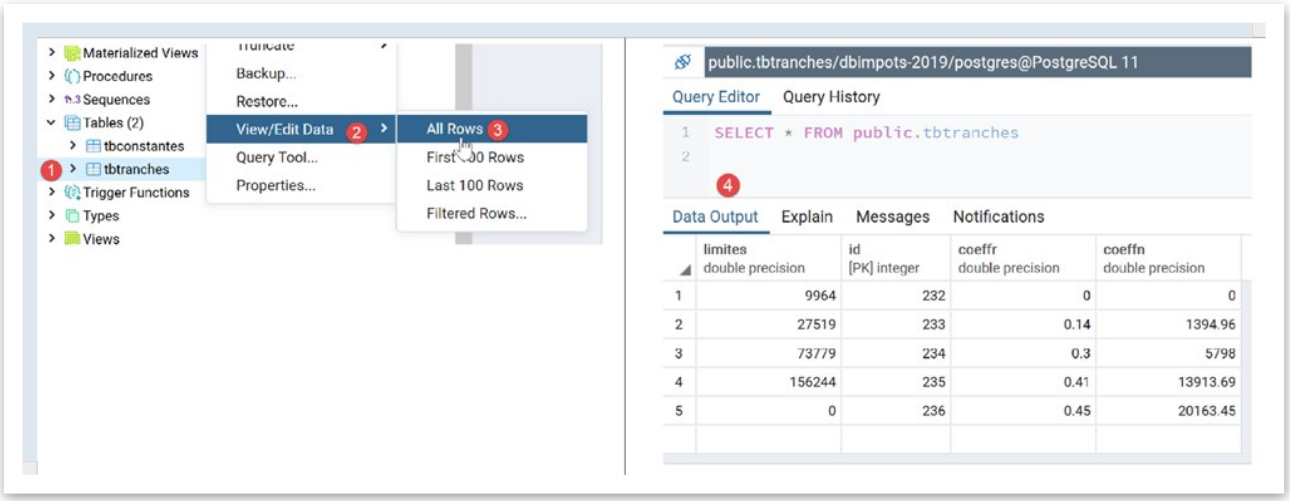




Colonne **[3]**, on voit les valeurs attribuées par MySQL à la clé primaire **[id]**. La numérotation démarre à 1. La copie d'écran ci-dessus a été obtenue après plusieurs exécutions du script.



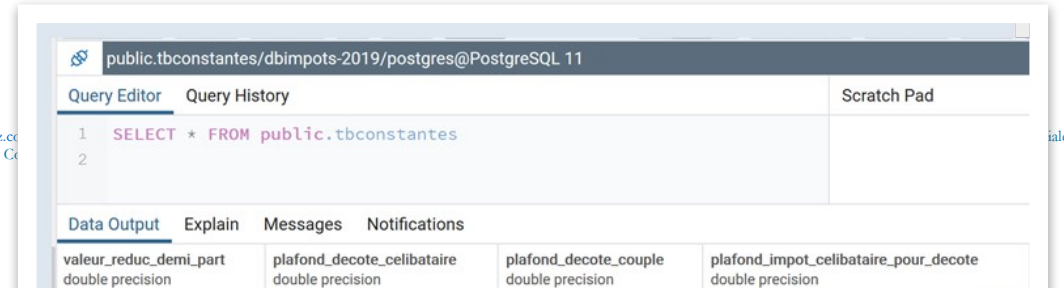
Avec la base PostgreSQL les résultats sont les suivants :

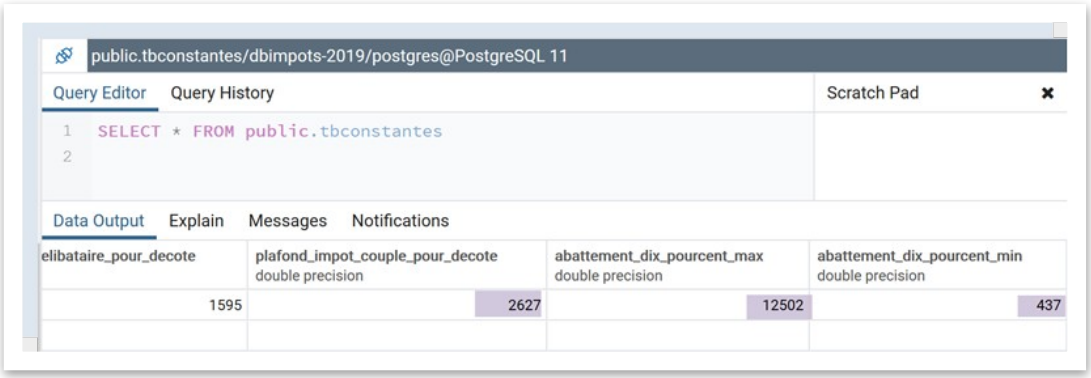


- on clique droit sur **[1]**, puis ensuite **[2-3]** ;
- en **[4]**, on a bien les données des tranches d'impôts ;

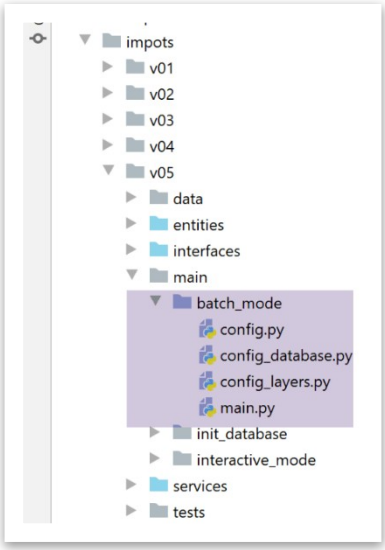
On refait la même chose pour la table des constantes **[tbconstantes]** :





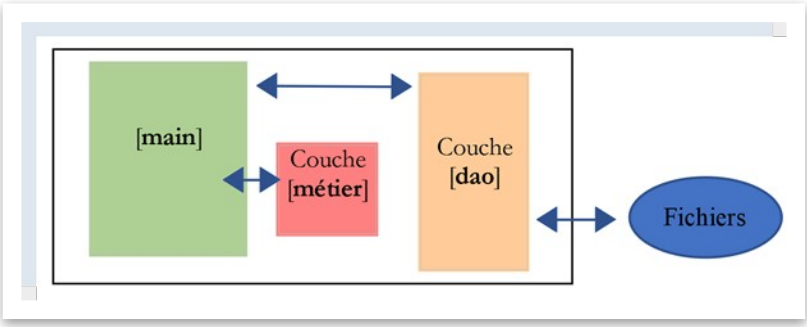


20.2 Application 2 : calcul de l'impôt en mode batch



20.2.1 Architecture

L'application de calcul de l'impôt de la version 4 utilisait l'architecture suivante :

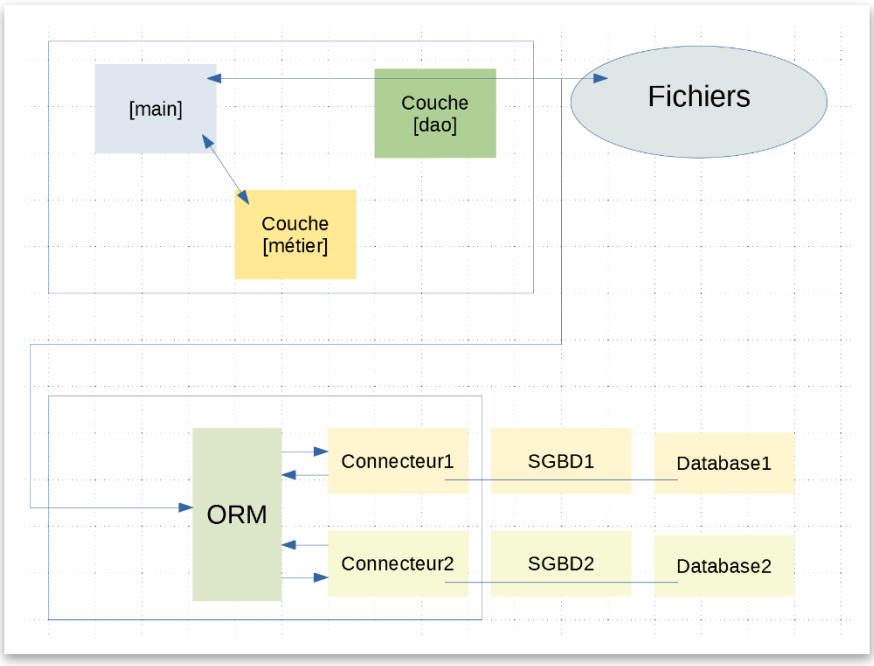


La couche **[dao]** implémente une interface **[InterfaceImpôtsDao]**. Nous avons construit une classe implémentant cette interface :

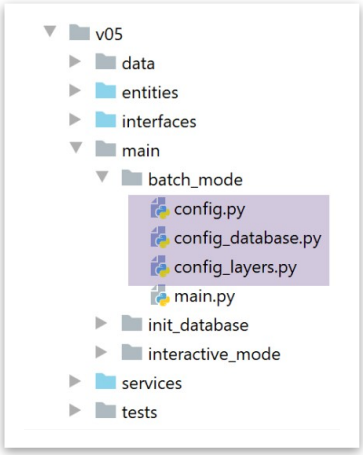
- **[ImpôtsDaoWithAdminDataInJsonFile]** qui allait chercher les données fiscales dans un fichier jSON. C'était la version 3 ;

Nous allons implémenter l'interface **[InterfaceImpôtsDao]** par une nouvelle classe **[ImpotsDaoWithTaxAdminDataInDatabase]** qui ira chercher les données de l'administration fiscale dans une base de données. La couche **[dao]**, comme précédemment, écrira les résultats dans un fichier jSON et trouvera les données des contribuables dans un fichier texte. Nous savons que si nous continuons à respecter l'interface **[InterfaceImpôtsDao]**, la couche **[métier]** n'aura pas à être modifiée.

La nouvelle architecture sera la suivante :



20.2.2 Configuration de l'application



Le fichier de configuration **[config_database]** reste ce qu'il était dans l'application 1. La configuration **[config]** intègre des éléments nouveaux :

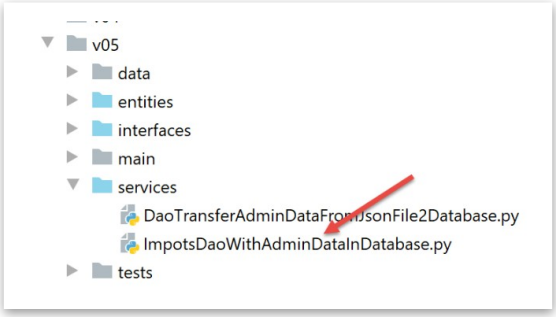
```
1.      # étape 2 ------
2.      # on complète la configuration de l'application
3.      config.update({
4.          # chemins absolus des fichiers de données
5.          "admindataFilename": f"{script_dir}/../../data/input/admindata.json",
6.          "taxpayersFilename": f"{script_dir}/../../data/input/taxpayersdata.txt",
7.          "errorsFilename": f"{script_dir}/../../data/output/errors.txt",
8.          "resultsFilename": f"{script_dir}/../../data/output/résultats.json"
9.      })
```

- lignes 6-8 : les chemins absolus des fichiers texte utilisés par l'application 2 ;

La configuration des couches **[config_layers]** évolue de la façon suivante :

```
1.  def configure(config: dict) -> dict:
2.      # instanciation couche dao
3.      from ImpotsDaoWithAdminDataInDatabase import ImpotsDaoWithAdminDataInDatabase
4.      config["dao"] = ImpotsDaoWithAdminDataInDatabase(config)
5.
6.      # instanciation couche [métier]
7.      from ImpôtsMétier import ImpôtsMétier
8.      config['métier'] = ImpôtsMétier()
9.
10.     # on rend la config
11.     return config
```

- lignes 3-4 : la couche **[dao]** est désormais implémentée par la classe **[ImpotsDaoWithAdminDataInDatabase]**. Cette classe est nouvelle mais implémente la même interface **[InterfaceDao]** que la version 4 de l'exercice d'application ;
- lignes 7-8 : la couche **[métier]** est implémentée par la classe **[ImpôtsMétier]**. C'est la classe utilisée dans la version 4 de l'exercice d'application ;



20.2.3 La couche [dao]

La classe d'implémentation **[ImpotsDaoWithAdminDataInDatabase]** de l'interface **[InterfaceImpôtsDao]** sera la suivante :

```
1.  # imports
2.  from sqlalchemy.exc import DatabaseError, IntegrityError, InterfaceError
3.
4.  from AbstractImpôtsDao import AbstractImpôtsDao
5.  from AdminData import AdminData
6.  from Constantes import Constantes
7.  from ImpôtsError import ImpôtsError
8.  from Tranche import Tranche
9.
10.
11. class ImpotsDaoWithAdminDataInDatabase(AbstractImpôtsDao):
12.     # constructeur
13.     def __init__(self, config: dict):
14.         # config["taxPayersFilename"] : le nom du fichier texte des contribuables
15.         # config["taxPayersResultsFilename"] : le nom du fichier jSON des résultats
16.         # config["errorsFilename"] : enregistre les erreurs trouvées dans taxPayersFilename
17.         # config["database"] : configuration de la base de données
18.
19.         # initialisation de la classe Parent
20.         AbstractImpôtsDao.__init__(self, config)
21.         # mémorisation paramètre
22.         self.__config = config
23.         # admindata
24.         self.__admindata = None
25.
26.     # implémentation de l'interface
27.     def get_admindata(self):
28.         # admindata a-t-il été mémorisé ?
29.         if self.__admindata:
30.             return self.__admindata
31.         # on fait une requête en BD
32.         session = None
33.         config = self.__config
34.         try:
35.             # une session
36.             database_config = config["database"]
```

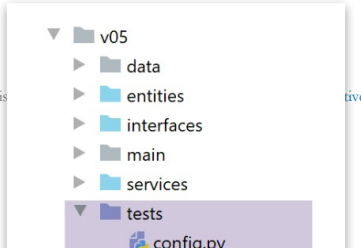
```
37.             session = database_config["session"]
38.
39.             # on lit la table des tranches de l'impôt
40.             tranches = session.query(Tranche).all()
41.
42.             # on lit la table des constantes (1 seule ligne)
43.             constantes = session.query(Constantes).first()
44.
45.             # on crée l'instance admindata
46.             admindata = AdminData()
47.             # on y crée les tableaux limtes, coeffR, coeffN
48.             limites = admindata.limites = []
49.             coeffr = admindata.coeffr = []
50.             coeffn = admindata.coeffn = []
51.             for tranche in tranches:
52.                 limites.append(float(tranche.limite))
53.                 coeffr.append(float(tranche.coeffr))
54.                 coeffn.append(float(tranche.coeffn))
55.             # on y rajoute les constantes
56.             admindata.fromdict(constantes.asdict())
57.             # on mémorise admindata
58.             self.__admindata = admindata
59.             # on rend la valeur
60.             return self.__admindata
61.         except (IntegrityError, DatabaseError, InterfaceError) as erreur:
62.             # on relance l'exception sous une autre forme
63.             raise ImpôtsError(27, f"{erreur}")
64.         finally:
65.             # on ferme la session
66.             if session:
67.                 session.close()
```

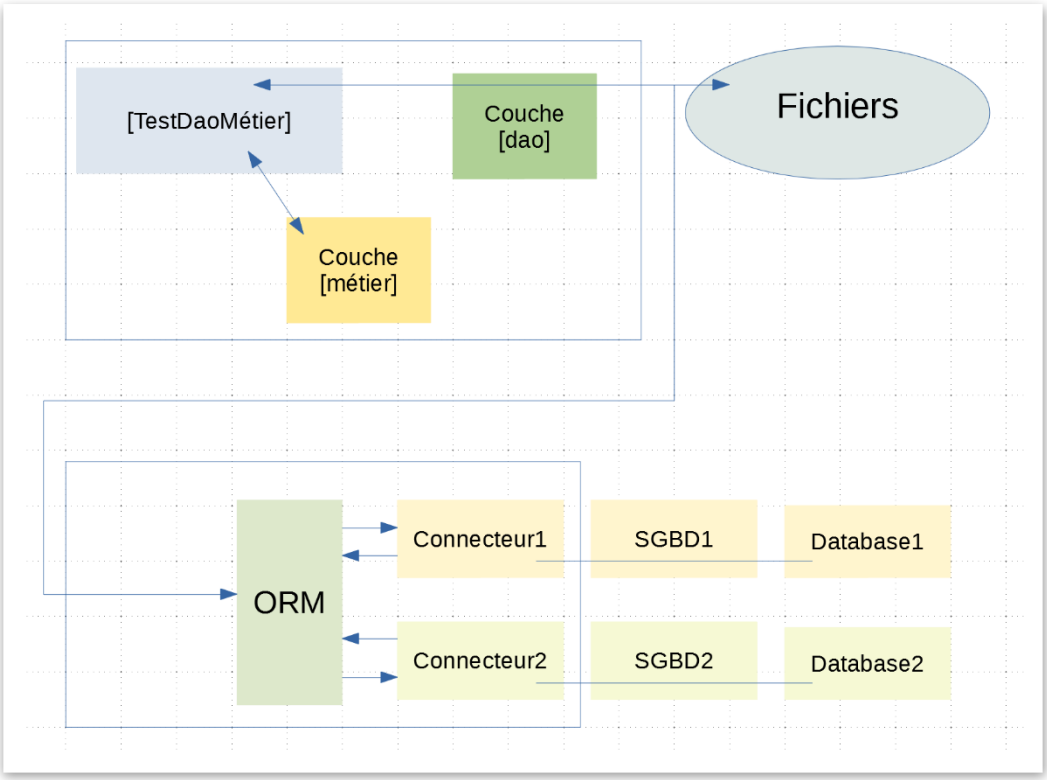
Notes

- ligne 11 : la classe **[ImpotsDaoWithAdminDataInDatabase]** hérite de la classe [AbstractImpôtsDao] présentée dans la version 4. On sait que cette dernière implémente l'interface [InterfaceDao] présentée dans cette même version. C'est le respect de cette interface qui nous permet de ne pas changer la couche **[métier]** ;
- ligne 13 : le constructeur de la classe reçoit en paramètre le dictionnaire de la configuration de l'application ;
- ligne 20 : la classe parent **[]** est initialisée. Elle implémente partiellement l'interface **[InterfaceDao]** :
  - **[get_taxpayers_data]** lit le fichier **[taxpayersdata.txt]** qui contient les données des contribuables ;
  - **[write_taxpayers_results]** écrit les résultats dans le fichier jSON **[résultats.json]** ;
  - **[get_admindata]** n'est pas implémentée ;
- ligne 22 : on mémorise la configuration passée en paramètres ;
- ligne 27 : implémentation de la méthode **[get_admindata]** de l'interface **[InterfaceDao]** :
- lignes 28-30 : la méthode **[get_admindata]** récupère les données de l'administration fiscale dans un objet de type **[AdminData]** et mémorise cet objet dans **[self.__admindata]**. Si la méthode **[get_admindata]** est appelée plusieurs fois, on n'interroge pas la base de données plusieurs fois. On l'interroge seulement la première fois. Les fois suivantes, on rend l'objet **[self.__admindata]** ;
- lignes 36-37 : on récupère la session **[sqlalchemy]** qui a été créée lors de la configuration de l'application par **[config_database]** ;
- lignes 40 : on récupère les tranches de l'impôt dans une liste ;
- lignes 43 : on récupère les constantes du calcul de l'impôt ;
- ligne 46 : on crée une instance de la classe [AdminData]. On rappelle qu'elle dérive de **[BaseEntity]** ;
- lignes 48-54 : on initialise les tableaux **[limites, coeffr, coeffn]** de l'instance **[AdminData]** ;
- lignes 55-56 : on initialise les autres propriétés de **[AdminData]** avec les constantes du calcul de l'impôt. On avait pris soin de donner les mêmes noms aux propriétés des classes **[AdminData]** et **[Constantes]**, ce qui simplifie le code ;
- lignes 57-58 : l'instance **[AdminData]** est mémorisée dans la couche **[dao]** pour la rendre lors des prochains appels à la méthode **[get_admindata]** ;
- ligne 60 : on rend la valeur demandée par le code appelant ;
- lignes 61-63 : gestion d'une éventuelle erreur ;
- lignes 64-67 : la base de données ne fait l'objet que d'une unique requête. On peut donc fermer la session **[sqlalchemy]** ;

20.2.4 Test de la couche [dao]

Dans la version 4 de cette application, nous avions construit une classe de test de la couche **[métier]**. Plus exactement, elle testait à la fois les couches **[métier]** et **[dao]**. Nous reprenons ce test pour vérifier que la couche **[dao]** fonctionne comme attendu. En effet, la couche **[métier]** elle ne change pas.





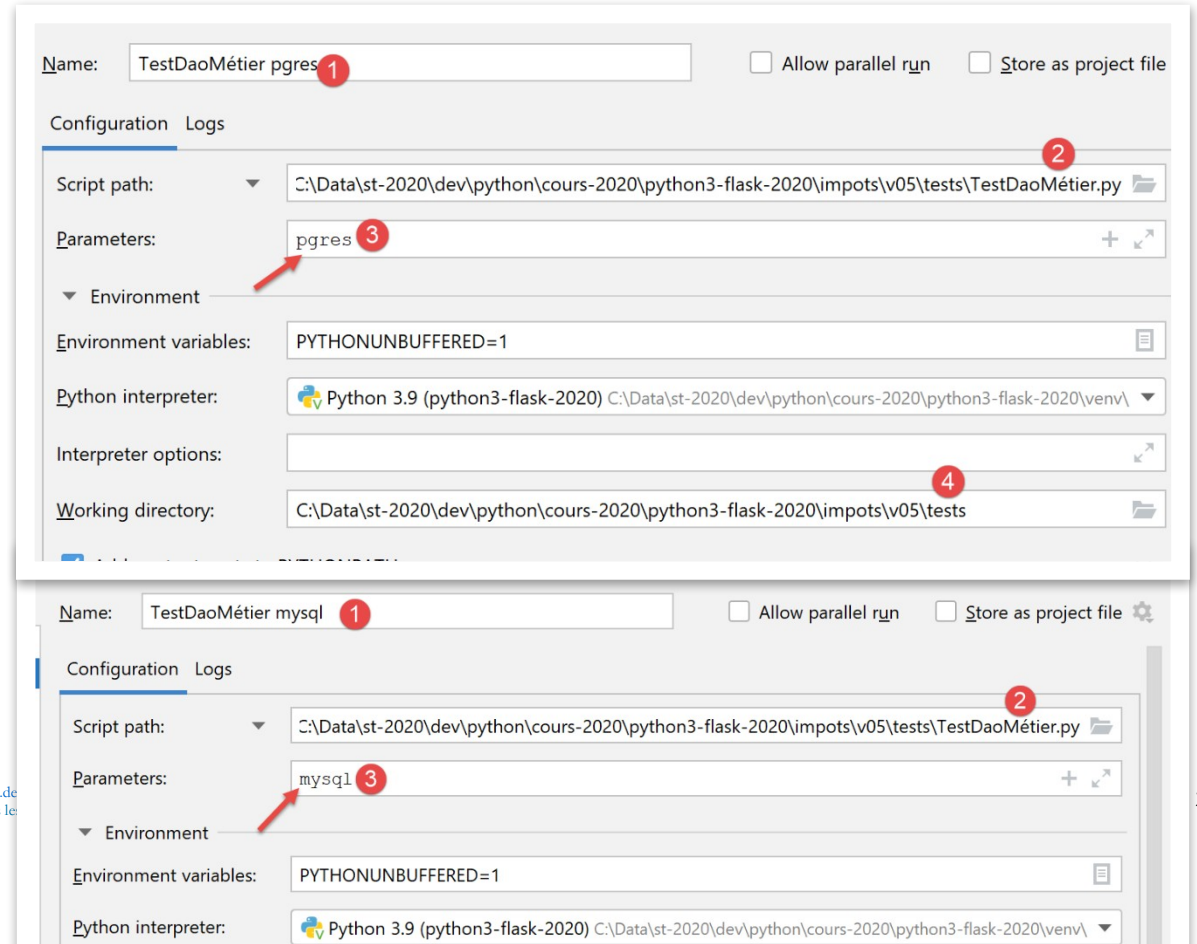
Le test **[TestDaoMétier]** est le suivant :

```
1.  import unittest
2.
3.
4.  class TestDaoMétier(unittest.TestCase):
5.
6.      def test_1(self) -> None:
7.          from TaxPayer import TaxPayer
8.
9.          # {'marié': 'oui', 'enfants': 2, 'salaire': 55555,
10.         # 'impôt': 2814, 'surcôte': 0, 'décôte': 0, 'réduction': 0, 'taux': 0.14}
11.         taxpayer = TaxPayer().fromdict({"marié": "oui", "enfants": 2, "salaire": 55555})
12.         métier.calculate_tax(taxpayer, admindata)
13.         # vérification
14.         self.assertAlmostEqual(taxpayer.impôt, 2815, delta=1)
15.         self.assertEqual(taxpayer.décôte, 0)
16.         self.assertEqual(taxpayer.réduction, 0)
17.         self.assertAlmostEqual(taxpayer.taux, 0.14, delta=0.01)
18.         self.assertEqual(taxpayer.surcôte, 0)
19.
20.     …
21.
22.     def test_11(self) -> None:
23.         from TaxPayer import TaxPayer
24.
25.         # {'marié': 'oui', 'enfants': 3, 'salaire': 200000,
26.         # 'impôt': 42842, 'surcôte': 17283, 'décôte': 0, 'réduction': 0, 'taux': 0.41}
27.         taxpayer = TaxPayer().fromdict({'marié': 'oui', 'enfants': 3, 'salaire': 200000})
28.         métier.calculate_tax(taxpayer, admindata)
29.         # vérifications
30.         self.assertAlmostEqual(taxpayer.impôt, 42842, 1)
31.         self.assertEqual(taxpayer.décôte, 0)
32.         self.assertEqual(taxpayer.réduction, 0)
33.         self.assertAlmostEqual(taxpayer.taux, 0.41, delta=0.01)
34.         self.assertAlmostEqual(taxpayer.surcôte, 17283, delta=1)
35.
36.
37. if __name__ == '__main__':
38.     # on attend un paramètre mysql ou pgres
39.     import sys
```

```
40.     syntaxe = f"{sys.argv[0]} mysql / pgres"
41.     erreur = len(sys.argv) != 2
42.     if not erreur:
43.         sgbd = sys.argv[1].lower()
44.         erreur = sgbd != "mysql" and sgbd != "pgres"
45.     if erreur:
46.         print(f"syntaxe : {syntaxe}")
47.         sys.exit()
48.
49.     # on configure l'application
50.     import config
51.     config = config.configure({'sgbd': sgbd})
52.     # couche métier
53.     métier = config['métier']
54.     try:
55.         # admindata
56.         admindata = config['dao'].get_admindata()
57.     except BaseException as ex:
58.         # affichage
59.         print((f"L'erreur suivante s'est produite : {ex}"))
60.         # fin
61.         sys.exit()
62.     # on enève le paramètre reçu par le script
63.     sys.argv.pop()
64.     # on exécute les méthodes de test
65.     print("tests en cours...")
66.     unittest.main()
```

- nous ne revenons pas sur les 11 tests décrits au paragraphe |test couche **[métier]** version 4| ;
- lignes 37-66 : nous allons exécuter le script des tests comme une application normale et non pas comme un test UnitTest. C'est la ligne66qui fera intervenir le framework UnitTest. Dans les tests précédents, nous utilisions la méthode **[setUp]** pour configurer l'exécution de chaque test. On refaisait 11 fois la même configuration puisque la fonction **[setUp]** est exécutée avant chaque test. Ici, nous faisons la configuration 1 fois. Elle consiste à définir des variables globales **[métier]** ligne 53, **[admindata]** ligne 56 qui seront ensuite utilisées par les méthodes de **[TestDaoMétier]**, ligne 12 par exemple ;
- lignes 39-47 : le script de test attend un paramètre **[mysql / pgres]** qui indique si on utilise une base MySQL ou PostgreSQL ;
- lignes 50-51 : le test est configuré ;
- ligne 53 : on récupère la couche **[métier]** dans la configuration ;
- ligne 56 : on fait de même avec la couche **[dao]**. On récupère alors l'instance **[admindata]** qui encapsule les données nécessaires au calcul de l'impôt ;
- les tests ont montré que la méthode **[unittest.main()]** de la ligne 66 n'ignorait pas le paramètre **[mysql / pgres]** reçu par le script mais lui donnait une signification autre. La ligne 63 fait en sorte que cette méthode n'ait plus aucun paramètre ;

Nous créons deux configurations d'exécution :



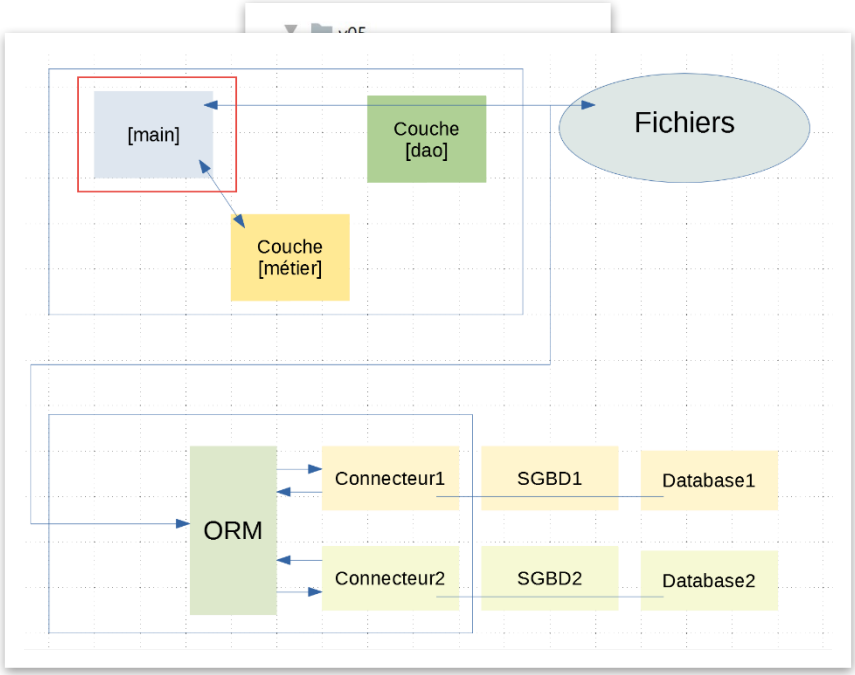
Si nous exécutons l'une de ces deux configurations, nous obtenons les résultats suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/v05/tests/TestDaoMétier.py mysql
2.  tests en cours...
3.  ..........
4.  ----------------------------------------------------------------------
5.  Ran 11 tests in 0.001s
6.
7.  OK
8.
9.  Process finished with exit code 0
```

- lignes 5 et 7 : les 11 tests ont été réussis ;

Rappelons que ces tests ne vérifient que 11 cas du calcul de l'impôt. Leur réussite peut néanmoins suffire pour nous donner confiance dans la couche **[dao]**.

20.2.5 Le script principal



Le script principal **[main]** est le même que dans la version 4 :

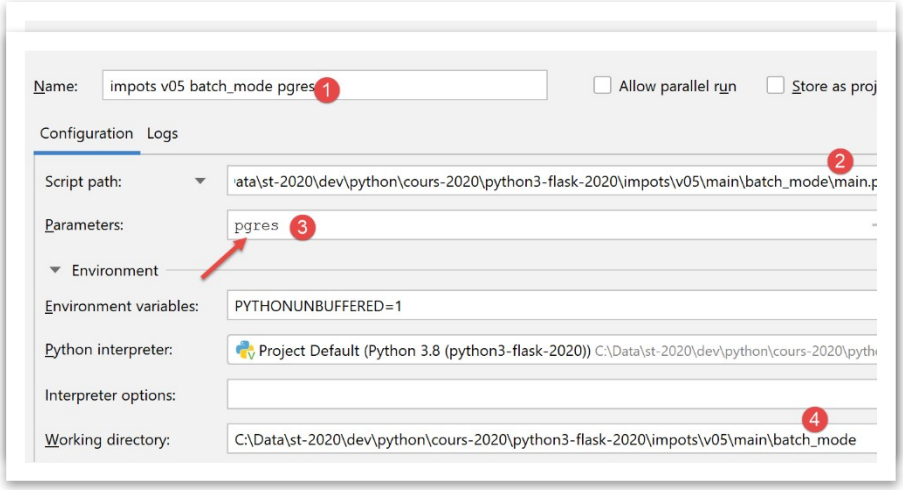
```
1.  # on attend un paramètre mysql ou pgres
2.  import sys
3.  syntaxe = f"{sys.argv[0]} mysql / pgres"
4.  erreur = len(sys.argv) != 2
5.  if not erreur:
6.      sgbd = sys.argv[1].lower()
7.      erreur = sgbd != "mysql" and sgbd != "pgres"
8.  if erreur:
9.      print(f"syntaxe : {syntaxe}")
10.     sys.exit()
11.
12. # on configure l'application
13. import config
14. config = config.configure({'sgbd': sgbd})
15.
16. # le syspath est établi - on peut faire les imports
17. from ImpôtsError import ImpôtsError
18.
19. # on récupère les couches de l'application (elles sont déjà instanciées)
20. dao = config["dao"]
21. métier = config["métier"]
22.
23. try:
24.     # récupération des tranches de l'impôt
```

```
25.      admindata = dao.get_admindata()
26.      # lecture des données des contribuables
27.      taxpayers = dao.get_taxpayers_data()["taxpayers"]
28.      # des contribuables ?
29.      if not taxpayers:
30.          raise ImpôtsError(57, f"Pas de contribuables valides dans le fichier {config['taxpayersFilename']}")
31.      # calcul de l'impôt des contribuables
32.      for taxPayer in taxpayers:
33.          # taxPayer est à la fois un paramètre d'entrée et de sortei
34.          # taxPayer va être modifié
35.          métier.calculate_tax(taxPayer, admindata)
36.      # écriture des résultats dans un fichier texte
37.      dao.write_taxpayers_results(taxpayers)
38.  except ImpôtsError as erreur:
39.      # affichage de l'erreur
40.      print(f"L'erreur suivante s'est produite : {erreur}")
41.  finally:
42.      # terminé
43.      print("Travail terminé...")
```

Notes

- lignes 1-10 : on récupère le paramètre **[mysql / pgres]** qui indique le SGBD à utiliser ;
- lignes 12-14 : l'application est configurée ;
- lignes 16-17 : la classe **[ImpôtsError]** est importée. On en a besoin ligne 38 ;
- lignes 19-21 : on récupère des références sur les couches de l'application ;
- ligne 25 : on demande à la couche **[dao]** les données de l'administration fiscale. La couche **[métier]** en a besoin pour le calcul de l'impôt ;
- ligne 27 : on récupère dans une liste, les données (id, marié, enfants, salaire) des contribuables ;
- lignes 29-30 : si cette liste est vide, on lance une exception ;
- lignes 32-35 : calcul de l'impôt des éléments de la liste **[taxpayers]** ;
- ligne 37 : écriture des résultats dans le fichier jSON**[résultats.json]** ;
- lignes 38-40 : gestion de l'éventuelle erreur ;

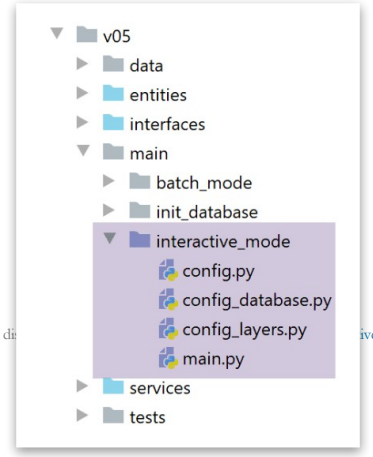
Pour l'exécution du script, on crée deux |configurations d'exécution| :

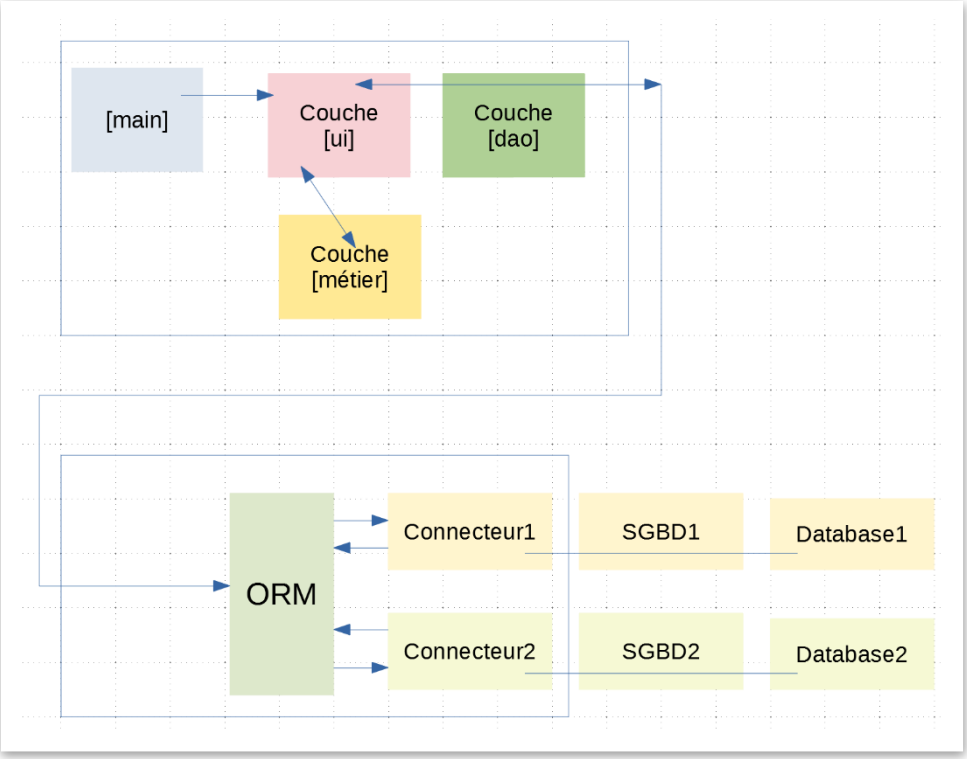


Les résultats obtenus dans le fichier **[résultats.json]** sont ceux de la version 4.

20.3 Application 3 : calcul de l'impôt en mode interactif

Nous introduisons maintenant l'application permettant de calculer l'impôt de façon interactive. C'est un portage de l'application 2 de la version 4.





- le script **[main]** lance le dialogue avec l'utilisateur avec la méthode **[ui.run]** de la couche **[ui]** ;
- la couche **[ui]** :
  - utilise la couche **[dao]** pour obtenir les données permettant de faire le calcul de l'impôt ;
  - demande à l'utilisateur les renseignements concernant le contribuable dont on veut calculer l'impôt ;
  - utilise la couche **[métier]** pour faire ce calcul ;

Le fichier **[config_layers]** instancie une couche supplémentaire :

```
1.  def configure(config: dict) -> dict:
2.      # instanciation couche dao
3.      from ImpotsDaoWithAdminDataInDatabase import ImpotsDaoWithAdminDataInDatabase
4.      config["dao"] = ImpotsDaoWithAdminDataInDatabase(config)
5.
6.      # instanciation couche [métier]
7.      from ImpôtsMétier import ImpôtsMétier
8.      config['métier'] = ImpôtsMétier()
9.
10.     # ui
11.     from ImpôtsConsole import ImpôtsConsole
12.     config['ui'] = ImpôtsConsole(config)
13.
14.     # on rend la config
15.     return config
```

La classe **[ImpôtsConsole]**, lignes 11-12, est la même que dans la |version 4|.

Le script principal **[main]** est le suivant :

```
1.  # on attend un paramètre mysql ou pgres
2.  import sys
3.  syntaxe = f"{sys.argv[0]} mysql / pgres"
4.  erreur = len(sys.argv) != 2
5.  if not erreur:
6.      sgbd = sys.argv[1].lower()
7.      erreur = sgbd != "mysql" and sgbd != "pgres"
8.  if erreur:
9.      print(f"syntaxe : {syntaxe}")
10.     sys.exit()
11.
12. # on configure l'application
13. import config
14. config = config.configure({'sgbd': sgbd})
```

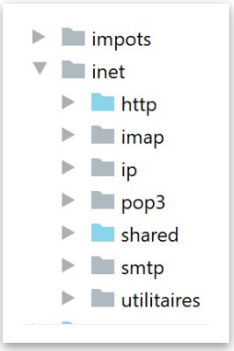
```
15.
16. # le syspath est configuré - on peut faire les imports
17. from ImpôtsError import ImpôtsError
18.
19. # on récupère la couche [ui]
20. ui = config["ui"]
21.
22. # code
23. try:
24.     # exécution de la couche [ui]
25.     ui.run()
26. except ImpôtsError as ex1:
27.     # on affiche le message d'erreur
28.     print(f"L'erreur 1 suivante s'est produite : {ex1}")
29. except BaseException as ex2:
30.     # on affiche le message d'erreur
31.     print(f"L'erreur 2 suivante s'est produite : {ex2}")
32. finally:
33.     # exécuté dans tous les cas
34.     print("Travail terminé...")
```

- lignes 1-10, le script attend un paramètre **[mysql / pgres]** qui indique le SGBD à utiliser ;
- lignes 12-14 : l'application est configurée ;
- lignes 19-20 : on récupère la couche **[ui]** dans la configuration ;
- ligne 25 : on l'exécute ;

Les résultats sont identiques à ceux de la |version 4|. Il ne pouvait en être autrement puisque toutes les interfaces de la version 4 ont été respectées dans la version 5.

21 Fonctions internet

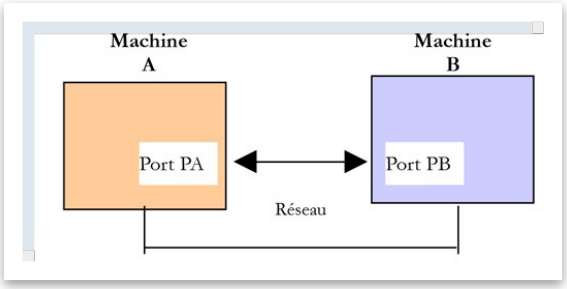
Nous abordons maintenant les fonctions internet de Python qui nous permettent de faire de la programmation TCP / IP (**T**ransfer **C**ontrol **P**rotocol / **I**nternet **P**rotocol).



21.1 Les bases de la programmation internet

21.1.1 Généralités

Considérons la communication entre deux machines distantes A et B :



Lorsque une application *AppA* d'une machine A veut communiquer avec une application *AppB* d'une machine B de l'Internet, elle doit connaître plusieurs choses :

- l'adresse IP (**I**nternet **P**rotocol) ou le nom de la machine B ;
- le numéro du port avec lequel travaille l'application *AppB*. En effet la machine B peut supporter de nombreuses applications qui travaillent sur l'Internet. Lorsqu'elle reçoit des informations provenant du réseau, elle doit savoir à quelle application sont destinées ces informations. Les applications de la machine B ont accès au réseau via des guichets appelés également des **ports de communication**. Cette information est contenue dans le paquet reçu par la machine B afin qu'il soit délivré à la bonne application ;
- les protocoles de communication compris par la machine B. Dans notre étude, nous utiliserons uniquement les protocoles TCP-IP ;
- le protocole de dialogue accepté par l'application *AppB*. En effet, les machines A et B vont se "parler". Ce qu'elles vont dire va être encapsulé dans les protocoles TCP-IP. Néanmoins, lorsqu'au bout de la chaîne, l'application *AppB* va recevoir l'information envoyée par l'applicaton *AppA*, il faut qu'elle soit capable de l'interpréter. Ceci est analogue à la situation où deux personnes A et B communiquent par téléphone : leur dialogue est transporté par le téléphone. La parole va être codée sous forme de signaux par le téléphone A, transportée par des lignes téléphoniques, arriver au téléphone B pour y être décodée. La personne B entend alors des paroles. C'est là qu'intervient la notion de protocole de dialogue : si A parle français et que B ne comprend pas cette langue, A et B ne pourront dialoguer utilement ;

  Aussi les deux applications communicantes doivent-elles être d'accord sur le type de dialogue qu'elles vont adopter. Par exemple, le dialogue avec un service *ftp* n'est pas le même qu'avec un service *pop* : ces deux services n'acceptent pas les mêmes commandes. Elles ont un protocole de dialogue différent ;

21.1.2 Les caractéristiques du protocole TCP

Nous n'étudierons ici que des communications réseau utilisant le protocole de transport TCP dont voici les principales caractéristiques :

- le processus qui souhaite émettre établit tout d'abord une **connexion** avec le processus destinataire des informations qu'il va émettre. Cette connexion se fait entre un port de la machine émettrice et un port de la machine réceptrice. Il y a entre les deux ports un chemin virtuel qui est ainsi créé et qui sera réservé aux deux seuls processus ayant réalisé la connexion ;
- tous les paquets émis par le processus source suivent ce chemin virtuel et arrivent dans l'ordre où ils ont été émis ;
- l'information émise a un aspect continu. Le processus émetteur envoie des informations à son rythme. Celles-ci ne sont pas nécessairement envoyées tout de suite : le protocole TCP attend d'en avoir assez pour les envoyer. Elles sont stockées dans une structure appelée *segment TCP*. Ce segment une fois rempli sera transmis à la couche IP où il sera encapsulé dans un paquet IP ;
- chaque segment envoyé par le protocole TCP est numéroté. Le protocole TCP destinataire vérifie qu'il reçoit bien les segments en séquence. Pour chaque segment correctement reçu, il envoie un accusé de réception à l'expéditeur ;
- lorsque ce dernier le reçoit, il l'indique au processus émetteur. Celui-ci peut donc savoir qu'un segment est arrivé à bon port ;
- si au bout d'un certain temps, le protocole TCP ayant émis un segment ne reçoit pas d'accusé de réception, il retransmet le segment en question, garantissant ainsi la qualité du service d'acheminement de l'information ;
- le circuit virtuel établi entre les deux processus qui communiquent est *full-duplex* : cela signifie que l'information peut transiter dans les deux sens. Ainsi le processus destination peut envoyer des accusés de réception alors même que le processus source continue d'envoyer des informations. Cela permet par exemple au protocole TCP source d'envoyer plusieurs segments sans attendre d'accusé de réception. S'il réalise au bout d'un certain temps qu'il n'a pas reçu l'accusé de réception d'un certain segment n° *n*, il reprendra l'émission des segments à ce point ;

21.1.3 La relation client-serveur

Souvent, la communication sur Internet est dissymétrique : la machine A initie une connexion pour demander un service à la machine B : il précise qu'il veut ouvrir une connexion avec le service SB1 de la machine B. Celle-ci accepte ou refuse. Si elle accepte, la machine A peut envoyer ses demandes au service SB1. Celles-ci doivent se conformer au protocole de dialogue compris par le service SB1. Un dialogue demande-réponse s'instaure ainsi entre la machine A qu'on appelle machine **cliente** et la machine B qu'on appelle machine **serveur**. L'un des deux partenaires fermera la connexion.

21.1.4 Architecture d'un client

L'architecture d'un programme réseau demandant les services d'une application serveur sera la suivante :

```
1.  ouvrir la connexion avec le service SB1 de la machine B
2.  si réussite alors
3.          tant que ce n'est pas fini
4.                  préparer une demande
5.                  l'émettre vers la machine B
6.                  attendre et récupérer la réponse
7.                  la traiter
8.          fin tant que
9.  finsi
10. fermer la connexion
```

21.1.5 Architecture d'un serveur

L'architecture d'un programme offrant des services sera la suivante :

```
1.  ouvrir le service sur la machine locale
2.  tant que le service est ouvert
3.          se mettre à l'écoute des demandes de connexion sur un port dit port d'écoute
4.          lorsqu'il y a une demande, la faire traiter par une autre tâche sur un autre port dit port de service
5.  fin tant que
```

Le programme serveur traite différemment la demande de connexion initiale d'un client de ses demandes ultérieures visant à obtenir un service. Le programme n'assure pas le service lui-même. S'il le faisait, pendant la durée du service il ne serait plus à l'écoute des demandes de connexion et des clients ne seraient alors pas servis. Il procède autrement : dès qu'une demande de connexion est reçue sur le port d'écoute puis acceptée, le serveur crée une tâche chargée de rendre le service demandé par le client. Ce service est rendu sur un autre port de la machine serveur appelé **port de service**. On peut ainsi servir plusieurs clients en même temps.

Une tâche de service aura la structure suivante :

```
1.  tant que le service n'a pas été rendu totalement
2.          attendre une demande sur le port de service
3.          lorsqu'il y en a une, élaborer la réponse
4.          transmettre la réponse via le port de service
5.  fin tant que
6.  libérer le port de service
```

21.2 Découvrir les protocoles de communication de l'internet

21.2.1 Introduction

Lorsqu'un client s'est connecté à un serveur, s'établit ensuite un dialogue entre-eux. La nature de celui-ci forme ce qu'on appelle le protocole de communication du serveur. Parmi les protocoles les plus courants de l'internet on trouve les suivants :

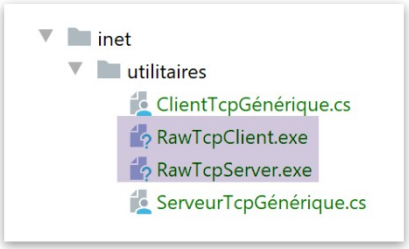
- HTTP : **H**yper**T**ext **T**ransfer **P**rotocol - le protocole de dialogue avec un serveur web (serveur HTTP) ;
- SMTP : **S**imple **M**ail **T**ransfer **P**rotocol - le protocole de dialogue avec un serveur d'envoi de courriers électroniques (serveur SMTP) ;
- POP : **P**ost **O**ffice **P**rotocol - le protocole de dialogue avec un serveur de stockage du courrier électronique (serveur POP). Il s'agit là de récupérer les courriers électroniques reçus et non d'en envoyer ;
- IMAP : **I**nternet **M**essage **A**ccess **P**rotocol - le protocole de dialogue avec un serveur de stockage du courrier électronique (serveur IMAP). Ce protocole a remplacé progressivement le protocole POP plus ancien ;
- FTP : **F**ile **T**ransfer **P**rotocol - le protocole de dialogue avec un serveur de stockage de fichiers (serveur FTP) ;

Tous ces protocoles ont la particularité d'être des protocoles à lignes de texte : le client et le serveur s'échangent des lignes de texte. Si on a un client capable de :

- créer une connexion avec un serveur TCP ;
- afficher à la console les lignes de texte que le serveur lui envoie ;
- envoyer au serveur les lignes de texte qu'un utilisateur saisirait au clavier ;

alors on est capable de dialoguer avec un serveur TCP ayant un protocole à lignes de texte pour peu qu'on connaisse les règles de ce protocole.

21.2.2 Utilitaires TCP



Dans les codes associés à ce document, on trouve deux utilitaires de communication TCP :

- **`[RawTcpClient]`** permet de se connecter sur le port P d'un serveur S ;
- **`[RawTcpServer]`** permet de créer un serveur qui attend des clients sur un port P ;

Ce sont deux programmes C# dont les codes sources vous sont donnés. Vous pouvez donc les modifier.

Le serveur TCP **`[RawTcpServer]`** s'appelle avec la syntaxe **`[RawTcpServeur port]`** pour créer un service TCP sur le port **`[port]`** de la machine locale (l'ordinateur sur lequel vous travaillez) :

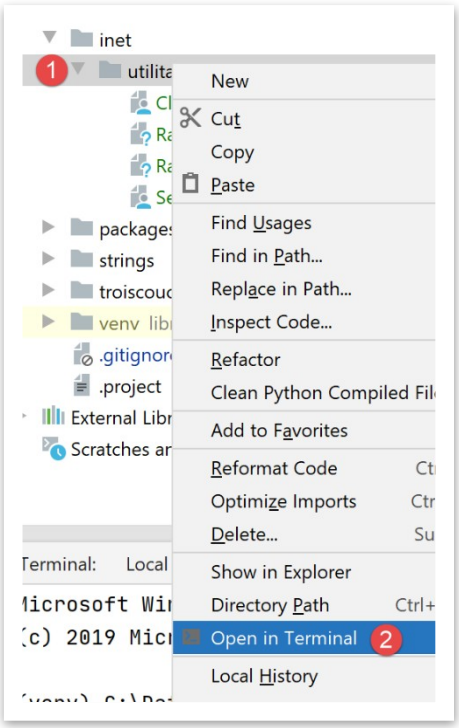
- le serveur peut servir plusieurs clients simultanément ;
- le serveur exécute les commandes tapées par l'utilisateur tapées au clavier. Celles-ci sont les suivantes :
  - **list** : liste les clients actuellement connectés au serveur. Ceux-ci sont affichés sous la forme **`[id=x-nom=y]`**. Le champ **`[id]`** sert à identifier les clients ;
  - **send x `[texte]`** : envoie **texte** au client n° **x** (id=x). Les crochets **`[]`** ne sont pas envoyés. Ils sont nécessaires dans la commande. Ils servent à délimiter visuellement le texte envoyé au client ;
  - **close x** : ferme la connexion avec le client n° **x** ;
  - **quit** : ferme toutes les connexions et arrête le service ;
- les lignes envoyées par le client au serveur sont affichées sur la console ;
- l'ensemble des échanges est logué dans un fichier texte portant le nom **`[machine-port.txt]`** où
  - **`[machine]`** est le nom de la machine sur laquelle s'exécute le code ;
  - **`[port]`** est le port de service qui répond aux demandes du client ;

Le client TCP **`[RawTcpClient]`** s'appelle avec la syntaxe **`[RawTcpClient serveur port]`** pour se connecter au port **`[port]`** du serveur **`[serveur]`** :

- les lignes tapées par l'utilisateur au clavier sont envoyées au serveur ;

- les lignes envoyées par le serveur sont affichées sur la console ;
- l'ensemble des échanges est logué dans un fichier texte portant le nom **[serveur-port.txt]** ;

Voyons un exemple. On ouvre deux fenêtres terminal PyCharm et on se positionne dans chacune d'elles sur le dossier des utilitaires :



Dans l'une des fenêtres on lance le serveur **[RawTcpServer]** sur le port 100 :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpServer.exe 100
2.  server : Serveur générique lancé sur le port 0.0.0.0:100
3.  server : Attente d'un client...
4.  server : Commandes disponibles : [list, send id [texte], close id, quit]
5.  user :
```

- ligne 1, nous sommes placés dans le dossier des utilitaires ;
- ligne 1, nous lançons le serveur TCP sur le port 100 ;
- lignes 2-4, le serveur se met en attente d'un client TCP et affiche une liste de commandes que l'utilisateur au clavier peut taper ;
- ligne 5, le serveur attend une commande tapée par l'utilisateur au clavier ;

Dans l'autre fenêtre de commandes, on lance le client TCP :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost
    100
2.  Client [DESKTOP-30FF5FB:51173] connecté au serveur [localhost-100]
3.  Tapez vos commandes (quit pour arrêter) :
```

- ligne 1, nous sommes placés dans le dossier des utilitaires ;
- ligne 1 nous lançons le client TCP : nous lui disons de se connecter au port 100 de la machine locale (celle sur laquelle s'exécute le code de **[RawTcpClient]**) ;
- ligne 2, le client a réussi à se connecter au serveur. On indique les coordonnées du client : il est sur la machine **[DESKTOP-30FF5FB]** (la machine locale dans cet exemple) et utilise le port **[51173]** pour communiquer avec le serveur :
- ligne 3, le client attend une commande tapée par l'utilisateur au clavier ;

Revenons sur la fenêtre du serveur. Son contenu a évolué :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpServer.exe 100
2.  server : Serveur générique lancé sur le port 0.0.0.0:100
3.  server : Attente d'un client...
4.  server : Commandes disponibles : [list, send id [texte], close id, quit]
```

```
5.  user : server : Client 1-DESKTOP-30FF5FB-51173 connecté...
6.  server : Attente d'un client...
```

- ligne 5, un client a été détecté. Le serveur lui a donné le n° 1. Le serveur a correctement identifié le client distant (machine et port) ;
- ligne 6, le serveur se remet en attente d'un nouveau client ;

Revenons sur la fenêtre du client et envoyons une commande au serveur :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost
    100
2.  Client [DESKTOP-30FF5FB:51173] connecté au serveur [localhost-100]
3.  Tapez vos commandes (quit pour arrêter) :
4.  hello from client
```

- ligne 4, la commande envoyée au serveur ;

Revenons sur la fenêtre du serveur. Son contenu a évolué :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpServer.exe 100
2.  server : Serveur générique lancé sur le port 0.0.0.0:100
3.  server : Attente d'un client...
4.  server : Commandes disponibles : [list, send id [texte], close id, quit]
5.  user : server : Client 1-DESKTOP-30FF5FB-51173 connecté...
6.  server : Attente d'un client...
7.  client 1 : [hello from client]
```

- ligne 7, entre crochets, le message reçu par le serveur ;

Envoyons une réponse au client :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpServer.exe 100
2.  server : Serveur générique lancé sur le port 0.0.0.0:100
3.  server : Attente d'un client...
4.  server : Commandes disponibles : [list, send id [texte], close id, quit]
5.  user : server : Client 1-DESKTOP-30FF5FB-51173 connecté...
6.  server : Attente d'un client...
7.  client 1 : [hello from client]
8.  send 1 [hello from server]
9.  user :
```

- ligne 8, la réponse envoyée au client 1. Seul le texte entre les crochets est envoyé, pas les crochets eux-mêmes ;

Revenons à la fenêtre du client :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost
    100
2.  Client [DESKTOP-30FF5FB:51173] connecté au serveur [localhost-100]
3.  Tapez vos commandes (quit pour arrêter) :
4.  hello from client
5.  <-- [hello from server]
```

- ligne 5, la réponse reçue par le client. Le texte reçu est celui entre crochets ;

Revenons à la fenêtre du serveur pour voir d'autres commandes :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpServer.exe 100
2.  server : Serveur générique lancé sur le port 0.0.0.0:100
3.  server : Attente d'un client...
4.  server : Commandes disponibles : [list, send id [texte], close id, quit]
5.  user : server : Client 1-DESKTOP-30FF5FB-51173 connecté...
6.  server : Attente d'un client...
7.  client 1 : [hello from client]
8.  send 1 [hello from server]
9.  user : list
10. server : id=1-name=DESKTOP-30FF5FB-51173
11. user : close 1
12. server : Connexion client 1 fermée...
13. user : quit
14. server : fin du service
```

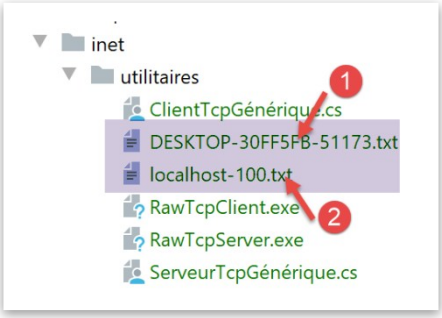
- ligne 9, nous demandons la liste des clients ;
- ligne 10, la réponse ;
- ligne 11, nous fermons la connexion avec le client n° 1 ;
- ligne 12, la confirmation du serveur ;
- ligne 13, nous arrêtons le serveur ;
- ligne 14, la confirmation du serveur ;

Revenons à la fenêtre du client :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost
    100
2.  Client [DESKTOP-30FF5FB:51173] connecté au serveur [localhost-100]
3.  Tapez vos commandes (quit pour arrêter) :
4.  hello from client
5.  <-- [hello from server]
6.  Perte de la connexion avec le serveur...
```

- ligne 6, le client a détecté la fin du service ;

Deux fichiers de logs ont été créés, un pour le serveur, un autre pour le client :



- en **[1]**, les logs du serveur : le nom du fichier est le nom du client sous la forme **[machine-port]**. Cela permet d'avoir des fichiers de logs différents pour des clients différents ;
- en **[2]**, les logs du client : le nom du fichier est le nom du serveur sous la forme **[machine-port]** ;

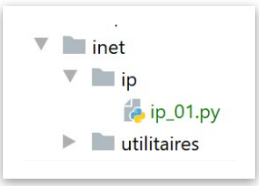
Les logs du serveur sont les suivants :

```
1.  <-- [hello from client]
1.  --> [hello from server]
```

Les logs du client sont les suivants :

```
1.  --> [hello from client]
2.  <-- [hello from server]
```

21.3 Obtenir le nom ou l'adresse IP d'une machine de l'Internet



Les machines de l'internet sont identifiées par une adresse IP (IPv4 ou IPv6) et le plus souvent par un nom. Mais finalement seule l'adresse IP est utilisée par les protocoles de communication de l'internet. Il faut donc connaître l'adresse IP d'une machine identifiée par son nom.

Le script **[ip-01.py]** est le suivant :

```
1.  # imports
2.  import socket
3.
4.
```

```
5.  # ------------------------------------------------
6.  def get_ip_and_name(nom_machine: str):
7.      # nom_machine : nom de la machine dont on veut l'adresse IP
8.      try:
9.          # nom_machine-->adresse IP
10.         ip = socket.gethostbyname(nom_machine)
11.         print(f"ip[{nom_machine}]={ip}")
12.     except socket.error as erreur:
13.         # on affiche l'erreur
14.         print(f"ip[{nom_machine}]={erreur}")
15.         return
16.
17.     try:
18.         # adresse IP --> nom_machine
19.         names = socket.gethostbyaddr(ip)
20.         print(f"names[{ip}]={names}")
21.     except socket.error as erreur:
22.         # on affiche l'erreur
23.         print(f"names[{ip}]={erreur}")
24.         return
25.
26.
27. # ---------------------------------------- main
28.
29. # les machines internet
30. hosts = ["istia.univ-angers.fr", "www.univ-angers.fr", "sergetahe.com", "localhost", "xx"]
31.
32. # adresses IP des machines de HOTES
33. for host in hosts:
34.     print("-------------------------------------")
35.     get_ip_and_name(host)
36. # fin
37. print("Terminé...")
```

Commentaires

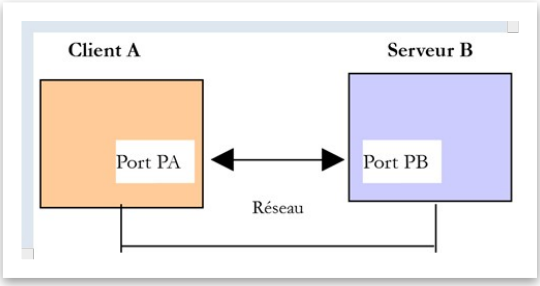
- ligne 2 : le module **[socket]** fournit les fonctions nécessaires à la gestion des sockets internet. **[socket]** signifie prise électrique, prise de réseau ;
- ligne 6 : la fonction **[get_ip_and_name]** permet à partir du nom internet d'une machine d'obtenir :
  - l'adresse IP de la machine ;
  - le nom de la machine obtenu à partir de l'adresse IP précédente ;
- ligne 10 : la fonction **[socket.gethostbyname]** permet d'obtenir l'adresse IP d'une machine à partir d'un de ces noms (une machine internet peut avoir un nom principal et des alias) ;
- ligne 12 : les fonctions sur les sockets lancent l'exception **[socket.error]** dès qu'une erreur survient ;
- ligne 19 : la fonction **[socket.gethostbyaddr]** permet d'obtenir le nom d'une machine à partir de son adresse IP. On va voir qu'on peut obtenir un nom différent de celui passé ligne 6 ;
- ligne 30 : une liste de noms de machines. Le dernier nom est erroné. Le nom **[localhost]** désigne la machine sur laquelle vous travaillez et qui exécute le script ;
- ligne 33-35 : on affiche les IP de ces machines ;

Résultats :

```
3.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/inet/ip/ip_01.py
4.  -------------------------------------
5.  ip[istia.univ-angers.fr]=193.49.144.41
6.  names[193.49.144.41]=('ametys-fo-2.univ-angers.fr', [], ['193.49.144.41'])
7.  -------------------------------------
8.  ip[www.univ-angers.fr]=193.49.144.41
9.  names[193.49.144.41]=('ametys-fo-2.univ-angers.fr', [], ['193.49.144.41'])
10. -------------------------------------
11. ip[sergetahe.com]=87.98.154.146
12. names[87.98.154.146]=('cluster026.hosting.ovh.net', [], ['87.98.154.146'])
13. -------------------------------------
14. ip[localhost]=127.0.0.1
15. names[127.0.0.1]=('DESKTOP-30FF5FB', [], ['127.0.0.1'])
16. -------------------------------------
17. ip[xx]=[Errno 11001] getaddrinfo failed
18. Terminé...
19.
20. Process finished with exit code 0
```

21.4 Le protocole HTTP (HyperText Transfer Protocol)

21.4.1 Exemple 1



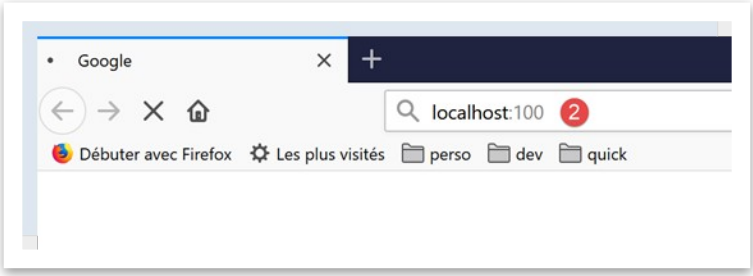
Lorsqu'un navigateur affiche une URL, il est le client d'un serveur web ou dit autrement d'un serveur HTTP. C'est lui qui prend l'initiative et il commence par envoyer un certain nombre de commandes au serveur. Pour ce premier exemple :

- le serveur sera l'utilitaire **[RawTcpServer]** ;
- le client sera un navigateur ;

Nous lançons d'abord le serveur sur le port 100 :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpServer.exe 100
2.  server : Serveur générique lancé sur le port 0.0.0.0:100
3.  server : Attente d'un client...
4.  server : Commandes disponibles : [list, send id [texte], close id, quit]
5.  user :
```

Puis avec un navigateur, nous demandons l'URL **[http://localhost:100]**, ç-a-d que nous disons que le serveur HTTP interrogé travaille sur le port 100 de la machine locale :



Revenons sur la fenêtre du serveur :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpServer.exe 100
2.  server : Serveur générique lancé sur le port 0.0.0.0:100
3.  server : Attente d'un client...
4.  server : Commandes disponibles : [list, send id [texte], close id, quit]
5.  user : server : Client 1-DESKTOP-30FF5FB-51438 connecté...
6.  server : Attente d'un client...
7.  server : Client 2-DESKTOP-30FF5FB-51439 connecté...
8.  server : Attente d'un client...
9.  client 1 : [GET / HTTP/1.1]
10. client 1 : [Host: localhost:100]
11. client 1 : [Connection: keep-alive]
12. client 1 : [DNT: 1]
13. client 1 : [Upgrade-Insecure-Requests: 1]
14. client 1 : [User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/83.0.4103.116 Safari/537.36]
15. client 1 : [Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/
    *;q=0.8,application/signed-exchange;v=b3;q=0.9]
16. client 1 : [Sec-Fetch-Site: none]
17. client 1 : [Sec-Fetch-Mode: navigate]
18. client 1 : [Sec-Fetch-User: ?1]
19. client 1 : [Sec-Fetch-Dest: document]
20. client 1 : [Accept-Encoding: gzip, deflate, br]
21. client 1 : [Accept-Language: fr-FR,fr;q=0.9,en-US;q=0.8,en;q=0.7]
22. client 1 : []
23. server : Client 3-DESKTOP-30FF5FB-51441 connecté...
24. server : Attente d'un client...
```

- ligne 5, le client qui s'est connecté ;
- lignes 9-22 : la série de lignes de texte qu'il a envoyées :
  - ligne 9 : cette ligne a le format **[GET URL HTTP/1.1]**. Elle demande l'URL / et demande au serveur d'utiliser le protocole HTTP 1.1 ;
  - ligne 10 : cette ligne a le format **[Host: serveur:port]**. La casse de la commande **[Host]** n'importe pas. On rappelle ici que le client interroge un serveur local opérant sur le port 100 ;
  - ligne 14 : la commande **[User-Agent]** donne l'identité du client ;
  - ligne 15 : la commande **[Accept]** indique quels types de document sont acceptés par le client ;
  - ligne 21 : la commande **[Accept-Language]** indique dans quelle langue sont souhaités les documents demandés s'ils existent en plusieurs langues ;
  - ligne 11 : la commande **[Connection]** indique le mode de connexion souhaité : **[keep-alive]** indique que la connexion doit être maintenue jusqu'à ce que les échanges soient terminés ;
  - ligne 22 : le client termine ses commandes par une ligne vide ;

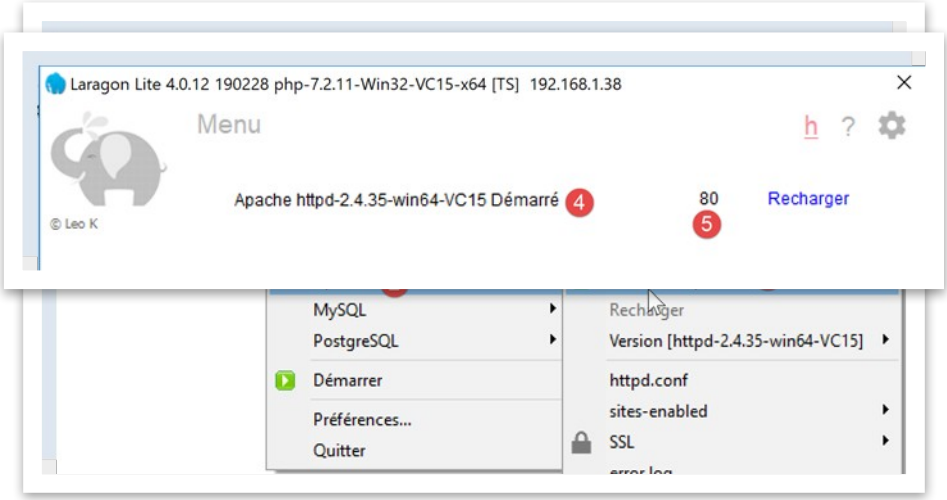
Nous terminons la connexion en terminant le serveur :

```
1.  client 1 : []
2.  server : Client 3-DESKTOP-30FF5FB-51441 connecté...
3.  server : Attente d'un client...
4.  quit
5.  server : fin du service
```

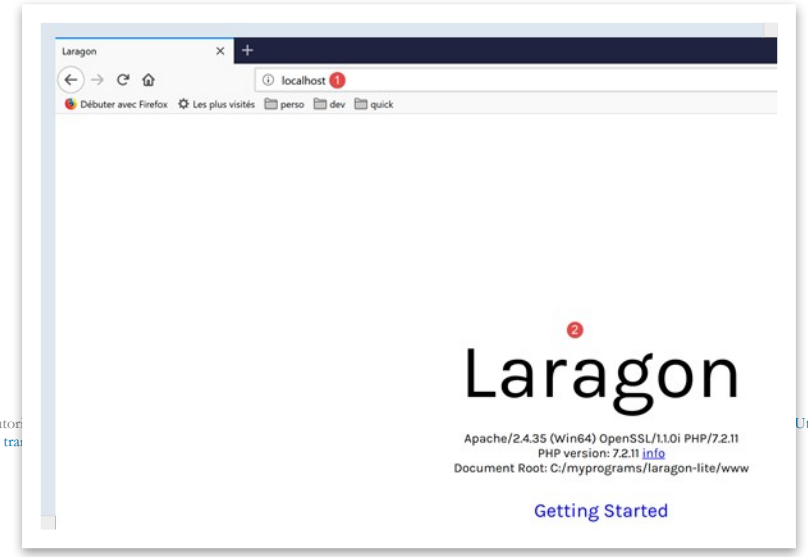
21.4.2 Exemple 2

Maintenant que nous connaissons les commandes envoyées par un navigateur pour réclamer une URL, nous allons réclamer cette URL avec notre client TCP **[RawTcpClient]**. Le serveur Apache de Laragon (paragraphe |Installation de Laragon|) sera notre serveur web.

Lançons Laragon puis le serveur web Apache :

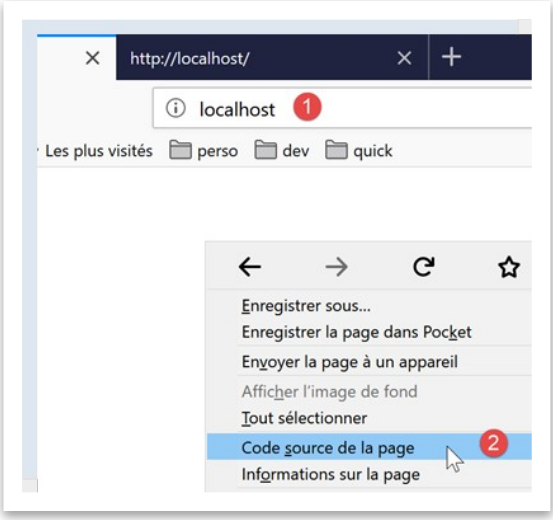


Maintenant avec un navigateur, demandons l'URL **[http://localhost:80]**. Ici nous ne précisons que le serveur **[localhost:80]** et pas d'URL de document. Dans ce cas c'est l'URL / qui est demandée, ç-à-d la racine du serveur web :



- en **[1]**, l'URL demandée. On a tapé initialement **[http://localhost:80]** et le navigateur (Firefox ici) l'a transformée simplement en **[localhost]** car le protocole **[http]** est implicite lorsqu'aucun protocole n'est mentionné et le port **[80]** est implicite lorsque le port n'est pas précisé ;
- en **[2]**, la page racine / du serveur web interrogé ;

Maintenant, visualisons le texte reçu par le navigateur :



- on clique droit sur la page reçue et on choisit l'option **[2]**. On obtient le code source suivant :

```
<!DOCTYPE html>
<html>
<head>
    <title>Laragon</title>

    <link href="https://fonts.googleapis.com/css?family=Karla:400" rel="stylesheet" type="text/css">

    <style>
        html, body {
            height: 100%;
        }

        body {
            margin: 0;
            padding: 0;
            width: 100%;
            display: table;
            font-weight: 100;
            font-family: 'Karla';
        }

        .container {
            text-align: center;
            display: table-cell;
            vertical-align: middle;
        }

        .content {
            text-align: center;
            display: inline-block;
        }

        .title {
            font-size: 96px;
        }

        .opt {
            margin-top: 30px;
        }

            .opt a {
                text-decoration: none;
                font-size: 150%;
```

```
            }

        a:hover {
            color: red;
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="content">
            <div class="title" title="Laragon">Laragon</div>

            <div class="info">
                <br />
                Apache/2.4.35 (Win64) OpenSSL/1.1.1b PHP/7.2.19<br />
                PHP version: 7.2.19   <span><a title="phpinfo()" href="/?q=info">info</a></span><br />
                Document Root: C:/MyPrograms/laragon/www<br />

            </div>
            <div class="opt">
                <div><a title="Getting Started" href="https://laragon.org/docs">Getting Started</a></div>
            </div>
        </div>

    </div>
</body>
</html>
```

Maintenant demandons l'URL **[http://localhost:80]** avec notre client TCP :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost 80
2.  Client [DESKTOP-30FF5FB:51541] connecté au serveur [localhost-80]
3.  Tapez vos commandes (quit pour arrêter) :
```

- ligne 1, nous nous connectons au port 80 du serveur *localhost*. C'est là qu'opère le serveur web de Laragon ;

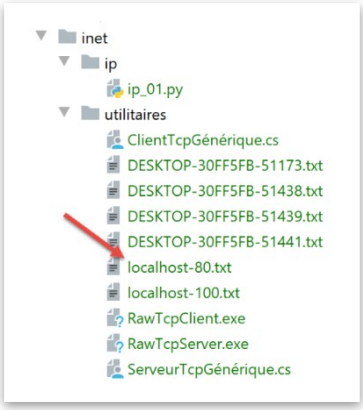
Nous tapons maintenant les commandes que nous avons découvertes dans le paragraphe précédent :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost
    80
2.  Client [DESKTOP-30FF5FB:51544] connecté au serveur [localhost-80]
3.  Tapez vos commandes (quit pour arrêter) :
4.  GET / HTTP/1.1
5.  Host: localhost:80
6.
7.  <-- [HTTP/1.1 200 OK]
8.  <-- [Date: Sun, 05 Jul 2020 12:42:14 GMT]
9.  <-- [Server: Apache/2.4.35 (Win64) OpenSSL/1.1.1b PHP/7.2.19]
10. <-- [X-Powered-By: PHP/7.2.19]
11. <-- [Content-Length: 1776]
12. <-- [Content-Type: text/html; charset=UTF-8]
13. <-- []
14. <-- [<!DOCTYPE html>]
15. <-- [<html>]
16. <-- [    <head>]
17. <-- [        <title>Laragon</title>]
18. <-- []
19. <-- [        <link href="https://fonts.googleapis.com/css?family=Karla:400" rel="stylesheet"
    type="text/css">]
20. <-- []
21. <-- [        <style>]
22. <-- [            html, body {]
23. <-- [                height: 100%;]
24. <-- [            }]
25. <-- []
26. <-- [            body {]
27. <-- [                margin: 0;]
28. <-- [                padding: 0;]
29. <-- [                width: 100%;]
30. <-- [                display: table;]
31. <-- [                font-weight: 100;]
32. <-- [                font-family: 'Karla';]
33. <-- [            }]
34. <-- []
35. <-- [            .container {]
36. <-- [                text-align: center;]
```

```
37. <-- [                display: table-cell;]
38. <-- [                vertical-align: middle;]
39. <-- [            }]
40. <-- []
41. <-- [            .content {]
42. <-- [                text-align: center;]
43. <-- [                display: inline-block;]
44. <-- [            }]
45. <-- []
46. <-- [            .title {]
47. <-- [                font-size: 96px;]
48. <-- [            }]
49. <-- []
50. <-- [            .opt {]
51. <-- [                margin-top: 30px;]
52. <-- [            }]
53. <-- []
54. <-- [            .opt a {]
55. <-- [              text-decoration: none;]
56. <-- [              font-size: 150%;]
57. <-- [            }]
58. <-- [            ]
59. <-- [            a:hover {]
60. <-- [              color: red;]
61. <-- [            }]
62. <-- [        </style>]
63. <-- [    </head>]
64. <-- [    <body>]
65. <-- [        <div class="container">]
66. <-- [            <div class="content">]
67. <-- [                <div class="title" title="Laragon">Laragon</div>]
68. <-- [     ]
69. <-- [                <div class="info"><br />]
70. <-- [                      Apache/2.4.35 (Win64) OpenSSL/1.1.1b PHP/7.2.19<br />]
71. <-- [                      PHP version: 7.2.19   <span><a title="phpinfo()"
    href="/?q=info">info</a></span><br />]
72. <-- [                      Document Root: C:/MyPrograms/laragon/www<br />]
73. <-- []
74. <-- [                </div>]
75. <-- [                <div class="opt">]
76. <-- [                  <div><a title="Getting Started" href="https://laragon.org/docs">Getting
    Started</a></div>]
77. <-- [                </div>]
78. <-- [            </div>]
79. <-- []
80. <-- [        </div>]
81. <-- [    </body>]
82. <-- [</html>]
83. Perte de la connexion avec le serveur...
```

- ligne 4, la commande **[GET]**. On demande la racine / du serveur web ;
- ligne 5, la commande **[Host]** ;
- ce sont les deux seules commandes indispensables. Pour les autres commandes, le serveur web prendra des valeurs par défaut ;
- ligne 6, la ligne vide qui doit terminer les commandes du client ;
- dessous la ligne 6, vient la réponse du serveur web ;
- lignes 7-12 : les entêtes HTTP de la réponse du serveur ;
- ligne 13 : la ligne vide qui signale la fin des entêtes http ;
- lignes 14-82, le document HTML demandé ligne 4 ;

Nous chargeons le fichier de logs **[localhost-80.txt]** :



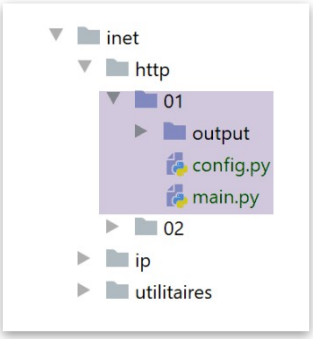

```
1.  --> [GET / HTTP/1.1]
2.  --> [Host: localhost:80]
3.  --> []
4.  <-- [HTTP/1.1 200 OK]
5.  <-- [Date: Sun, 05 Jul 2020 12:42:14 GMT]
6.  <-- [Server: Apache/2.4.35 (Win64) OpenSSL/1.1.1b PHP/7.2.19]
7.  <-- [X-Powered-By: PHP/7.2.19]
8.  <-- [Content-Length: 1776]
9.  <-- [Content-Type: text/html; charset=UTF-8]
10. <-- []
11. <-- [<!DOCTYPE html>]
12. <-- [<html>]
13. <-- [    <head>]
14. <-- [        <title>Laragon</title>]
15. <-- []
16. <-- [        <link href="https://fonts.googleapis.com/css?family=Karla:400" rel="stylesheet"
    type="text/css">]
17. <-- []
18. <-- [        <style>]
19. <-- [            html, body {]
20. <-- [                height: 100%;]
21. <-- [            }]
22. <-- []
23. <-- [            body {]
24. <-- [                margin: 0;]
25. <-- [                padding: 0;]
26. <-- [                width: 100%;]
27. <-- [                display: table;]
28. <-- [                font-weight: 100;]
29. <-- [                font-family: 'Karla';]
30. <-- [            }]
31. <-- []
32. <-- [            .container {]
33. <-- [                text-align: center;]
34. <-- [                display: table-cell;]
35. <-- [                vertical-align: middle;]
36. <-- [            }]
37. <-- []
38. <-- [            .content {]
39. <-- [                text-align: center;]
40. <-- [                display: inline-block;]
41. <-- [            }]
42. <-- []
43. <-- [            .title {]
44. <-- [                font-size: 96px;]
45. <-- [            }]
46. <-- []
47. <-- [            .opt {]
48. <-- [                margin-top: 30px;]
49. <-- [            }]
50. <-- []
51. <-- [            .opt a {]
52. <-- [              text-decoration: none;]
53. <-- [              font-size: 150%;]
54. <-- [            }]
55. <-- [            ]
56. <-- [            a:hover {]
```

```
57. <-- [              color: red;]
58. <-- [            }]
59. <-- [        </style>]
60. <-- [    </head>]
61. <-- [    <body>]
62. <-- [        <div class="container">]
63. <-- [            <div class="content">]
64. <-- [                <div class="title" title="Laragon">Laragon</div>]
65. <-- [     ]
66. <-- [                <div class="info"><br />]
67. <-- [                      Apache/2.4.35 (Win64) OpenSSL/1.1.1b PHP/7.2.19<br />]
68. <-- [                      PHP version: 7.2.19   <span><a title="phpinfo()"
    href="/?q=info">info</a></span><br />]
69. <-- [                      Document Root: C:/MyPrograms/laragon/www<br />]
70. <-- []
71. <-- [                </div>]
72. <-- [                <div class="opt">]
73. <-- [                  <div><a title="Getting Started" href="https://laragon.org/docs">Getting
    Started</a></div>]
74. <-- [                </div>]
75. <-- [            </div>]
76. <-- []
77. <-- [        </div>]
78. <-- [    </body>]
79. <-- [</html>]
```

- lignes 11-79 : le document HTML reçu. Dans l'exemple précédent, Firefox avait reçu le même ;

Nous avons désormais les bases pour programmer un client TCP qui demanderait une URL.

21.4.3 Exemple 3



Le script **[http/01/main.py]** est un client HTTP configuré par le fichier **[config.py]**. Le contenu de celui-ci est le suivant :

```
1.  def configure():
2.      # URLs à interroger
3.      urls = [
4.          # site : nom du site auquel se connecter
5.          # port : port du service web
6.          # GET : URL demandée
7.          # headers : entêtes HTTP à envoyer dans la requête
8.          # endOfLine : marque de fin de ligne dans les entêtes HTTP envoyés
9.          # encoding : encodage de la réponse du serveur
10.         # timeout : temps d'attente maximum d'une réponse du serveur
11.         {
12.             "site": "localhost",
13.             "port": 80,
14.             "GET": "/",
15.             "headers": {
16.                 "Host": "localhost:80",
17.                 "User-Agent": "client Python",
18.                 "Accept": "text/HTML",
19.                 "Accept-Language": "fr"
20.             },
21.             "endOfLine": "\r\n",
22.             "encoding": "utf-8",
23.             "timeout": 0.5
24.         },
```

```
25.         {
26.             "site": "sergetahe.com",
27.             "port": 80,
28.             "GET": "/",
29.             "headers": {
30.                 "Host": "sergetahe.com:80",
31.                 "User-Agent": "client Python",
32.                 "Accept": "text/HTML",
33.                 "Accept-Language": "fr"
34.             },
35.             "endOfLine": "\r\n",
36.             "encoding": "utf-8",
37.             "timeout": 5
38.         },
39.         {
40.             "site": "tahe.developpez.com",
41.             "port": 443,
42.             "GET": "/",
43.             "headers": {
44.                 "Host": "tahe.developpez.com:443",
45.                 "User-Agent": "client Python",
46.                 "Accept": "text/HTML",
47.                 "Accept-Language": "fr"
48.             },
49.             "endOfLine": "\r\n",
50.             "encoding": "utf-8",
51.             "timeout": 2
52.         },
53.         {
54.             "site": "www.sergetahe.com",
55.             "port": 80,
56.             "GET": "/cours-tutoriels-de-programmation/",
57.             "headers": {
58.                 "Host": "sergetahe.com:80",
59.                 "User-Agent": "client Python",
60.                 "Accept": "text/HTML",
61.                 "Accept-Language": "fr"
62.             },
63.             "endOfLine": "\r\n",
64.             "encoding": "utf-8",
65.             "timeout": 5
66.         }
67.     ]
68.     # on rend la configuration
69.     return {
70.         "urls": urls
71.     }
```

- le contenu du fichier est une liste d'URL, chaque élément de la liste étant un dictionnaire. Ce dictionnaire indique comment se connecter au sité désigné par la clé **[site]** ;
- lignes 4-10 : la signification des clés de chaque dictionnaire ;

Le script **[http/01/main.py]** est le suivant :

```
1.  # imports
2.  import codecs
3.  import socket
4.
5.
6.  # ------------------------------------------------------------------------
7.  def get_url(url: dict, suivi: bool = True):
8.      # lit l'URL url["GET"] du site url[site] et la stocke dans le fichier url[site].html
9.      # le dialogue client /serveur se fait selon le protocole HTTP indiqué dans le dictionnaire [url]
10.     # on laisse remonter les exceptions
11.
12.     sock = None
13.     html = None
14.     try:
15.         # connexion à [site] sur le port 80 avec un timeout
16.         site = url['site']
17.         sock = socket.create_connection((site, int(url['port'])), float(url['timeout']))
18.
19.         # connexion représente un flux de communication bidirectionnel
20.         # entre le client (ce programme) et le serveur web contacté
21.         # ce canal est utilisé pour les échanges de commandes et d'informations
22.         # le protocole de dialogue est HTTP
```

```
23.
24.         # création du fichier site.html - on change les caractères gênants pour un nom de fichier
25.         site2 = site.replace("/", "_")
26.         site2 = site2.replace(".", "_")
27.         html_filename = f'{site2}.html'
28.         html = codecs.open(f"output/{html_filename}", "w", "utf-8")
29.
30.         # le client va commencer le dialogue HTTP avec le serveur
31.         if suivi:
32.             print(f"Client : début de la communication avec le serveur [{site}]")
33.
34.         # selon les serveurs, les lignes du client doivent se terminer par \n ou \r\n
35.         end_of_line = url["endOfLine"]
36.         # le client envoie la commande GET pour demander l'URL config["GET"]
37.         # syntaxe GET URL HTTP/1.1
38.         commande = f"GET {url['GET']} HTTP/1.1{end_of_line}"
39.         # suivi ?
40.         if suivi:
41.             print(f"--> {commande}", end='')
42.         # on envoie la commande au serveur
43.         sock.send(bytearray(commande, 'utf-8'))
44.         # émission des entêtes HTTP
45.         for verb, value in url['headers'].items():
46.             # on construit la commande à envoyer
47.             commande = f"{verb}: {value}{end_of_line}"
48.             # suivi ?
49.             if suivi:
50.                 print(f"--> {commande}", end='')
51.             # on envoie la commande au serveur
52.             sock.send(bytearray(commande, 'utf-8'))
53.         # on envoie l'entête HTTP [Connection: close] pour demander au serveur web
54.         # de fermer la connexion lorsqu'il aura envoyé le document demandé
55.         sock.send(bytearray(f"Connection: close{end_of_line}", 'utf-8'))
56.         # les entêtes (headers) du protocole HTTP doivent se terminer par une ligne vide
57.         sock.send(bytearray(end_of_line, 'utf-8'))
58.         #
59.         # le serveur va maintenant répondre sur le canal sock. Il va envoyer toutes
60.         # ses données puis fermer le canal. Le client lit donc tout ce qui arrive de sock
61.         # jusqu'à la fermeture du canal
62.         #
63.         # on lit tout d'abord les entêtes HTTP envoyés par le serveur
64.         # ils se terminent eux-aussi par une ligne vide
65.         if suivi:
66.             print(f"Réponse du serveur [{site}]")
67.
68.         # lecture de la socket comme si elle était un fichier texte
69.         encoding = f"{url['encoding']}" if url['encoding'] else None
70.         if encoding:
71.             file = sock.makefile(encoding=encoding)
72.         else:
73.             file = sock.makefile()
74.         # on exploite ce fichier ligne par ligne
75.         fini = False
76.         while not fini:
77.             # lecture ligne courante
78.             ligne = file.readline().strip()
79.             # a-t-on une ligne non vide ?
80.             if ligne:
81.                 if suivi:
82.                     # on affiche l'entête HTTP
83.                     print(f"<-- {ligne}")
84.             else:
85.                 # c'était la ligne vide - les entêtes HTTP sont terminés
86.                 fini = True
87.         # on lit le document HTML qui va suivre la ligne vide
88.         # lecture ligne courante
89.         ligne = file.readline()
90.         while ligne:
91.             # enregistrement dans le fichier de logs
92.             html.write(str(ligne))
93.             # ligne suivante
94.             ligne = file.readline()
95.             # la boucle se finit lorsque le serveur ferme la connexion
96.     finally:
97.         # le client ferme la connexion
98.         if sock:
99.             sock.close()
```

```
100.        # fermeture du fichier html
101.        if html:
102.            html.close()
103.
104.
105.# -------------------main
106.
107.# on configure l'application
108.import config
109.config = config.configure()
110.
111.# obtenir les URL du fichier de configuration
112.for url in config['urls']:
113.    print("-------------------------")
114.    print(url['site'])
115.    print("-------------------------")
116.    try:
117.        # lecture URL du site [site]
118.        get_url(url)
119.    except BaseException as erreur:
120.        print(f"L'erreur suivante s'est produite : {erreur}")
121.    finally:
122.        pass
123.# fin
124.print("Terminé...")
```

Commentaires du code :

- lignes 108-109 : le dictionnaire **[config]** du module **[config.py]** est récupéré ;
- ligne 111-122 : ce dictionnaire est exploité ;
- ligne 118, 7 : la fonction **[get_url(url)]** demande un document du site web *url***[site]** et le stocke dans le fichier texte *url***[site]**.*HTML*. Par défaut, les échanges client/serveur sont logués sur la console (suivi=True) ;
- on fait tout dans un **[try / finally]** (lignes 14-96). Il n'y a pas de clause **[except]**. Les exceptions vont remonter au code appelant et c'est celui-ci qui les arrête et les affiche (lignes 119-120) ;
- lignes 16-17 : ouverture d'une connexion vers le serveur web. La fonction **[socket.create_connection]** admet trois paramètres :
  - **[param1]** : est le nom de la machine de l'internet qu'on veut atteindre ;
  - **[param2]** : est le n° du port du service auquel on veut se connecter ;
  - **[param3]** : **[socket.create_connection]** rend un socket et **[param3]**, s'il est présent, désigne le timeout du socket créé. Le timeout est le délai maximal d'attente du socket lorsqu'il attend une réponse de la machine distante ;
- lignes 27-28 : création du fichier **[site.html]** dans lequel on stockera le document HTML reçu ;
- lignes 34-43 : la première commande du client doit être la commande **[GET URL HTTP/1.1]** ;
- ligne 43 : la fonction **[sock.send]** permet au client d'envoyer des données au serveur. Ici la ligne de texte envoyée a la signification suivante : "Je veux (GET) la page **[URL]** du site web auquel je suis connecté. Je travaille avec le protocole HTTP version 1.1" ;
- ligne 43 : l'instruction **[sock.send(bytearray(commande, 'utf-8'))]** envoie un tableau d'octets octets (bytearray). Ce tableau est obtenu par conversion de la chaîne **[commande]** en une suite d'octets codés en UTF-8 ;
- lignes 44-52 : on envoie les autres lignes du protocole HTTP **[Host, User-Agent, Accept, Accept-Language…]**. Leur ordre n'importe pas ;
- lignes 53-55 : on envoie l'entête HTTP **[Connection: close]** pour demander au serveur de fermer sa connexion lorsqu'il aura envoyé le document demandé. Par défaut il ne le fait pas. Il faut donc le lui demander explicitement. L'intérêt est que cette fermeture va être détectée côté client et c'est comme cela que celui-ci saura qu'il aura reçu tout le document demandé ;
- lignes 56-57 : on envoie une ligne vide au serveur pour signifier que le client a terminé d'envoyer ses entêtes HTTP et qu'il attend désormais le document demandé ;
- lignes 68-86 : le serveur va tout d'abord envoyer une série d'entêtes HTTP qui vont donner diverses informations sur le document demandé. Ces entêtes se terminent par une ligne vide ;
- lignes 69-73 : pour pouvoir lire la réponse du serveur, ligne par ligne, on utilise la méthode **[sock.makefile(encoding=encoding)]**. Le paramètre facultatif **[encoding]** précise l'encodage du texte attendu. Après cette opération, le flux de lignes envoyées par le serveur va pouvoir être lu comme un fichiet texte classique ;
- ligne 78 : on lit une ligne envoyée par le serveur avec la méthode **[readline]**. On la débarrasse de ses espaces (blancs, marque de fin de ligne) de début et fin de ligne ;
- lignes 81-83 : si la ligne n'est pas vide et que le suivi a été demandé, la ligne reçue est affichée sur la console ;
- lignes 84-86 : si on a récupéré la ligne vide qui marque la fin des entêtes HTTP envoyés par le serveur alors on arrête la boucle de la ligne 76 ;
- lignes 90-95 : les lignes de texte de la réponse du serveur peuvent être lues ligne par ligne avec une boucle *while* et enregistrées dans le fichier texte **[html]**. Lorsque le serveur web a envoyé la totalité de la page qu'on lui a demandée, il ferme sa connexion avec le client. Côté client, cela sera détecté comme une fin de fichier et on sortira de la boucle des lignes 90-95 ;
- lignes 96-102 : erreur ou pas, on libère toutes les ressources utilisées par le code ;

Résultats :

La console affiche les logs suivants :

```
C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/inet/http/01/main.py
-------------------------
localhost
-------------------------
Client : début de la communication avec le serveur [localhost]
--> GET / HTTP/1.1
--> Host: localhost:80
--> User-Agent: client Python
--> Accept: text/HTML
--> Accept-Language: fr
Réponse du serveur [localhost]
<-- HTTP/1.1 200 OK
<-- Date: Sun, 05 Jul 2020 16:27:46 GMT
<-- Server: Apache/2.4.35 (Win64) OpenSSL/1.1.1b PHP/7.2.19
<-- X-Powered-By: PHP/7.2.19
<-- Content-Length: 1776
<-- Connection: close
<-- Content-Type: text/html; charset=UTF-8
-------------------------
sergetahe.com
-------------------------
Client : début de la communication avec le serveur [sergetahe.com]
--> GET / HTTP/1.1
--> Host: sergetahe.com:80
--> User-Agent: client Python
--> Accept: text/HTML
--> Accept-Language: fr
Réponse du serveur [sergetahe.com]
<-- HTTP/1.1 302 Found
<-- Date: Sun, 05 Jul 2020 16:27:45 GMT
<-- Content-Type: text/html; charset=UTF-8
<-- Transfer-Encoding: chunked
<-- Connection: close
<-- Server: Apache
<-- X-Powered-By: PHP/7.3
<-- Location: http://sergetahe.com:80/cours-tutoriels-de-programmation
<-- Set-Cookie: SERVERID68971=2620178|XwH/h|XwH/h; path=/
<-- X-IPLB-Instance: 17106
-------------------------
tahe.developpez.com
-------------------------
Client : début de la communication avec le serveur [tahe.developpez.com]
--> GET / HTTP/1.1
--> Host: tahe.developpez.com:443
--> User-Agent: client Python
--> Accept: text/HTML
--> Accept-Language: fr
Réponse du serveur [tahe.developpez.com]
<-- HTTP/1.1 400 Bad Request
<-- Date: Sun, 05 Jul 2020 16:27:45 GMT
<-- Server: Apache/2.4.38 (Debian)
<-- Content-Length: 453
<-- Connection: close
<-- Content-Type: text/html; charset=iso-8859-1
-------------------------
www.sergetahe.com
-------------------------
Client : début de la communication avec le serveur [www.sergetahe.com]
--> GET /cours-tutoriels-de-programmation/ HTTP/1.1
--> Host: sergetahe.com:80
--> User-Agent: client Python
--> Accept: text/HTML
--> Accept-Language: fr
Réponse du serveur [www.sergetahe.com]
<-- HTTP/1.1 301 Moved Permanently
<-- Date: Sun, 05 Jul 2020 16:27:45 GMT
<-- Content-Type: text/html; charset=iso-8859-1
<-- Content-Length: 263
<-- Connection: close
<-- Server: Apache
```

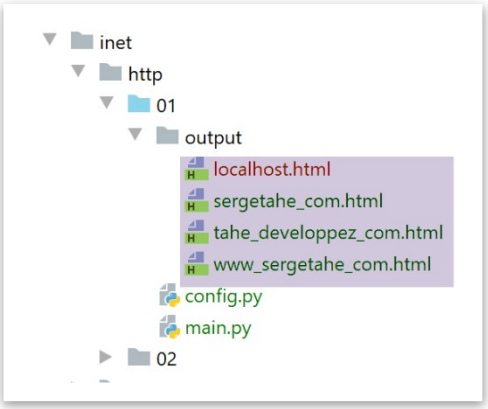
```
71. <-- Location: https://sergetahe.com/cours-tutoriels-de-programmation/
72. <-- Set-Cookie: SERVERID68971=2620178|XwH/h|XwH/h; path=/
73. <-- X-IPLB-Instance: 17095
74. Terminé...
75.
76. Process finished with exit code 0
```

Commentaires

- ligne 12 : l'URL **[http://localhost/]** a été trouvée (code 200) ;
- ligne 29 : l'URL **[http://sergetahe.com/]** n'a pas été trouvée (code 302). Le code 302 signifie que la page demandée a changé d'URL. La nouvelle URL est indiquée par l'entête HTTP **[Location]** de la ligne 36 ;
- ligne 49 : la requête qui a été faite au serveur **[http://tahe.developpez.com]** est incorrecte (code 400) ;
- ligne 65 : l'URL **[http://www.sergetahe.com/]** n'a pas été trouvée (code 301). Le code 301 signifie que la page demandée a changé d'URL et ce de façon définitive. La nouvelle URL est indiquée par l'entête HTTP **[Location]** de la ligne 71 ;

De façon générale les codes 3xx, 4xx et 5xx d'un serveur HTTP sont des codes d'erreur.

L'exécution a produit fichiers :



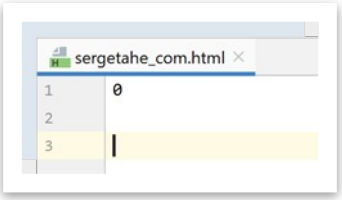
Le fichier **[output/localhost.HTML]** reçu est le suivant :

```
1.  <!DOCTYPE html>
2.  <html>
3.      <head>
4.          <title>Laragon</title>
5.
6.          <link href="https://fonts.googleapis.com/css?family=Karla:400" rel="stylesheet" type="text/css">
7.
8.          <style>
9.              html, body {
10.                 height: 100%;
11.             }
12.
13.             body {
14.                 margin: 0;
15.                 padding: 0;
16.                 width: 100%;
17.                 display: table;
18.                 font-weight: 100;
19.                 font-family: 'Karla';
20.             }
21.
22.             .container {
23.                 text-align: center;
24.                 display: table-cell;
25.                 vertical-align: middle;
26.             }
27.
28.             .content {
29.                 text-align: center;
30.                 display: inline-block;
31.             }
32.
33.             .title {
```

```
34.                 font-size: 96px;
35.             }
36.
37.             .opt {
38.                 margin-top: 30px;
39.             }
40.
41.             .opt a {
42.               text-decoration: none;
43.               font-size: 150%;
44.             }
45.
46.             a:hover {
47.               color: red;
48.             }
49.         </style>
50.     </head>
51.     <body>
52.         <div class="container">
53.             <div class="content">
54.                 <div class="title" title="Laragon">Laragon</div>
55.
56.                 <div class="info"><br />
57.                       Apache/2.4.35 (Win64) OpenSSL/1.1.1b PHP/7.2.19<br />
58.                       PHP version: 7.2.19   <span><a title="phpinfo()" href="/?q=info">info</a></span><br />
59.                       Document Root: C:/MyPrograms/laragon/www<br />
60.
61.                 </div>
62.                 <div class="opt">
63.                   <div><a title="Getting Started" href="https://laragon.org/docs">Getting Started</a></div>
64.                 </div>
65.             </div>
66.
67.         </div>
68.     </body>
69. </html>
```

Nous avons bien obtenu le même document qu'avec le navigateur Firefox.

Le document **[output/sergetahe_com.html]** reçu est le suivant :



La plupart des serveurs http envoient par morceaux leurs réponses aux requêtes qui leur sont faites. Chaque morceau envoyé est précédé d'une ligne indiquant le nombre d'octets du morceau qui suit. Cela permet au client de lire ce nombre exact d'octets pour avoir le morceau. Ici le 0 indique que le morceau qui suit a zéro octet. On rappelle que le serveur avait indiqué le document **[http://sergetahe.com/]** avait changé d'URL. Il n'a donc pas envoyé de document.

Le document **[output/tahe_developpez_com.html]** est le suivant :

```
1.  <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
2.  <html><head>
3.  <title>400 Bad Request</title>
4.  </head><body>
5.  <h1>Bad Request</h1>
6.  <p>Your browser sent a request that this server could not understand.<br />
7.  Reason: You're speaking plain HTTP to an SSL-enabled server port.<br />
8.   Instead use the HTTPS scheme to access this URL, please.<br />
9.  </p>
10. <hr>
11. <address>Apache/2.4.38 (Debian) Server at 2eurocents.developpez.com Port 80</address>
12. </body></html>
```

- lignes 1-12 : le serveur a envoyé un document HTML malgré le fait que la requête était incorrecte (ligne 49 des résultats). Le document HTML permet au serveur de préciser la cause de l'erreur. Celle-ci est indiquée aux lignes 6 et 7 :
  - ligne 7 : notre client a utilisé le protocole HTTP ;
  - ligne 8 : le serveur travaille avec le protocole HTTPS (S=sécurisé) et n'accepte pas le protocole HTTP ;

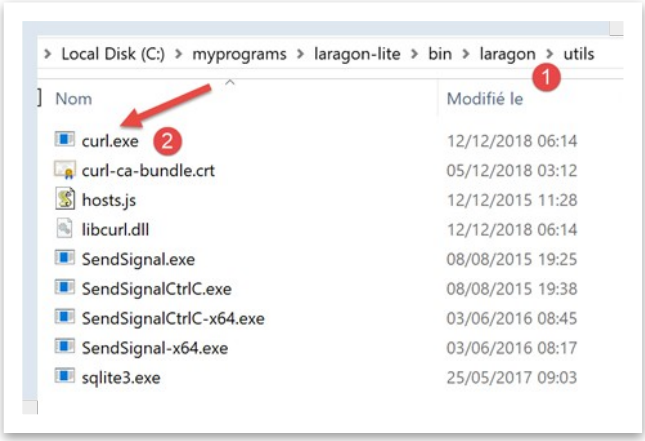
Le document **[output/www_sergetahe_com.html]** est le suivant :

```
1.  <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
2.  <html><head>
3.  <title>301 Moved Permanently</title>
4.  </head><body>
5.  <h1>Moved Permanently</h1>
6.  <p>The document has moved <a href="https://sergetahe.com/cours-tutoriels-de-programmation/">here</a>.</p>
7.  </body></html>
```

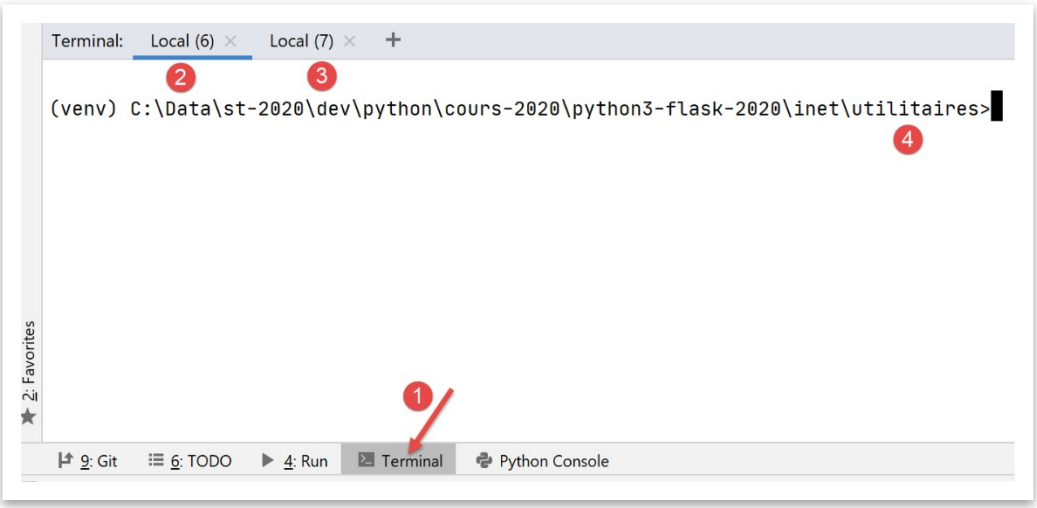
Là également, il s'est produit une erreur (ligne 3). Néanmoins, le serveur prend soin d'envoyer un document HTML détaillant celle-ci (lignes 1-7).

21.4.4 Exemple 4

Les exemples précédents nous ont montré que notre client HTTP était insuffisant. Nous allons maintenant présenter un outil appelé **[curl]** qui permet de récupérer des documents web en gérant les difficultés mentionnées : protocole HTTPS, document envoyé par morceaux, redirections… L'outil **[curl]** a été installé avec Laragon :



Ouvrons un terminal PyCharm **[1]** :



- en **[1]**, l'accès aux terminaux de PyCharm ;
- en **[2-3]**, les terminaux déjà actifs ;
- en **[4]**, le dossier dans lequel vous êtes. Dans ce qui suit il n'importe pas ;

Dans le terminal nous tapons la commande suivante :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>curl --help
2.  Usage: curl [options...] <url>
3.       --abstract-unix-socket <path> Connect via abstract Unix domain socket
4.       --anyauth       Pick any authentication method
5.    -a, --append       Append to target file when uploading
6.       --basic         Use HTTP Basic Authentication
```

```
7.        --cacert <CA certificate> CA certificate to verify peer against
8.   …
```

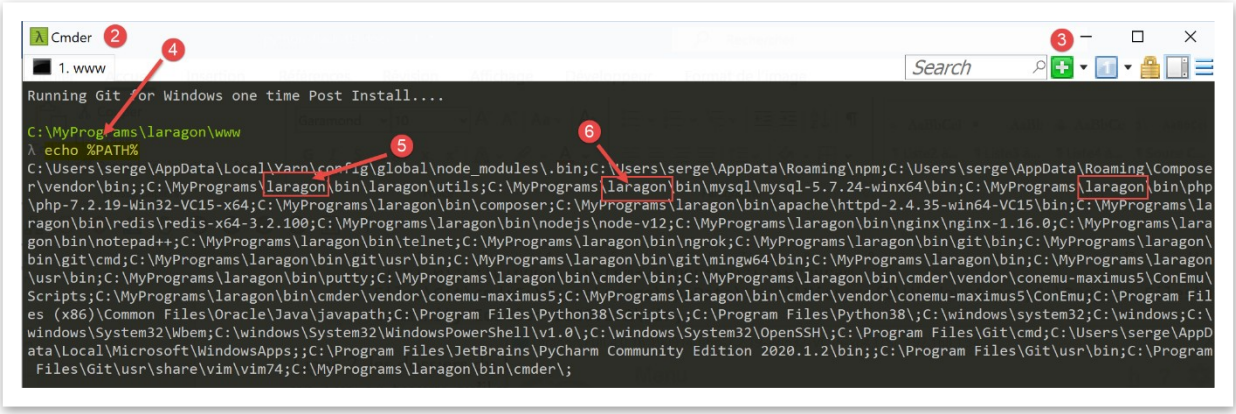
Le fait que la commande **[curl -help]** ait produit des résultats montre que la commande **[curl]** est dans le PATH du terminal. Sous Windows, le PATH est l'ensemble des dossiers explorés lorsque l'utilisateur tape une commande exécutable, ici **[curl]**. La valeur du PATH peut être connue :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>echo %PATH%
2.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts;C:\Program Files (x86)\Common
    Files\Oracle\Java\javapath;C:\Program Files\Python38\Scripts\;C:\Program Files\Python38\;C:
    \windows\system32;C:\windows;C:\windows\System32\Wbem;C:\windows\System32\WindowsPowerShell\v1.0\;C:
    \windows\System32\OpenSSH\;C:\Program Files\Git\cmd;C:\Users\serge\AppData\Local\Microsoft\WindowsApps;;C:
    \Program Files\JetBrains\PyCharm Community Edition 2020.1.2\bin;
```

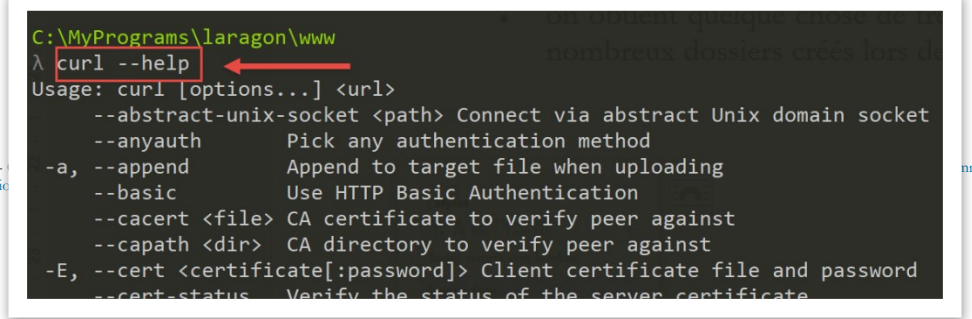
Ligne 2, les dossiers du PATH séparés par des points-virgules. Dans cette liste n'apparaît pas de dossier lié à Laragon. Si on enquête un peu, on trouve qu'il y a un **[curl]** dans le dossier **[c:\windows\system32]**. C'est celui-ci qui a répondu auparavant.

Si on veut utiliser l'outil **[curl]** livré avec Laragon, on pourra procéder comme suit :





- en **[2]**, le terminal Laragon ;
- en **[3]**, ce bouton permet de créer de nouveaux terminaux, chacun s'installant dans un onglet de la fenêtre ci-dessus ;
- en **[4]**, on demande le PATH du terminal Laragon ;
- on obtient quelque chose de très différent de ce qui avait été obtenu dans un terminal PyCharm. Ce PATH contient de nombreux dossiers créés lors de l'installation de Laragon. Le dossier contenant l'outil **[curl]** en fait partie :



Par la suite, utilisez le terminal de votre choix. Sachez simplement que lorsque vous voulez utiliser un outil amené par Laragon, le terminal Laragon est à préférer.

La commande **[curl --help]** fait afficher toutes les options de configuration de **[curl]**. Il y en a plusieurs dizaines. Nous en utiliserons très peu. Pour demander une URL il suffit de taper la commande **[curl URL]**. Cette commande affichera sur la console le document demandé. Si on veut de plus les échanges HTTP entre le client et le serveur on écrira **[curl --verbose URL]**. Enfin pour enregistrer le document HTML demandé dans un fichier on écrira **[curl --verbose --output fichier URL]**.

Pour éviter de polluer le système de fichiers de notre machine, déplaçons-nous à un autre endroit (j'utilise ici un terminal Laragon) :

```
3.  λ cd \Temp\
4.
5.  C:\Temp
6.  λ mkdir curl
7.
8.  C:\Temp
9.  λ cd curl\
10.
11. C:\Temp\curl
12. λ dir
13.  Le volume dans le lecteur C s'appelle Local Disk
14.  Le numéro de série du volume est B84C-D958
15.
16.  Répertoire de C:\Temp\curl
17.
18. 05/07/2020  19:31    <DIR>          .
19. 05/07/2020  19:31    <DIR>          ..
20.               0 fichier(s)              0 octets
21.               2 Rép(s)  892 388 098 048 octets libres
```

- ligne 3, on se déplace dans le dossier **[c:\temp]**. Si ce dossier n'existe pas, vous pouvez le créer ou en choisir un autre ;
- ligne 6, on crée un dossier appelé **[curl]** ;
- ligne 9, on se positionne dessus ;
- ligne 12, on liste son contenu. Il est vide (ligne 20);

Assurez-vous que le serveur Apache de Laragon est lancé et avec **[curl]** demandez l'URL **[http://localhost/]** avec la commande **[curl -verbose -output localhost.html http://localhost/]**. On obtient les résultats suivants :

```
1.  λ curl --verbose --output localhost.html http://localhost/
2.    % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
3.                                   Dload  Upload   Total   Spent    Left  Speed
4.    0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0*   Trying ::1...
5.  * TCP_NODELAY set
6.  *   Trying 127.0.0.1...
7.  * TCP_NODELAY set
8.    0     0    0     0    0     0      0      0 --:--:--  0:00:01 --:--:--     0* Connected to localhost
    (::1) port 80 (#0)
9.    0     0    0     0    0     0      0      0 --:--:--  0:00:01 --:--:--     0> GET / HTTP/1.1
10. > Host: localhost
11. > User-Agent: curl/7.63.0
12. > Accept: */*
13. >
14. < HTTP/1.1 200 OK
15. < Date: Sun, 05 Jul 2020 17:35:43 GMT
16. < Server: Apache/2.4.35 (Win64) OpenSSL/1.1.1b PHP/7.2.19
17. < X-Powered-By: PHP/7.2.19
18. < Content-Length: 1776
19. < Content-Type: text/html; charset=UTF-8
20. <
21. { [1776 bytes data]
22. 100  1776  100  1776    0     0   1062      0  0:00:01  0:00:01 --:--:--  1062
23. * Connection #0 to host localhost left intact
```

- lignes 10-13 : lignes envoyées par **[curl]** au serveur **[localhost]**. On reconnaît le protocole HTTP ;
- lignes 14-20 : lignes envoyées en réponse par le serveur ;
- ligne 14 : indique qu'on a bien eu le document demandé ;

Le fichier **[localhost.html]** contient le document demandé. Vous pouvez le vérifier en chargeant le fichier dans un éditeur de texte.

Maintenant demandons l'URL **[https://tahe.developpez.com:443/]**. Pour avoir cette URL, le client HTTP doit savoir parler HTTPS. C'est le cas du client **[curl]**.

Les résultats console sont les suivants :

```
1.  C:\Temp\curl
2.  λ curl --verbose --output tahe.developpez.com.html https://tahe.developpez.com:443/
3.    % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
4.                                   Dload  Upload   Total   Spent    Left  Speed
5.    0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0*   Trying 87.98.130.52...
6.  * TCP_NODELAY set
7.    0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0* Connected to
    tahe.developpez.com (87.98.130.52) port 443 (#0)
8.  * ALPN, offering h2
9.  * ALPN, offering http/1.1
10. * successfully set certificate verify locations:
11. *   CAfile: C:\MyPrograms\laragon\bin\laragon\utils\curl-ca-bundle.crt
12.   CApath: none
13. } [5 bytes data]
14. * TLSv1.3 (OUT), TLS handshake, Client hello (1):
15. } [512 bytes data]
16. * TLSv1.3 (IN), TLS handshake, Server hello (2):
17. { [122 bytes data]
18. * TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
19. { [25 bytes data]
20. * TLSv1.3 (IN), TLS handshake, Certificate (11):
21. { [2563 bytes data]
22. * TLSv1.3 (IN), TLS handshake, CERT verify (15):
23. { [264 bytes data]
24. * TLSv1.3 (IN), TLS handshake, Finished (20):
25. { [52 bytes data]
26. * TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
27. } [1 bytes data]
28. * TLSv1.3 (OUT), TLS handshake, Finished (20):
29. } [52 bytes data]
30. * SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
31. * ALPN, server accepted to use http/1.1
32. * Server certificate:
33. *  subject: CN=*.developpez.com
34. *  start date: Jul  1 15:38:30 2020 GMT
35. *  expire date: Sep 29 15:38:30 2020 GMT
36. *  subjectAltName: host "tahe.developpez.com" matched cert's "*.developpez.com"
37. *  issuer: C=US; O=Let's Encrypt; CN=Let's Encrypt Authority X3
38. *  SSL certificate verify ok.
39. } [5 bytes data]
40. > GET / HTTP/1.1
41. > Host: tahe.developpez.com
42. > User-Agent: curl/7.63.0
43. > Accept: */*
44. >
45. { [5 bytes data]
46. * TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
47. { [281 bytes data]
48. * TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
49. { [297 bytes data]
50. * old SSL session ID is stale, removing
51. { [5 bytes data]
52. < HTTP/1.1 200 OK
53. < Date: Sun, 05 Jul 2020 17:39:53 GMT
54. < Server: Apache/2.4.38 (Debian)
55. < X-Powered-By: PHP/5.3.29
56. < Vary: Accept-Encoding
57. < Transfer-Encoding: chunked
58. < Content-Type: text/html
59. <
60. { [6 bytes data]
61. 100   99k    0   99k    0     0  79343      0 --:--:--  0:00:01 --:--:-- 79343
62. * Connection #0 to host tahe.developpez.com left intact
```

- lignes 10-39 : les échanges client / serveur pour sécuriser la connexion : celle-ci sera chiffrée ;
- lignes 41-44 : les entêtes HTTP envoyés par le client **[curl]** au serveur ;
- ligne 52 : le document demandé a bien été trouvé ;
- ligne 57 : le document est envoyé par morceaux ;

[curl] gère correctement à la fois le protocole sécurisé HTTPS et le fait que le document soit envoyé par morceaux. Le document envoyé sera trouvé ici dans le fichier **[tahe.developpez.com.html]**.

Demandons maintenant l'URL **[http://sergetahe.com/cours-tutoriels-de-programmation]**. Nous avions vu que pour cette URL, il y avait une redirection vers l'URL **[http://sergetahe.com/cours-tutoriels-de-programmation/]** (avec un / à la fin).

Les résultats console sont alors les suivants :

```
1.  C:\Temp\curl
2.  λ curl --verbose --output sergetahe.com.html --location http://sergetahe.com/cours-tutoriels-de-
    programmation
3.    % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
4.                                   Dload  Upload   Total   Spent    Left  Speed
5.    0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0*   Trying 87.98.154.146...
6.  * TCP_NODELAY set
7.  * Connected to sergetahe.com (87.98.154.146) port 80 (#0)
8.  > GET /cours-tutoriels-de-programmation HTTP/1.1
9.  > Host: sergetahe.com
10. > User-Agent: curl/7.63.0
11. > Accept: */*
12. >
13. < HTTP/1.1 301 Moved Permanently
14. < Date: Sun, 05 Jul 2020 17:44:17 GMT
15. < Content-Type: text/html; charset=iso-8859-1
16. < Content-Length: 262
17. < Server: Apache
18. < Location: http://sergetahe.com/cours-tutoriels-de-programmation/
19. < Set-Cookie: SERVERID68971=2620178|XwIRd|XwIRd; path=/
20. < X-IPLB-Instance: 17095
21. <
22. * Ignoring the response-body
23. { [262 bytes data]
24. 100   262  100   262    0     0   1858      0 --:--:-- --:--:-- --:--:--  1858
25. * Connection #0 to host sergetahe.com left intact
26. * Issue another request to this URL: 'http://sergetahe.com/cours-tutoriels-de-programmation/'
27. * Found bundle for host sergetahe.com: 0x14385f8 [can pipeline]
28. * Could pipeline, but not asked to!
29. * Re-using existing connection! (#0) with host sergetahe.com
30. * Connected to sergetahe.com (87.98.154.146) port 80 (#0)
31. > GET /cours-tutoriels-de-programmation/ HTTP/1.1
32. > Host: sergetahe.com
33. > User-Agent: curl/7.63.0
34. > Accept: */*
35. >
36. < HTTP/1.1 301 Moved Permanently
37. < Date: Sun, 05 Jul 2020 17:44:17 GMT
38. < Content-Type: text/html; charset=iso-8859-1
39. < Content-Length: 263
40. < Server: Apache
41. < Location: https://sergetahe.com/cours-tutoriels-de-programmation/
42. < Set-Cookie: SERVERID68971=2620178|XwIRd|XwIRd; path=/
43. < X-IPLB-Instance: 17095
44. <
45. * Ignoring the response-body
46. { [263 bytes data]
47. 100   263  100   263    0     0    764      0 --:--:-- --:--:-- --:--:--   764
48. * Connection #0 to host sergetahe.com left intact
49. * Issue another request to this URL: 'https://sergetahe.com/cours-tutoriels-de-programmation/'
50. *   Trying 87.98.154.146...
51. * TCP_NODELAY set
52. * Connected to sergetahe.com (87.98.154.146) port 443 (#1)
53. * ALPN, offering h2
54. * ALPN, offering http/1.1
55. * successfully set certificate verify locations:
56. *   CAfile: C:\MyPrograms\laragon\bin\laragon\utils\curl-ca-bundle.crt
57.   CApath: none
58. } [5 bytes data]
59. * TLSv1.3 (OUT), TLS handshake, Client hello (1):
60. } [512 bytes data]
61. * TLSv1.3 (IN), TLS handshake, Server hello (2):
62. { [102 bytes data]
63. * TLSv1.2 (IN), TLS handshake, Certificate (11):
64. { [2572 bytes data]
65. * TLSv1.2 (IN), TLS handshake, Server key exchange (12):
66. { [333 bytes data]
67. * TLSv1.2 (IN), TLS handshake, Server finished (14):
```

```
68. { [4 bytes data]
69. * TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
70. } [70 bytes data]
71. * TLSv1.2 (OUT), TLS change cipher, Change cipher spec (1):
72. } [1 bytes data]
73. * TLSv1.2 (OUT), TLS handshake, Finished (20):
74. } [16 bytes data]
75.   0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0* TLSv1.2 (IN), TLS
    handshake, Finished (20):
76. { [16 bytes data]
77. * SSL connection using TLSv1.2 / ECDHE-RSA-AES128-GCM-SHA256
78. * ALPN, server accepted to use h2
79. * Server certificate:
80. *  subject: CN=sergetahe.com
81. *  start date: May 10 01:41:15 2020 GMT
82. *  expire date: Aug  8 01:41:15 2020 GMT
83. *  subjectAltName: host "sergetahe.com" matched cert's "sergetahe.com"
84. *  issuer: C=US; O=Let's Encrypt; CN=Let's Encrypt Authority X3
85. *  SSL certificate verify ok.
86. * Using HTTP2, server supports multi-use
87. * Connection state changed (HTTP/2 confirmed)
88. * Copying HTTP/2 data in stream buffer to connection buffer after upgrade: len=0
89. } [5 bytes data]
90. * Using Stream ID: 1 (easy handle 0x2bee870)
91. } [5 bytes data]
92. > GET /cours-tutoriels-de-programmation/ HTTP/2
93. > Host: sergetahe.com
94. > User-Agent: curl/7.63.0
95. > Accept: */*
96. >
97. { [5 bytes data]
98. * Connection state changed (MAX_CONCURRENT_STREAMS == 128)!
99. } [5 bytes data]
100.  0     0    0     0    0     0      0      0 --:--:--  0:00:01 --:--:--     0< HTTP/2 200
101.< date: Sun, 05 Jul 2020 17:44:19 GMT
102.< content-type: text/html; charset=UTF-8
103.< server: Apache
104.< x-powered-by: PHP/7.3
105.< link: <https://sergetahe.com/cours-tutoriels-de-programmation/wp-json/>; rel="https://api.w.org/"
106.< link: <https://sergetahe.com/cours-tutoriels-de-programmation/>; rel=shortlink
107.< vary: Accept-Encoding
108.< x-iplb-instance: 17080
109.< set-cookie: SERVERID68971=2620178|XwIRd|XwIRd; path=/
110.<
111.{ [5 bytes data]
112.100 49634    0 49634    0     0  26040      0 --:--:--  0:00:01 --:--:-- 37830
113.* Connection #1 to host sergetahe.com left intact
```

- ligne 2 : on utilise l'option **[--location]** pour indiquer qu'on veut suivre les redirections envoyées par le serveur ;
- ligne 13 : le serveur indique que le document demandé a changé d'URL ;
- ligne 18 : il indique la nouvelle URL du document demandé ;
- ligne 31 : **[curl]** émet une nouvelle requête vers cette fois la nouvelle URL ;
- ligne 36 : le serveur répond de nouveau que l'URL a changé ;
- ligne 41 : la nouvelle URL est exactement la même que celle qui a été redirigée à un détail près : le procole a changé. Il est devenu HTTPS (ligne 41) alors qu'il était http auparavant (ligne 31) ;
- ligne 49 : une nouvelle requête st émise vers la nouvelle URL. Celle-ci est chiffrée. Aussi tout un dialogue de mise en place de la sécurité se met en place, lignes 53-91 ;
- ligne 92 : la nouvelle URL est demandée avec cette fois le protocole HTTP/2 ;
- ligne 100 : le document a été trouvé ;

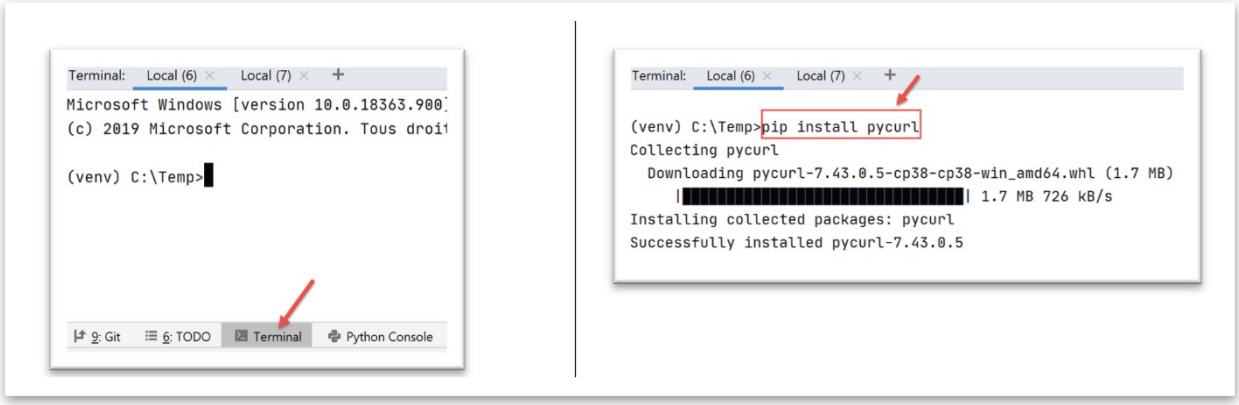
Le document demandé sera trouvé dans le fichier **[sergetahe.com.html]**.

```
1.  C:\Temp\curl
2.  λ dir
3.   Le volume dans le lecteur C s'appelle Local Disk
4.   Le numéro de série du volume est B84C-D958
5.
6.   Répertoire de C:\Temp\curl
7.
8.  05/07/2020  19:44    <DIR>          .
9.  05/07/2020  19:44    <DIR>          ..
10. 05/07/2020  19:35             1 776 localhost.html
11. 05/07/2020  19:44            49 634 sergetahe.com.html
12. 05/07/2020  19:39           101 639 tahe.developpez.com.html
```

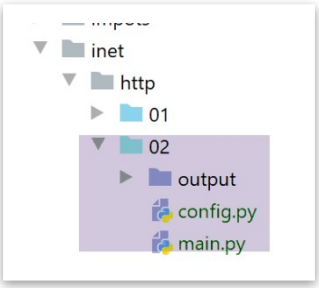
```
13.               3 fichier(s)          153 049 octets
14.               2 Rép(s)  892 385 628 160 octets libres
```

21.4.5 Exemple 5

Python possède un module appelé **[pyccurl]** qui permet d'utiliser les capacités de l'outil **[curl]** dans un programme Python. Nous installons ce module :



Nous allons écrire un nouveau script **[http/02/main.py]** :



Le fichier **[http/02/config]** est le suivant :

```
1.  def configure():
2.      # liste des URL à interroger
3.      urls = [
4.          # site : serveur auquel se connecter
5.          # timeout : délai maximal d'attente d'une réponse du serveur
6.          # target : url à demander
7.          # encoding : encodage de la réponse du serveur
8.          {
9.              "site": "sergetahe.com",
10.             "timeout": 2000,
11.             "target": "http://sergetahe.com",
12.             "encoding": "utf-8"
13.         },
14.         {
15.             "site": "tahe.developpez.com",
16.             "timeout": 500,
17.             "target": "https://tahe.developpez.com",
18.             "encoding": "iso-8859-1"
19.         },
20.         {
21.             "site": "www.polytech-angers.fr",
22.             "timeout": 500,
23.             "target": "http://www.polytech-angers.fr",
24.             "encoding": "utf-8"
25.         },
26.         {
27.             "site": "localhost",
28.             "timeout": 500,
29.             "target": "http://localhost",
30.             "encoding": "utf-8"
31.         }
```

```
    ]
    # on rend la configuration
    return {
        'urls': urls
    }
```

Le fichier contient une liste de dictionnaires où chacun d'eux a la structure suivante :

- *site* : le nom d'un serveur web ;
- *encoding* : le type d'encodage du document attendu ;
- *timeout* : durée maximale d'attente de la réponse du serveur exprimée en **millisecondes**. Au-delà, le client se déconnectera ;
- *url* : URL du document demandé ;

Le code du script **[http/02/main.py]** est le suivant :

```
# imports
import codecs
from io import BytesIO

import pycurl


# ----------------------------------------------------------------------
def get_url(url: dict, suivi=True):
    # lit l'URL url[url] et la stocke dans le fichier output/url['site'].html
    # si [suivi=True] alors il y a un suivi console de l'échange client / serveur
    # url[timeout] est le timeout des appels client;
    # url [encoding] est l'encodage du document demandé

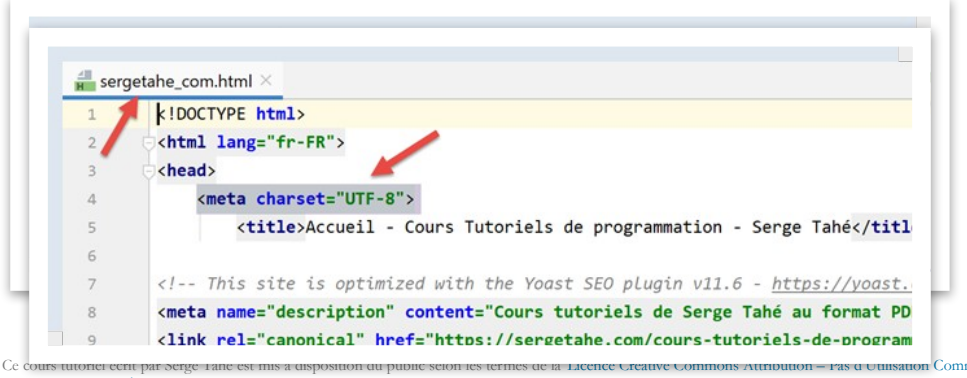
    # on récupère les données de configuration
    server = url['site']
    timeout = url['timeout']
    target = url['target']
    encoding = url['encoding']
    # suivi
    print(f"Client : début de la communication avec le serveur [{server}]")

    # on laisse remonter les exceptions
    html = None
    curl = None
    try:
        # Initialisation d'une session cURL
        curl = pycurl.Curl()
        # flux binaire
        flux = BytesIO()
        # options de curl
        options = {
            # URL
            curl.URL: target,
            # WRITEDATA : là où les données reçues seront stockées
            curl.WRITEDATA: flux,
            # mode verbose
            curl.VERBOSE: suivi,
            # nouvelle connexion - pas de cache
            curl.FRESH_CONNECT: True,
            # timeout de la requête (en secondes)
            curl.TIMEOUT: timeout,
            curl.CONNECTTIMEOUT: timeout,
            # ne pas vérifier la validité des certificats SSL
            curl.SSL_VERIFYPEER: False,
            # suivre les redirections
            curl.FOLLOWLOCATION: True
        }
        # paramétrage de curl
        for option, value in options.items():
            curl.setopt(option, value)
        # Execution de la requête CURL ainsi paramétrée
        curl.perform()
        # création du fichier server.html - on change les caractères gênants pour un nom de fichier
        server2 = server.replace("/", "_")
        server2 = server2.replace(".", "_")
        html_filename = f'{server2}.html'
        html = codecs.open(f"output/{html_filename}", "w", encoding)
        # enregistrement du document reçu dans le fichier HTML
        html.write(flux.getvalue().decode(encoding))
```

```
61.     finally:
62.         # libération des ressources
63.         if curl:
64.             curl.close()
65.         if html:
66.             html.close()
67.
68.
69. # -------------------main
70. # on configure l'application
71. import config
72. config = config.configure()
73.
74. # obtenir les URL du fichier de configuration
75. for url in config['urls']:
76.     print("-------------------------")
77.     print(url['site'])
78.     print("-------------------------")
79.     try:
80.         # lecture URL du site [site]
81.         get_url(url)
82.     # except BaseException as erreur:
83.     #     print(f"L'erreur suivante s'est produite : {erreur}")
84.     finally:
85.         pass
86. # fin
87. print("Terminé...")
```

Commentaires

- ligne 5 : on importe le module **[pycurl]** ;
- ligne 3 : on importe la classe **[BytesIO]** qui va nous permettre de stocker les données reçues du serveur dans un flux binaire ;
- lignes 70-72 : on récupère la configuration de l'application ;
- lignes 75-85 : on boucle sur la liste des URL trouvées dans la configuration ;
- ligne 81 : pour chacune des URL, on appelle la fonction **[get_url]** qui va télécharger l'URL `url['target']` avec un timeout `url['timeout']` ;
- ligne 9 : la fonction **[get_url]** reçoit la configuration de l'URL à interroger ;
- lignes 16-19 : on récupère la configuration de l'URL dans des variables séparées ;
- lignes 26, 61 : on fait toutes les opérations au sein d'un try / finally. On n'arrête pas les exceptions qui remonteront alors au code appelant qui lui les arrête ;
- ligne 28 : on prépare une session **[curl]**. **[pycurl.Curl()]** rend une ressource **[curl]** qui va opérer la transaction avec un serveur ;
- ligne 30 : instanciation du flux binaire qui va stocker les données reçues ;
- lignes 32-48 : le dictionnaire **[options]** va paramétrer la connexion **[curl]** au serveur. leur rôle est indiqué dans les commentaires ;
- lignes 49-51 : les options de la connexion sont transmises à la ressource **[curl]** ;
- ligne 53 : connexion à l'URL demandée avec les options définies. A cause de l'option **[curl.WRITEDATA: flux]** (ligne 36), la fonction **[curl.perform()]** va stocker les données reçues dans **[flux]** ;
- lignes 54-60 : on crée le fichier HTML qui va stocker le document HTML reçu ;
- ligne 60 : le flux binaire **[flux.getvalue()]** va être stocké comme une chaîne de caractères dans le fichier HTML. L'encodage de cette chaîne est précisé dans la méthode **[decode(encoding)]**. Il faut donc connaître l'encodage du document envoyé par le serveur. Si on se trompe, l'opération de décodage du flux binaire va échouer. L'encodage est précisé dans le fichier de configuration de l'URL (ligne 12 par exemple). On aurait pu gérer dynamiquement cette information car le serveur l'envoie dans ces entêtes HTTP. Cela aurait été préférable. Pour garder un code simple, nous ne l'avons pas fait. Pour connaître le type d'encodage du document, il suffit de demander l'URL désirée avec un navigateur et regarder les entêtes HTTP envoyés par celui-ci en mode débogage du navigateur (F12) ou bien le document lui-même car celui-ci précise également l'encodage :



- ligne 61-66 : les ressources allouées sont libérées ;

Lorsqu'on exécute le script **[main.py]** on obtient les résultats console suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/inet/http/02/main.py
2.  -------------------------
3.  sergetahe.com
4.  -------------------------
5.  Client : début de la communication avec le serveur [sergetahe.com]
6.  *   Trying 87.98.154.146:80...
7.  * TCP_NODELAY set
8.  * Connected to sergetahe.com (87.98.154.146) port 80 (#0)
9.  > GET / HTTP/1.1
10. Host: sergetahe.com
11. User-Agent: PycURL/7.43.0.5 libcurl/7.68.0 OpenSSL/1.1.1d zlib/1.2.11 c-ares/1.15.0 WinIDN libssh2/1.9.0
    nghttp2/1.40.0
12. Accept: */*
13.
14. * Mark bundle as not supporting multiuse
15. < HTTP/1.1 302 Found
16. < Date: Mon, 06 Jul 2020 06:45:52 GMT
17. < Content-Type: text/html; charset=UTF-8
18. < Transfer-Encoding: chunked
19. < Server: Apache
20. < X-Powered-By: PHP/7.3
21. < Location: http://sergetahe.com/cours-tutoriels-de-programmation
22. < Set-Cookie: SERVERID68971=26218|XwLIo|XwLIo; path=/
23. < X-IPLB-Instance: 17102
24. <
25. * Ignoring the response-body
26. * Connection #0 to host sergetahe.com left intact
27. * Issue another request to this URL: 'http://sergetahe.com/cours-tutoriels-de-programmation'
28. * Found bundle for host sergetahe.com: 0x25eacafb5d0 [serially]
29. * Can not multiplex, even if we wanted to!
30. * Re-using existing connection! (#0) with host sergetahe.com
31. * Connected to sergetahe.com (87.98.154.146) port 80 (#0)
32. > GET /cours-tutoriels-de-programmation HTTP/1.1
33. Host: sergetahe.com
34. User-Agent: PycURL/7.43.0.5 libcurl/7.68.0 OpenSSL/1.1.1d zlib/1.2.11 c-ares/1.15.0 WinIDN libssh2/1.9.0
    nghttp2/1.40.0
35. Accept: */*
36.
37. * Mark bundle as not supporting multiuse
38. < HTTP/1.1 301 Moved Permanently
39. < Date: Mon, 06 Jul 2020 06:45:52 GMT
40. < Content-Type: text/html; charset=iso-8859-1
41. < Content-Length: 262
42. < Server: Apache
43. < Location: http://sergetahe.com/cours-tutoriels-de-programmation/
44. < Set-Cookie: SERVERID68971=26218|XwLIo|XwLIo; path=/
45. < X-IPLB-Instance: 17102
46. <
47. * Ignoring the response-body
48. * Connection #0 to host sergetahe.com left intact
49. * Issue another request to this URL: 'http://sergetahe.com/cours-tutoriels-de-programmation/'
50. * Found bundle for host sergetahe.com: 0x25eacafb5d0 [serially]
51. * Can not multiplex, even if we wanted to!
52. * Re-using existing connection! (#0) with host sergetahe.com
53. * Connected to sergetahe.com (87.98.154.146) port 80 (#0)
54. > GET /cours-tutoriels-de-programmation/ HTTP/1.1
55. Host: sergetahe.com
56. User-Agent: PycURL/7.43.0.5 libcurl/7.68.0 OpenSSL/1.1.1d zlib/1.2.11 c-ares/1.15.0 WinIDN libssh2/1.9.0
    nghttp2/1.40.0
57. Accept: */*
58.
59. * Mark bundle as not supporting multiuse
60. < HTTP/1.1 301 Moved Permanently
61. < Date: Mon, 06 Jul 2020 06:45:52 GMT
62. < Content-Type: text/html; charset=iso-8859-1
63. < Content-Length: 263
64. < Server: Apache
65. < Location: https://sergetahe.com/cours-tutoriels-de-programmation/
66. < Set-Cookie: SERVERID68971=26218|XwLIo|XwLIo; path=/
67. < X-IPLB-Instance: 17102
68. <
69. * Ignoring the response-body
```

```
70. * Connection #0 to host sergetahe.com left intact
71. * Issue another request to this URL: 'https://sergetahe.com/cours-tutoriels-de-programmation/'
72. *   Trying 87.98.154.146:443...
73. * TCP_NODELAY set
74. * …
75. * Using Stream ID: 1 (easy handle 0x25eaec77010)
76. > GET /cours-tutoriels-de-programmation/ HTTP/2
77. Host: sergetahe.com
78. user-agent: PycURL/7.43.0.5 libcurl/7.68.0 OpenSSL/1.1.1d zlib/1.2.11 c-ares/1.15.0 WinIDN libssh2/1.9.0
    nghttp2/1.40.0
79. accept: */*
80.
81. * Connection state changed (MAX_CONCURRENT_STREAMS == 128)!
82. < HTTP/2 200
83. < date: Mon, 06 Jul 2020 06:45:53 GMT
84. < content-type: text/html; charset=UTF-8
85. < server: Apache
86. < x-powered-by: PHP/7.3
87. < link: <https://sergetahe.com/cours-tutoriels-de-programmation/wp-json/>; rel="https://api.w.org/"
88. < link: <https://sergetahe.com/cours-tutoriels-de-programmation/>; rel=shortlink
89. < vary: Accept-Encoding
90. < x-iplb-instance: 17080
91. < set-cookie: SERVERID68971=26218|XwLIp|XwLIp; path=/
92. <
93. * Connection #1 to host sergetahe.com left intact
94. ------------------------
95. tahe.developpez.com
96. ------------------------
97. Client : début de la communication avec le serveur [tahe.developpez.com]
98. *   Trying 87.98.130.52:443...
99. * TCP_NODELAY set
100.* Connected to tahe.developpez.com (87.98.130.52) port 443 (#0)
101.* ALPN, offering h2
102.* ALPN, offering http/1.1
103.* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
104.* ALPN, server accepted to use http/1.1
105.* Server certificate:
106.*  subject: CN=*.developpez.com
107.*  start date: Jul  1 15:38:30 2020 GMT
108.*  expire date: Sep 29 15:38:30 2020 GMT
109.*  subjectAltName: host "tahe.developpez.com" matched cert's "*.developpez.com"
110.*  issuer: C=US; O=Let's Encrypt; CN=Let's Encrypt Authority X3
111.*  SSL certificate verify result: unable to get local issuer certificate (20), continuing anyway.
112.> GET / HTTP/1.1
113.Host: tahe.developpez.com
114.User-Agent: PycURL/7.43.0.5 libcurl/7.68.0 OpenSSL/1.1.1d zlib/1.2.11 c-ares/1.15.0 WinIDN libssh2/1.9.0
    nghttp2/1.40.0
115.Accept: */*
116.
117.* old SSL session ID is stale, removing
118.* Mark bundle as not supporting multiuse
119.< HTTP/1.1 200 OK
120.< Date: Mon, 06 Jul 2020 06:45:53 GMT
121.< Server: Apache/2.4.38 (Debian)
122.< X-Powered-By: PHP/5.3.29
123.< Vary: Accept-Encoding
124.< Transfer-Encoding: chunked
125.< Content-Type: text/html
126.<
127.* Connection #0 to host tahe.developpez.com left intact
128.------------------------
129.www.polytech-angers.fr
130.------------------------
131.Client : début de la communication avec le serveur [www.polytech-angers.fr]
132.*   Trying 193.49.144.41:80...
133.* TCP_NODELAY set
134.* Connected to www.polytech-angers.fr (193.49.144.41) port 80 (#0)
135.> GET / HTTP/1.1
136.Host: www.polytech-angers.fr
137.User-Agent: PycURL/7.43.0.5 libcurl/7.68.0 OpenSSL/1.1.1d zlib/1.2.11 c-ares/1.15.0 WinIDN libssh2/1.9.0
    nghttp2/1.40.0
138.Accept: */*
139.
140.* Mark bundle as not supporting multiuse
141.< HTTP/1.1 301 Moved Permanently
142.< Date: Mon, 06 Jul 2020 06:45:54 GMT
143.< Server: Apache/2.4.29 (Ubuntu)
```

```
144.< Location: http://www.polytech-angers.fr/fr/index.html
145.< Cache-Control: max-age=1
146.< Expires: Mon, 06 Jul 2020 06:45:55 GMT
147.< Content-Length: 339
148.< Content-Type: text/html; charset=iso-8859-1
149.<
150.* Ignoring the response-body
151.* Connection #0 to host www.polytech-angers.fr left intact
152.* Issue another request to this URL: 'http://www.polytech-angers.fr/fr/index.html'
153.* Found bundle for host www.polytech-angers.fr: 0x25eacafb490 [serially]
154.* Can not multiplex, even if we wanted to!
155.* Re-using existing connection! (#0) with host www.polytech-angers.fr
156.* Connected to www.polytech-angers.fr (193.49.144.41) port 80 (#0)
157.> GET /fr/index.html HTTP/1.1
158.Host: www.polytech-angers.fr
159.User-Agent: PycURL/7.43.0.5 libcurl/7.68.0 OpenSSL/1.1.1d zlib/1.2.11 c-ares/1.15.0 WinIDN libssh2/1.9.0
    nghttp2/1.40.0
160.Accept: */*
161.
162.* Mark bundle as not supporting multiuse
163.< HTTP/1.1 200 OK
164.< Date: Mon, 06 Jul 2020 06:45:54 GMT
165.< Server: Apache/2.4.29 (Ubuntu)
166.< Last-Modified: Mon, 06 Jul 2020 04:50:09 GMT
167.< ETag: "85be-5a9be9bfcf228"
168.< Accept-Ranges: bytes
169.< Content-Length: 34238
170.< Cache-Control: max-age=1
171.< Expires: Mon, 06 Jul 2020 06:45:55 GMT
172.< Vary: Accept-Encoding
173.< Content-Type: text/html; charset=UTF-8
174.< Content-Language: fr
175.<
176.* Connection #0 to host www.polytech-angers.fr left intact
177.-------------------------
178.localhost
179.-------------------------
180.Client : début de la communication avec le serveur [localhost]
181.*   Trying ::1:80...
182.* TCP_NODELAY set
183.* Connected to localhost (::1) port 80 (#0)
184.> GET / HTTP/1.1
185.Host: localhost
186.User-Agent: PycURL/7.43.0.5 libcurl/7.68.0 OpenSSL/1.1.1d zlib/1.2.11 c-ares/1.15.0 WinIDN libssh2/1.9.0
    nghttp2/1.40.0
187.Accept: */*
188.
189.* Mark bundle as not supporting multiuse
190.< HTTP/1.1 200 OK
191.< Date: Mon, 06 Jul 2020 06:45:54 GMT
192.< Server: Apache/2.4.35 (Win64) OpenSSL/1.1.1b PHP/7.2.19
193.< X-Powered-By: PHP/7.2.19
194.< Content-Length: 1776
195.< Content-Type: text/html; charset=UTF-8
196.<
197.* Connection #0 to host localhost left intact
198.Terminé...
199.
200.Process finished with exit code 0
```

Commentaires

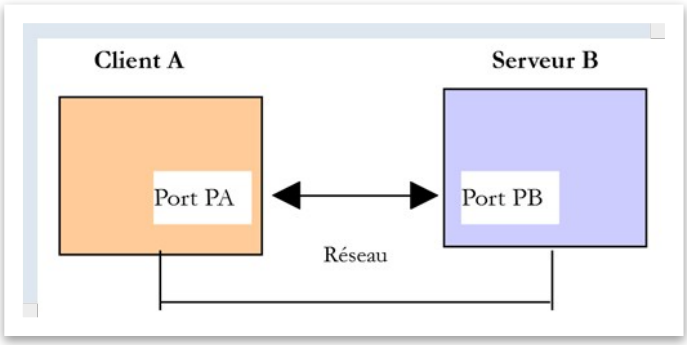
- en bleu, les commandes http envoyées au serveur ;
- en vert, les données reçues en réponse par le client ;
- on obtient les mêmes échanges qu'avec l'outil **[curl]** ;
  - ligne 9 : l'URL **[http://sergetahe.com/]** est demandée ;
  - ligne 15 : le serveur répond que la page a bougé. Ligne 21, la nouvelle URL ;
  - ligne 32 : l'URL **[http://sergetahe.com/cours-tutoriels-de-programmation]** est demandée ;
  - ligne 38 : le serveur répond que la page a bougé. Ligne 43, la nouvelle URL ;
  - ligne 54 : l'URL **[http://sergetahe.com/cours-tutoriels-de-programmation/]** est demandée ;
  - ligne 60 : le serveur répond que la page a bougé. Ligne 65, la nouvelle URL. Elle utilise le protocole sécurisé **[HTTPS]** ;
  - lignes 71-75 : le protocole sécurisé est mis en place avec le serveur ;
  - ligne 76 : l'URL **[https://sergetahe.com/cours-tutoriels-de-programmation/]** est demandée ;
  - ligne 82 : le document demandé a été trouvé ;

21.4.6 Conclusion

Nous avons, dans cette section, découvert le protocole HTTP et avons écrit un script **[http/02/main.py]** capable de télécharger une URL du web.

21.5 Le protocole SMTP (Simple Mail Transfer Protocol)

21.5.1 Introduction

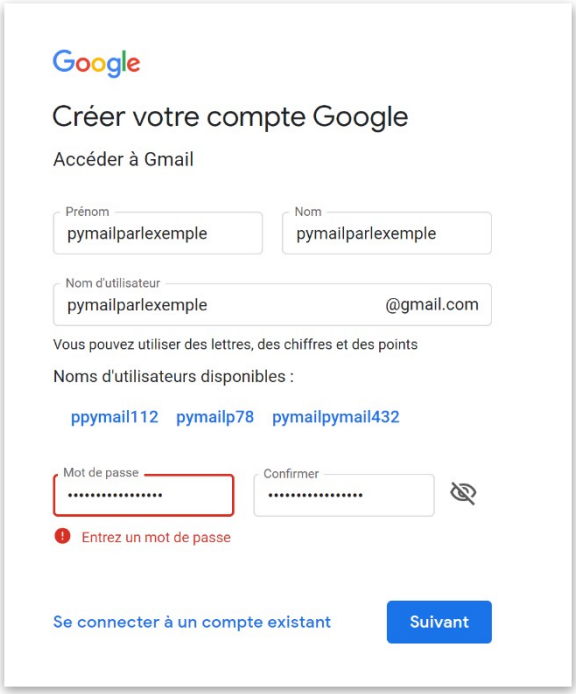


Dans ce chapitre :

- **[Serveur B]** sera un serveur SMTP local que nous installerons ;
- **[Client A]** sera un client SMTP de diverses formes :
  - le client **[RawTcpClient]** pour découvrir le protocole SMTP ;
  - un script Python rejouant le protocole SMTP du client **[RawTcpClient]** ;
  - un script Python utilisant le moduke **[smtplib]** permettant d'envoyer toutes sortes de mails ;

21.5.2 Création d'une adresse [gmail]

Pour faire nos tests SMTP, nous aurons besoin d'une adresse mail à qui écrire. Nous allons créer pour cela une adresse Gmail [https://www.google.com/intl/fr/gmail/about/] :

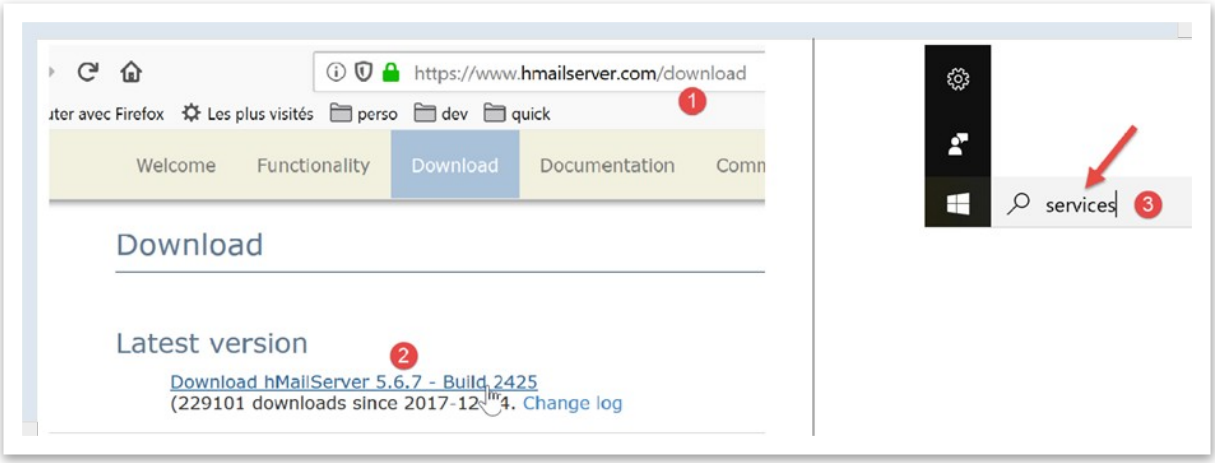


Note : Envoyez quelques mails à l'adresse que vous avez créée. Ne passez à la suite que lorsque vous êtes sûr que le compte créé est capable de recevoir des mails.

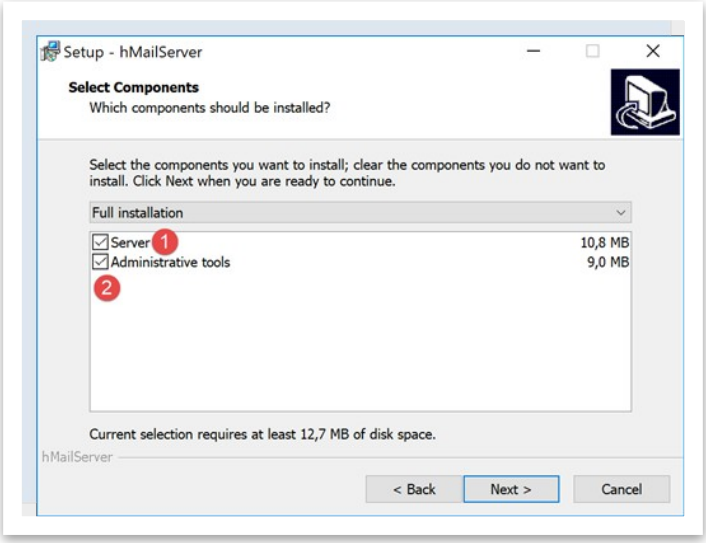
21.5.3 Installation d'un serveur SMTP

Pour nos tests, nous installerons le serveur de mail **[hMailServer]** qui est à la fois un serveur SMTP permettant d'envoyer des mails, un serveur POP3 (Post Office Protocol) permettant de lire les mails stockés sur le serveur, un serveur IMAP (Internet Message Access Protocol) qui lui aussi permet de lire les mails stockés sur le serveur mais va au-delà. Il permet notamment de gérer le stockage des mails sur le serveur.

Le serveur de mail **[hMailServer]** est disponible à l'URL **[https://www.hmailserver.com/]** (mai 2019).



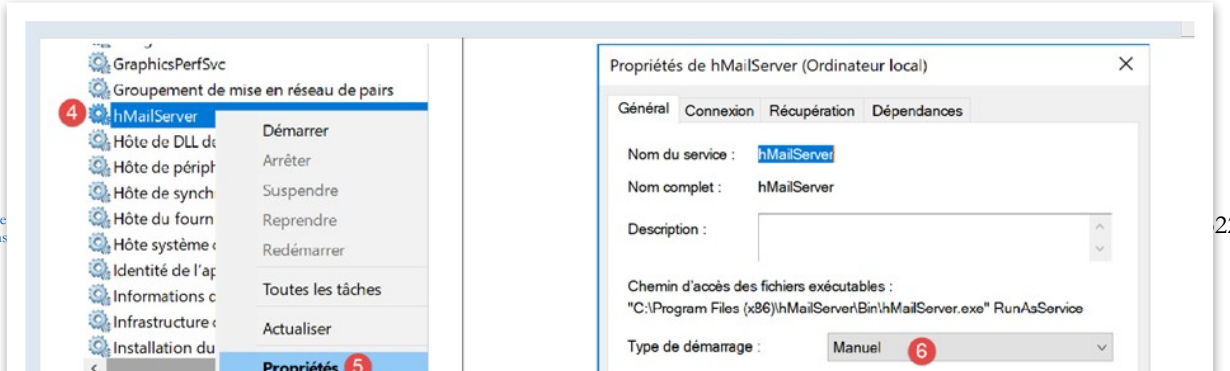
Au cours de l'installation, certains renseignements vous seront demandés :



- en **[1-2]**, sélectionnez à la fois le serveur de mails et les outils pour l'administrer ;
- durant l'installation le mot de l'administrateur vous sera demandé : **notez l**e, car il vous sera nécessaire ;

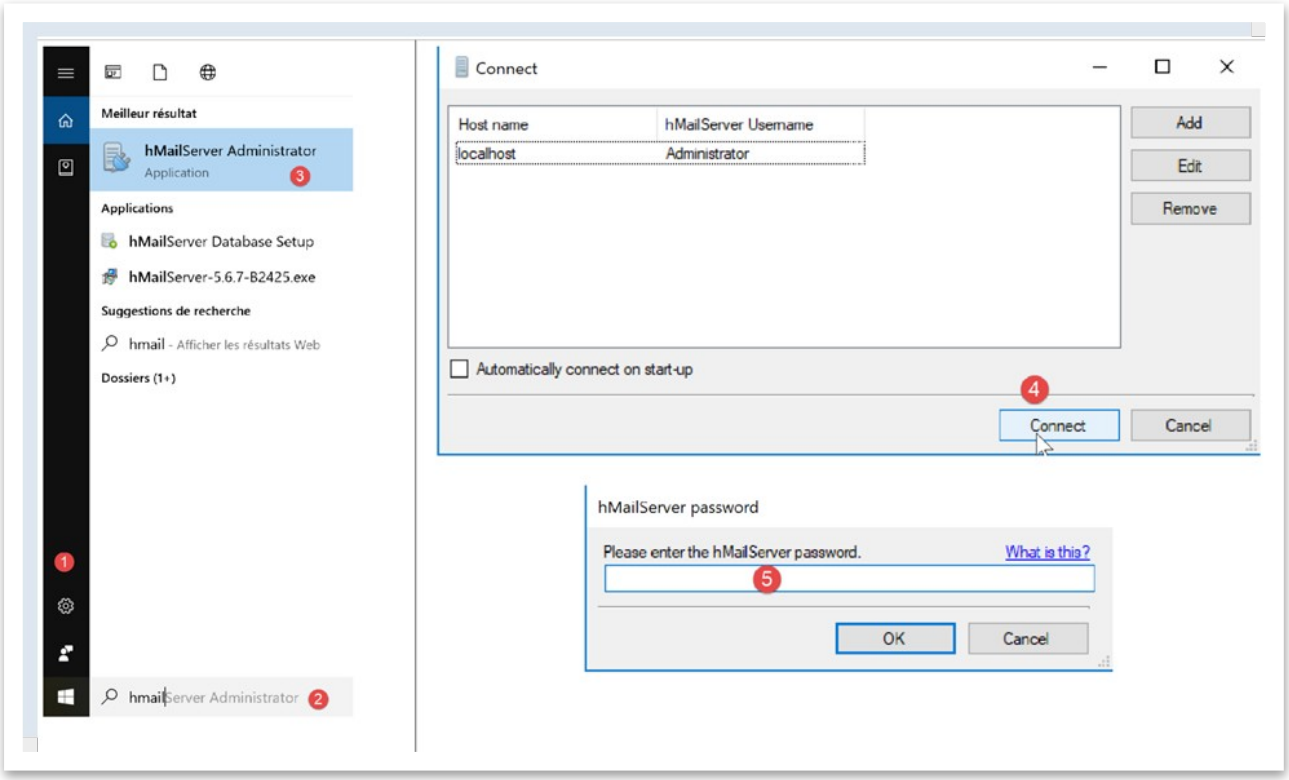
[hMailServer] s'installe comme un service Windows lancé automatiquement au démarrage de la machine. Il est préférable de choisir un démarrage manuel :

- en **[3]**, on tape **[services]** dans la zone de saisie de la barre d'état ;



- en **[4-8]**, on met le service en mode **[manuel]** (6), on le lance (7) ;

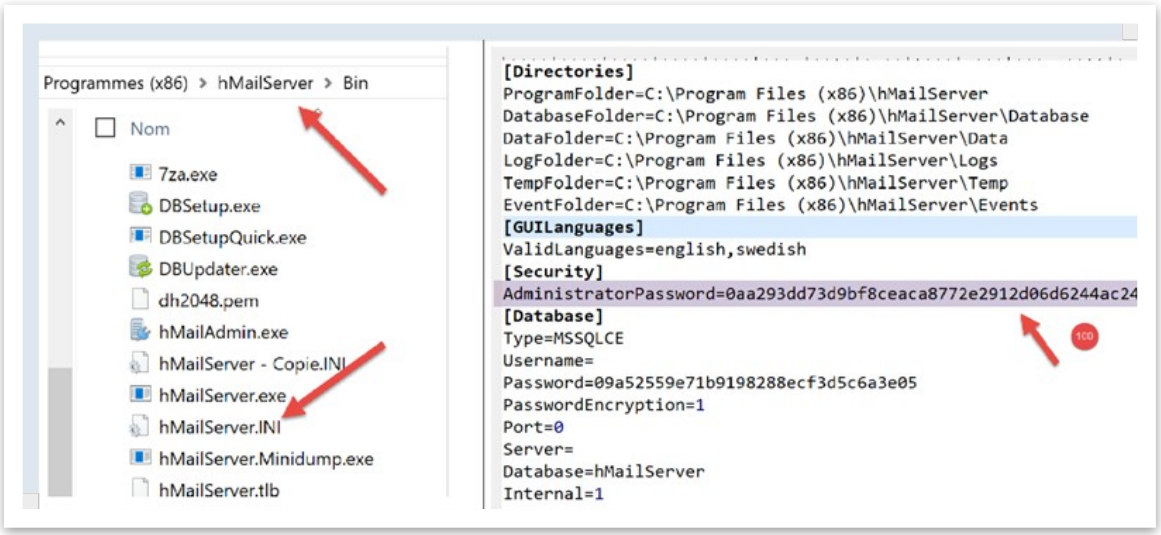
Une fois démarré, le serveur **[hMailServer]** doit être configuré. Le serveur a été installé avec un programme d'administration **[hMailServer Administrator]** :



- en **[2]**, dans la zone de saisie de la barre d'état, taper **[hmailserver]** ;
- en **[3]**, lancer l'administrateur ;
- en **[4]**, connecter l'administrateur au serveur **[hMailServer]** ;
- en **[5]**, taper le mot de passe saisi lors de l'installation de **[hMailServer]** ;

Si vous avez oublié le mot de passe, procédez comme suit :

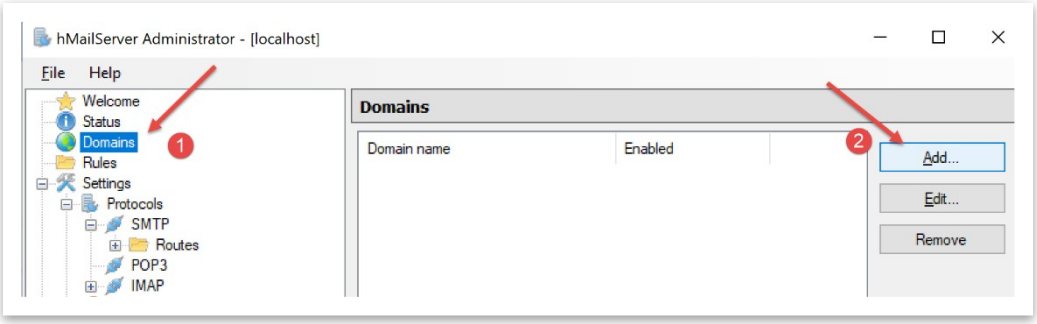
- arrêtez le serveur **[hMailServer]** ;
- ouvrez le fichier **[<hmailserver>/bin/hmailserver.ini]** où <hmailserver> est le dossier d'installation du serveur :



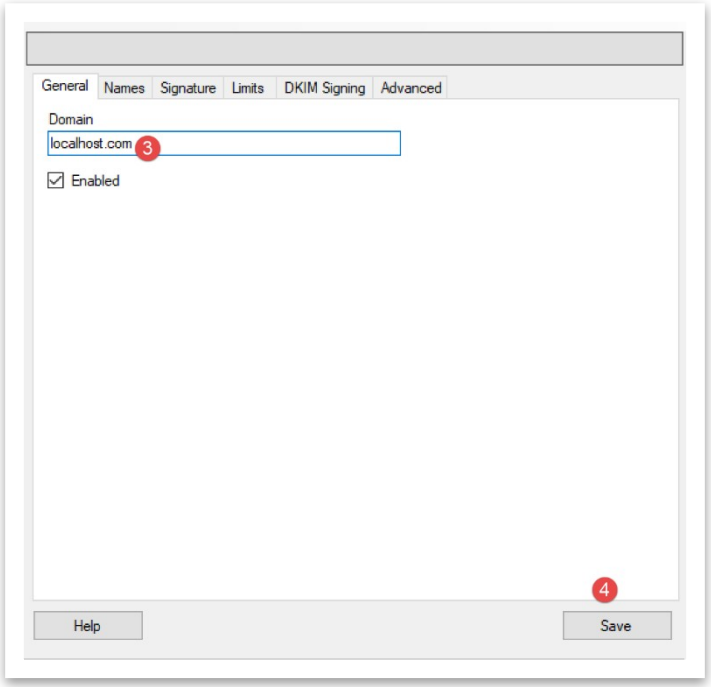
- en **[100]**, enlevez le mot de passe de la ligne **[AdministratorPassword]**. Cela aura pour effet que l'administrateur n'aura plus de mot de passe. Tapez simplement **[Entrée]** lorsque celui-ci vous sera demandé ;

```
1. ValidLanguages=english,swedish
2. [Security]
3. AdministratorPassword=
4. [Database]
```

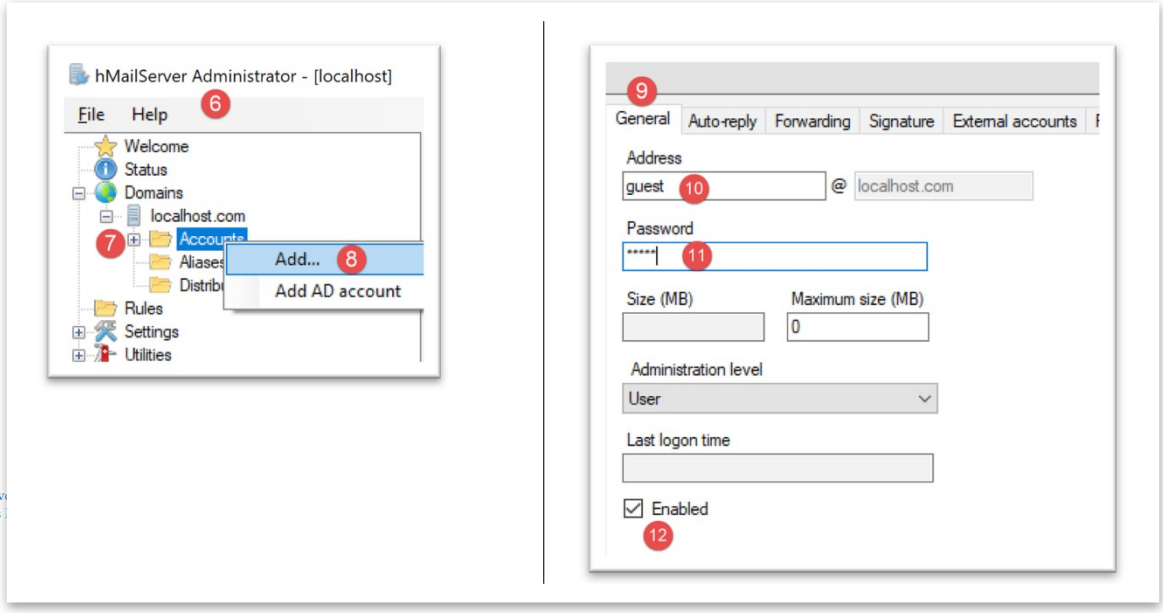
Continuons la configuration du serveur :



- en **[1-2]**, ajoutez un domaine (s'il n'existe pas déjà) ;

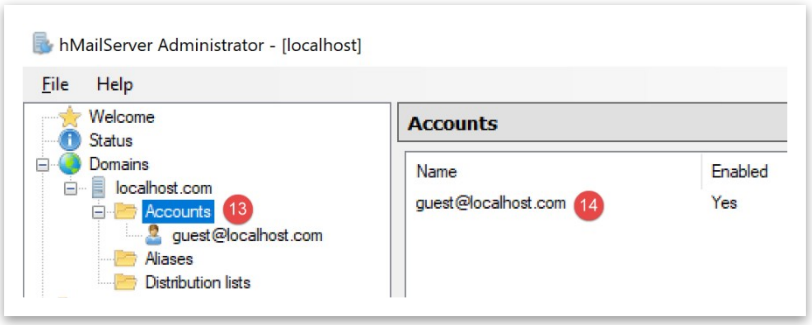


- en **[3]**, on peut mettre à peu près n'importe quoi pour les tests que nous allons opérer. Dans la réalité, il faudrait mettre le nom d'un domaine existant ;

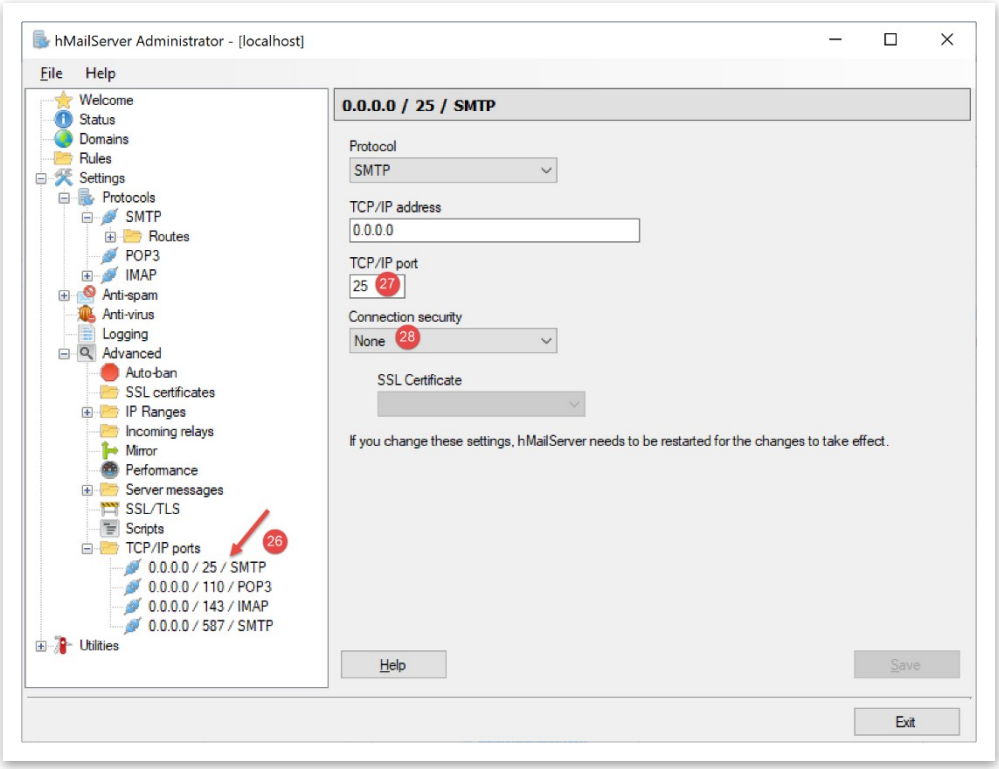


Nous allons créer un compte utilisateur :

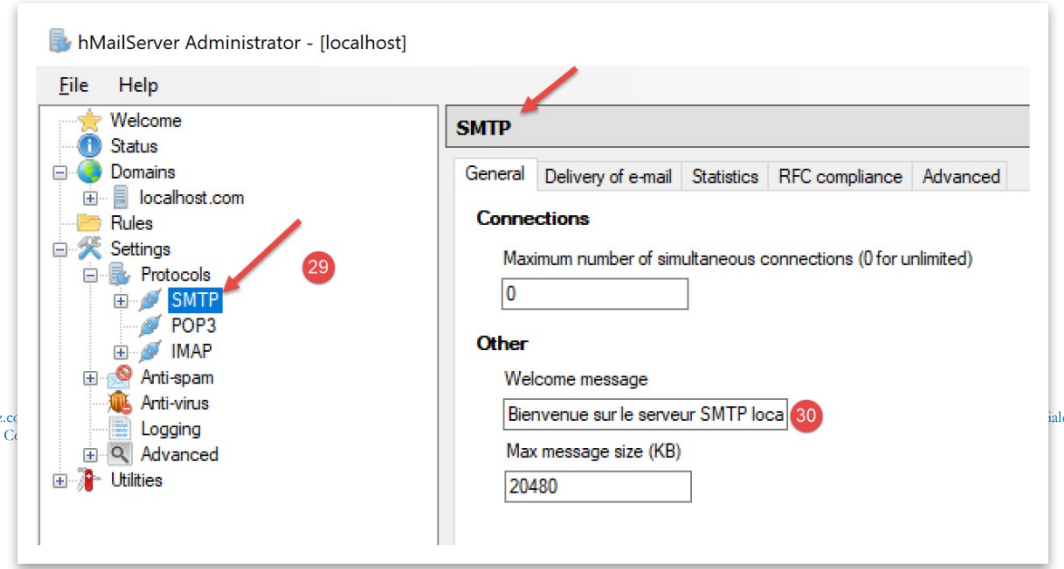
- cliquer droit sur **[Accounts]** (7) puis (8) pour ajouter un nouvel utilisateur ;
- dans l'onglet **[General]** (9), nous définissons un utilisateur **[guest]** (10) avec le mot de passe **[guest]** (11). Il aura l'adresse mail **[guest@localhost]** (10) ;
- en **[12]**, l'utilisateur **[guest]** est activé ;



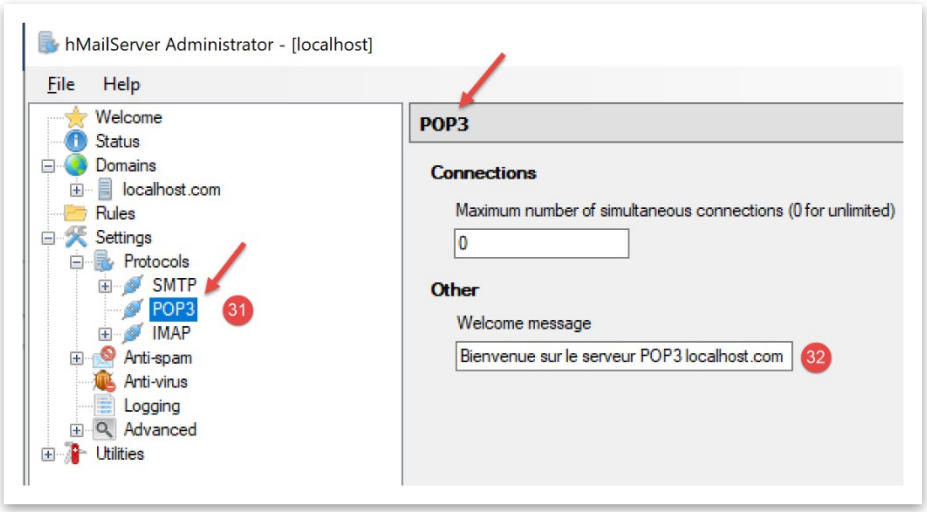
- en **[13-14]**, l'utilisateur créé ;



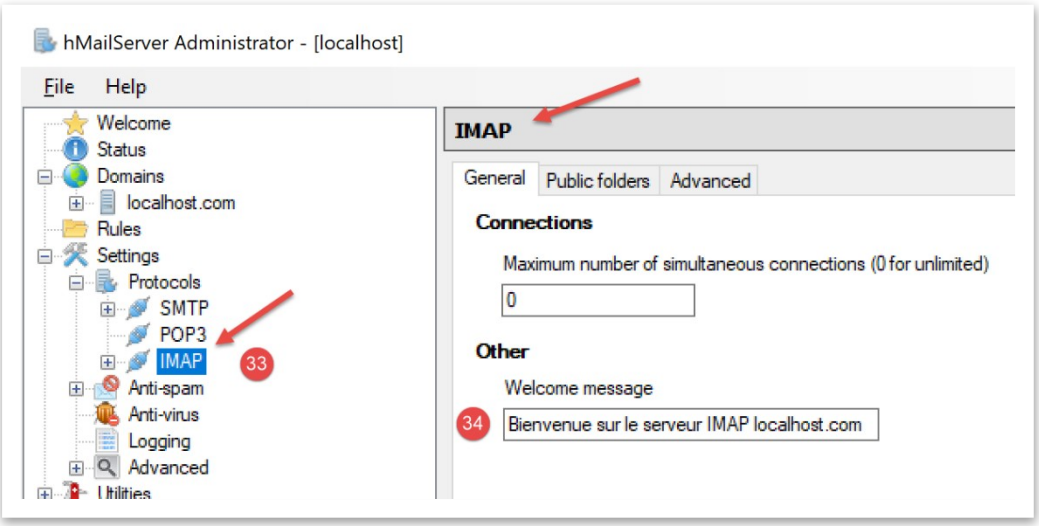
- en **[27]** le port du service SMTP ;
- en **[28]**, ce service ne nécessite pas d'authentification ;
- en **[30]**, mettez le message de bienvenue que le serveur SMTP enverra à ses clients ;



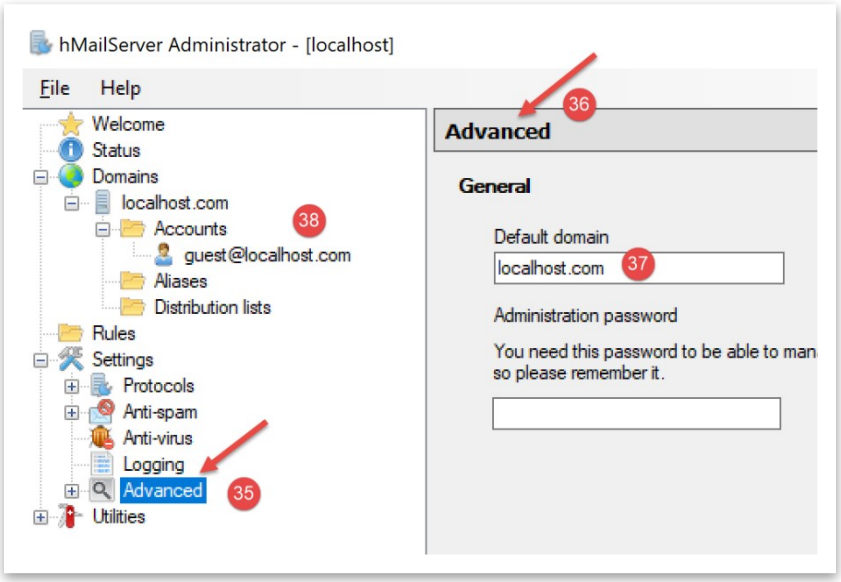
On fait de même avec le serveur POP3 :



On refait la même chose pour le serveur IMAP :

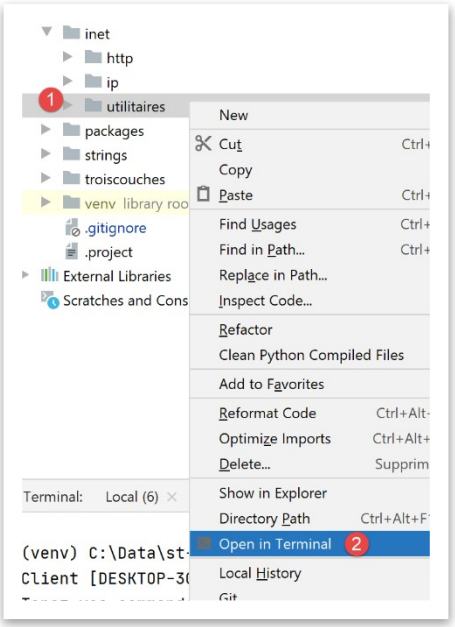


Nous indiquons le domaine par défaut du serveur **`[hMailServer]`** (il peut y en avoir plusieurs) :



- en **[37]**, indiquez que le domaine par défaut du serveur SMTP est celui que vous avez créé en **[38]** ;

Après avoir sauvegardé cette configuration, vous pouvez la tester de la façon suivante. Ouvrez un terminal PyCharm dans le dossier des utilitaires :



Puis tapez la commande suivante :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost 25
2.  Client [DESKTOP-30FF5FB:50170] connecté au serveur [localhost-25]
3.  Tapez vos commandes (quit pour arrêter) :
4.  <-- [220 Bienvenue sur le serveur SMTP localhost.com]
```

- ligne 1 : on se connecte au port 25 de la machine **[localhost]**. C'est là qu'officie un serveur SMTP non sécurisé du serveur **[hMailServer]** ;
- ligne 4 : on reçoit le message de bienvenue que nous avons configuré à l'étape 30 précédente ;

Le serveur SMTP est donc bien en place. Tapez la commande **[quit]** pour terminer le dialogue avec le serveur SMTP 25.

Maintenant faisons la même chose avec le port 587 qui est le port par défaut du service SMTP sécurisé de relève du courrier :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost 587
2.  Client [DESKTOP-30FF5FB:50217] connecté au serveur [localhost-587]
3.  Tapez vos commandes (quit pour arrêter) :
4.  <-- [220 Bienvenue sur le serveur SMTP localhost.com]
```

- ligne 4, la réponse du serveur SMTP officiant sur le port 587 ;

Maintenant faisons la même chose avec le port 110 qui est le port par défaut du service POP3 de relève du courrier :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost 110
2.  Client [DESKTOP-30FF5FB:50210] connecté au serveur [localhost-110]
3.  Tapez vos commandes (quit pour arrêter) :
4.  <-- [+OK Bienvenue sur le serveur POP3 localhost.com]
```

- ligne 4, on a reçu le message de bienvenue du serveur POP3 ;

Maintenant faisons la même chose avec le port 143 qui est le port par défaut du service IMAP de relève du courrier :

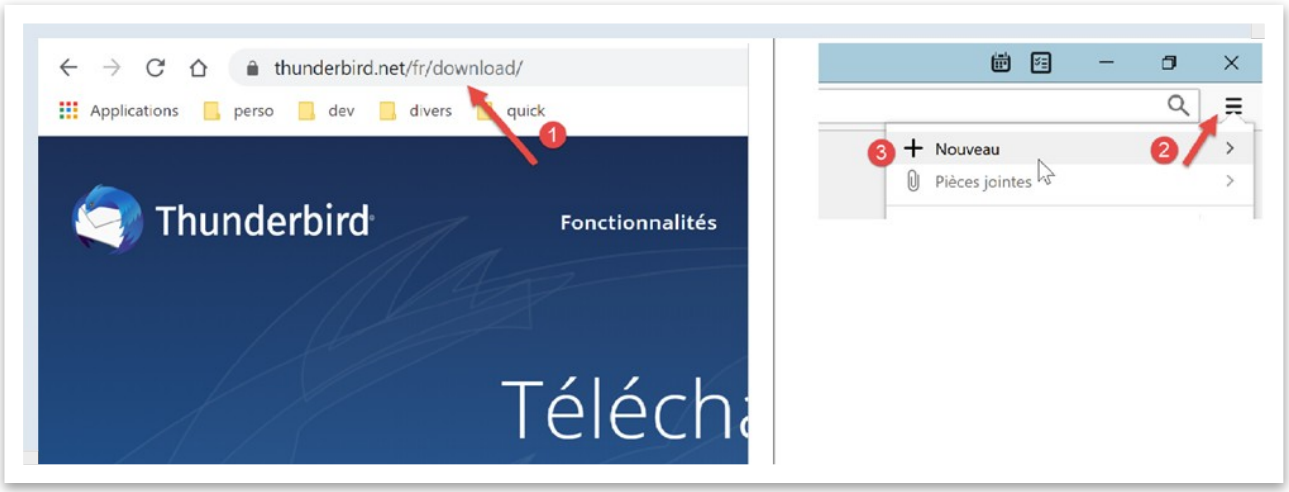
```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost 143
2.  Client [DESKTOP-30FF5FB:50212] connecté au serveur [localhost-143]
3.  Tapez vos commandes (quit pour arrêter) :
4.  <-- [* OK Bienvenue sur le serveur IMAP localhost.com]
```

- ligne 4, on a reçu le message de bienvenue du serveur IMAP ;

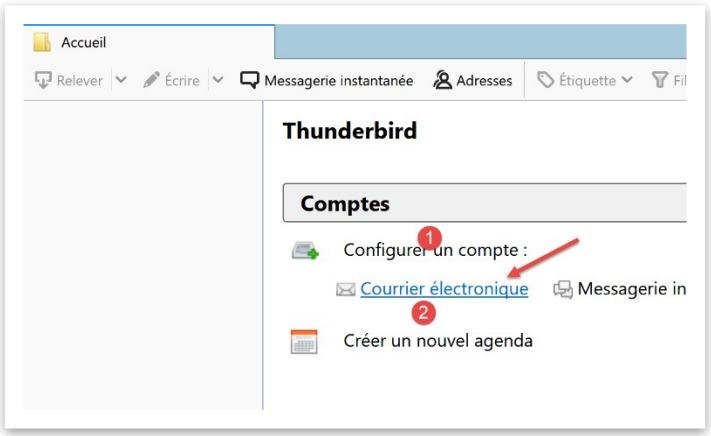
21.5.4 Installation d'un lecteur de courrier

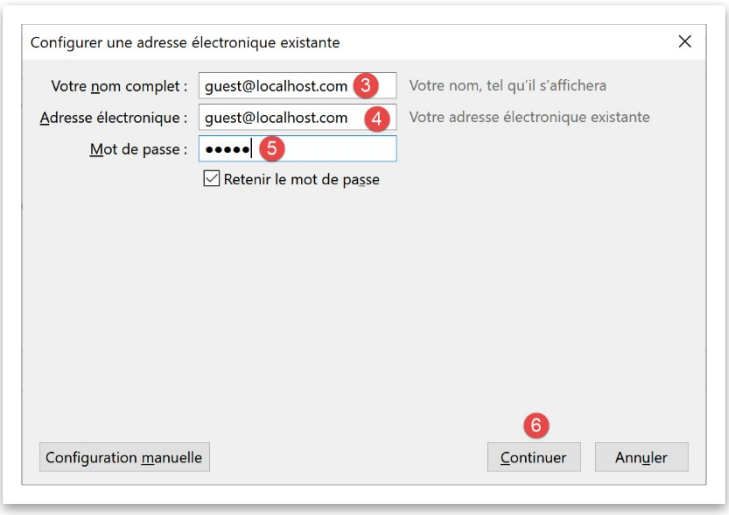
Pour lire le courrier que nous allons envoyer, il nous faut un lecteur de courrier. Pour ceux qui n'en ont pas, nous montrons l'installation et la configuration du lecteur **[Thunderbird]** :

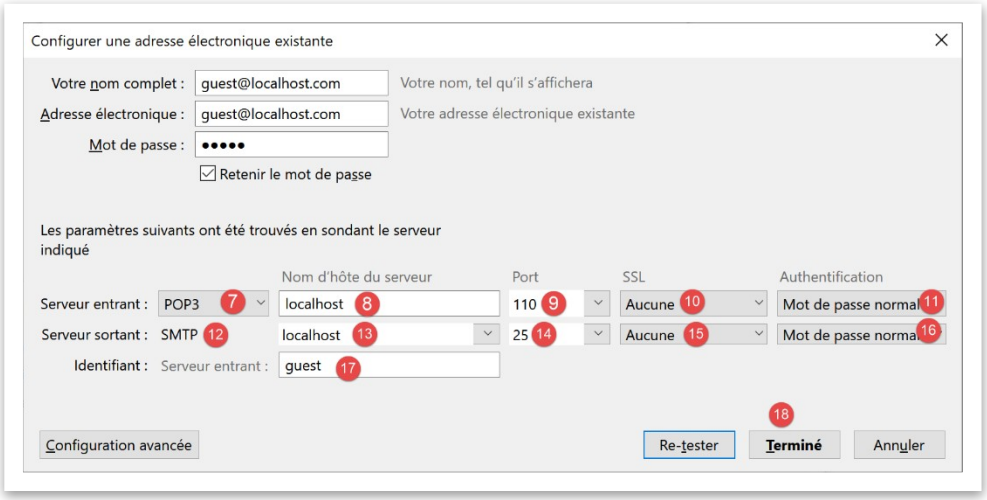
- en **[1]** : téléchargez **[thunderbird]** puis installez-le ;



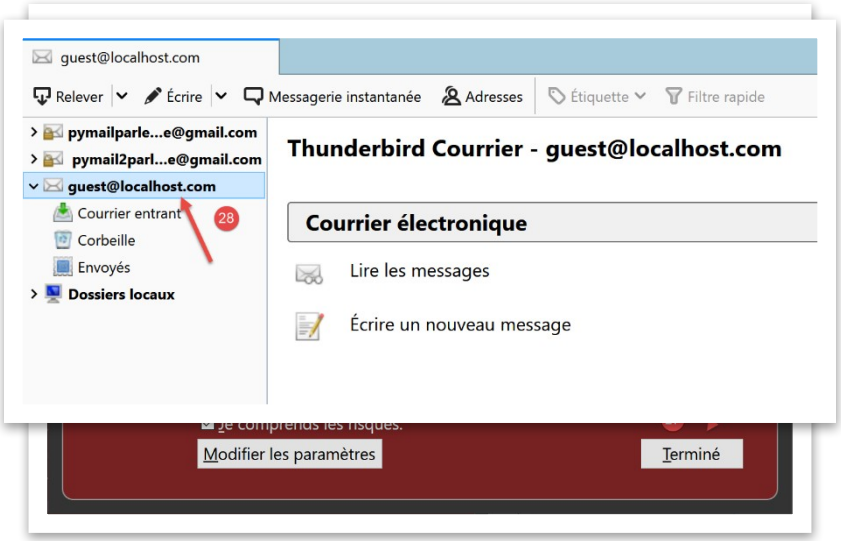
- lancez le serveur de mail **[hMailServer]** s'il ne l'est pas déjà ;
- en **[2-3]** : une fois Thunderbird lancé, nous allons créer un compte de messagerie pour l'utilisateur **[guest@localhost]** du serveur de mail **[hMailServer]** ;







- en **[7-11]** : le serveur POP3 qui va nous permettre de lire le courrier du serveur de mail **[hMailServer]** est à l'adresse **[localhost]** et officie sur le port 110 ;
- en **[12-16]** : le serveur SMTP qui va nous permettre d'envoyer du courrier de la part des utilisateurs du serveur de mail **[hMailServer]** est à l'adresse **[localhost]** et officie sur le port 25 ;
- **[18]** : on peut tester la validation de cette configuration ;



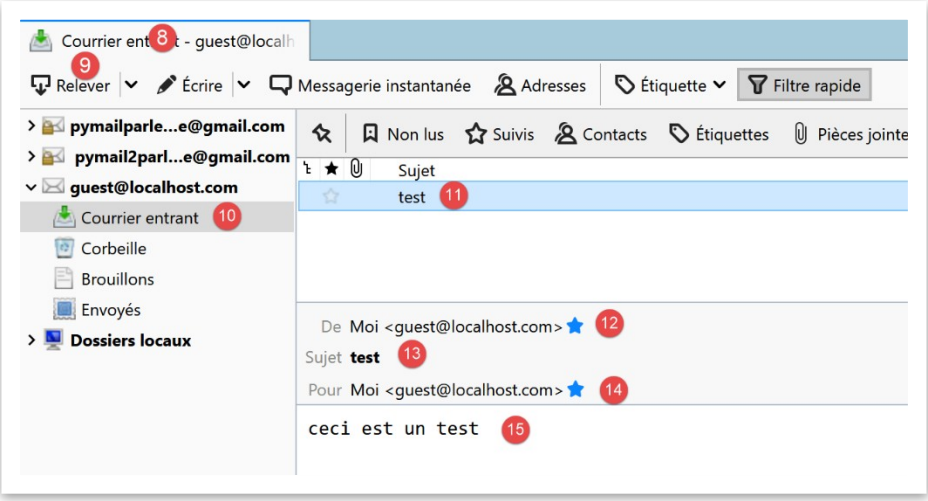
- en **[26]** : parce qu'on n'a pas de chiffrement SSL, Thunderbird nous avertit que notre configuration comporte des risques ;
- en **[28]** : le compte a été créé ;

Pour tester le compte créé, nous allons avec Thunderbird :

- envoyer un mail à l'utilisateur **[guest@localhost.com]** (protocole SMTP) ;
- lire le courrier reçu par cet utilisateur (protocole POP3) ;

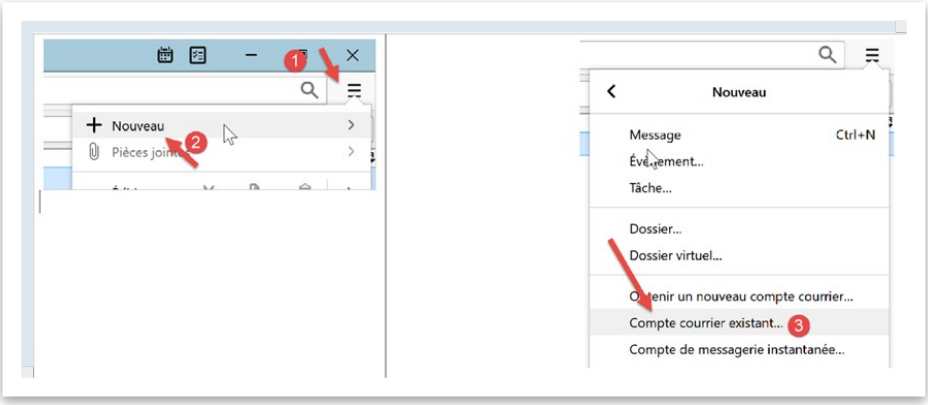


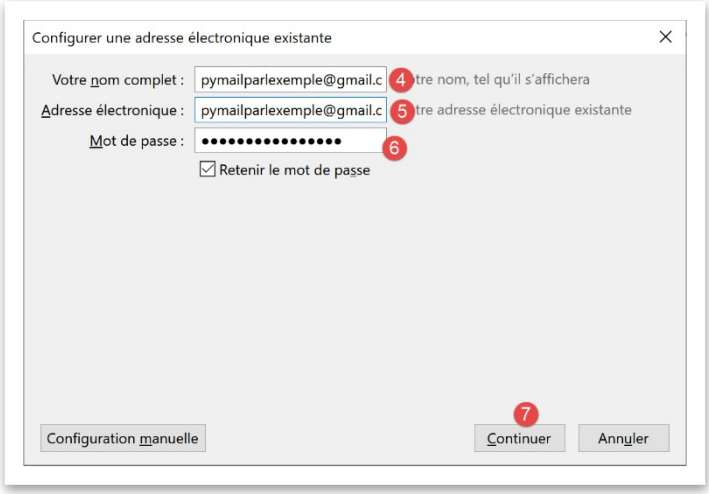
- en **[3]** : l'expéditeur ;
- en **[4]** : le destinataire ;
- en **[5]** : le sujet du mail ;
- en **[6]** : le contenu du mail ;
- en **[7]** : pour envoyer le mail ;



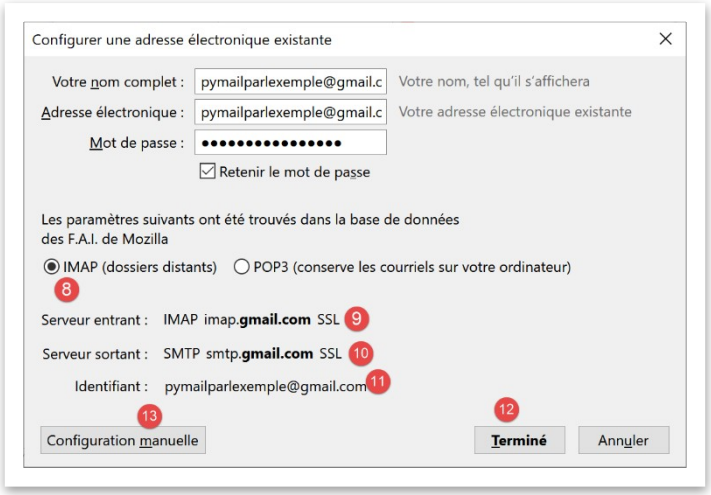
- en **[8-9]** : on relève le courrier de l'utilisateur **[guest@localhost]** ;
- en **[10-15]** : le message reçu ;

Nous allons envoyer également du courrier à l'utilisateur **[pymailparlexemple@gmail.com]**. Créons-lui un compte dans Thunderbird pour lire le courrier qu'il recevra :

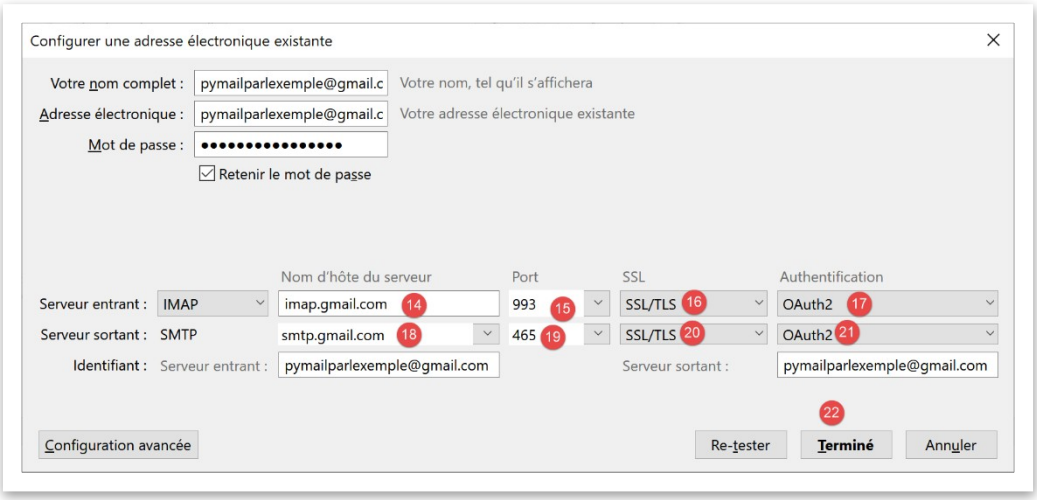




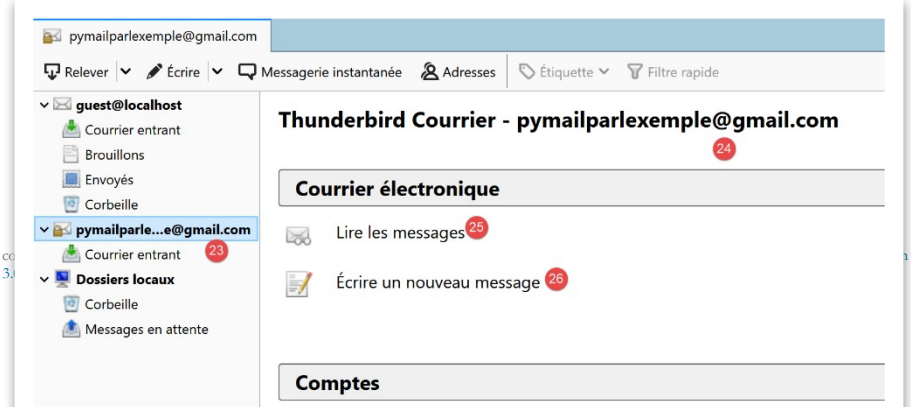
- en **[4]** : mettez ce que vous voulez ;
- en **[5]** : l'adresse est **[pymailparlexemple@gmail.com]** ;
- en **[6]** : tapez le mot de passe que vous avez donné à cet utilisateur lorsque vous l'avez créé ;
- en **[7]** : validez cette configuration ;



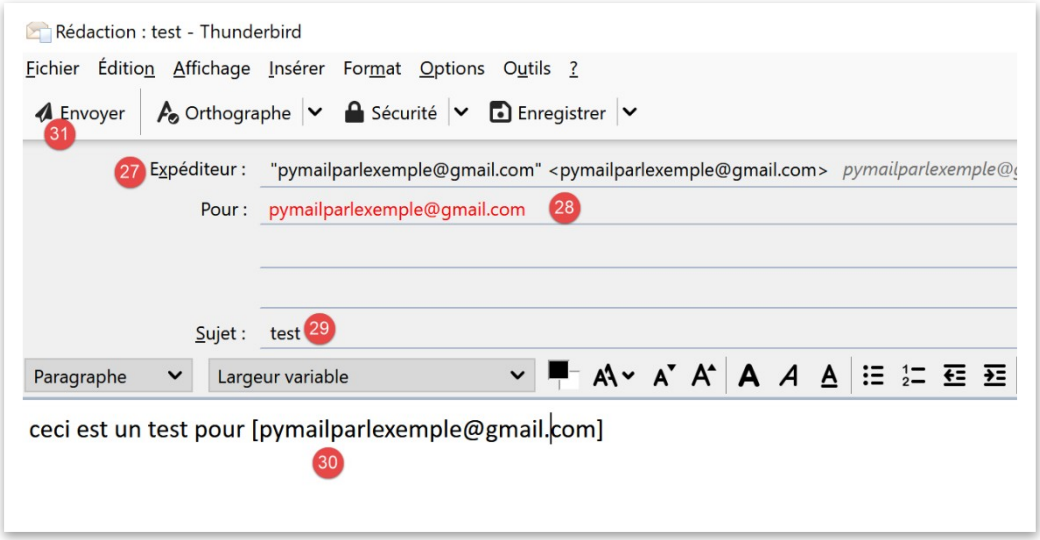
- en **[8]** : Thunderbird a récupéré les informations suivantes dans sa base de données ;
- en **[9]** : le protocole de lecture du courrier n'est plus POP3 mais IMAP. La principale différence entre les deux est que **[POP3]** ramène le courrier lu sur la machine locale où se trouve le lecteur de courrier et le supprime du serveur distant, alors que **[IMAP]** conserve le courrier sur le serveur distant ;
- en **[10]** : identification du serveur SMTP ;
- en **[13]** : pour avoir davantage d'informations sur les serveurs IMAP et SMTP, on passe en configuration manuelle ;



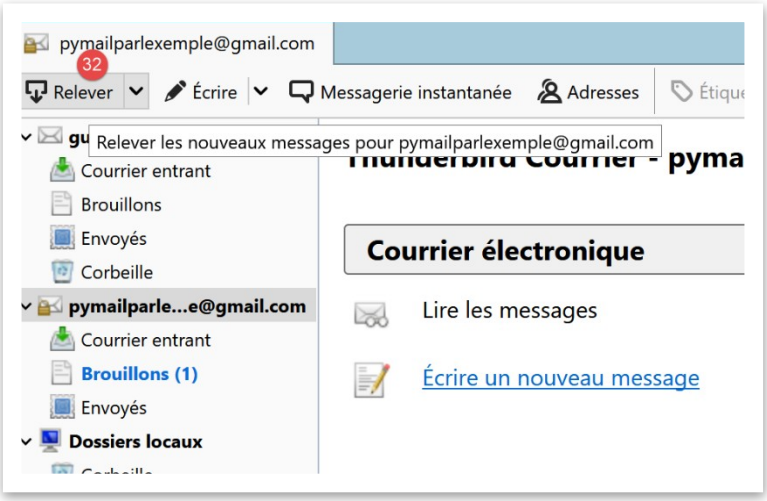
- en **[14-17]** : les caractéristiques du serveur IMAP ;
- en **[18-21]** : les caractéristiques du serveur SMTP ;
- en **[22]** : on termine la configuration ;



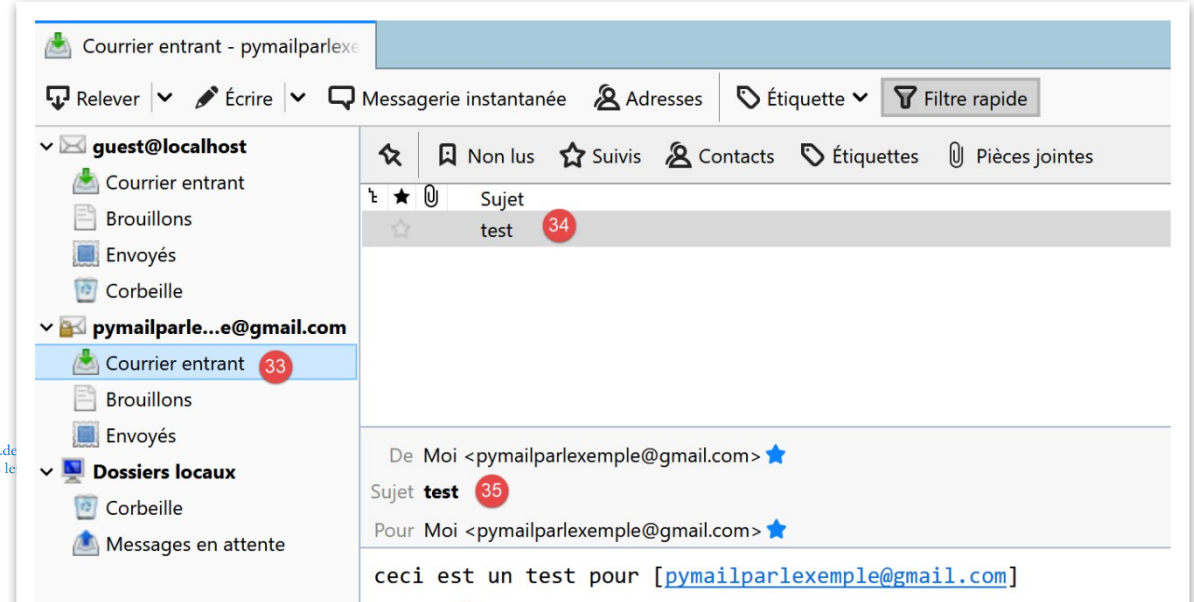
- en **[23-24]** : le nouveau compte Thunderbird ;
- en **[26]** : on écrit un nouveau message ;



- en **[27]** : l'expéditeur est **[pymailparlexemple@gmail.com]** ;
- en **[28]** : le destinataire est **[pymailparlexemple@gmail.com]** ;
- en **[29-30]** : le message ;
- en **[31]** : pour l'envoyer ;



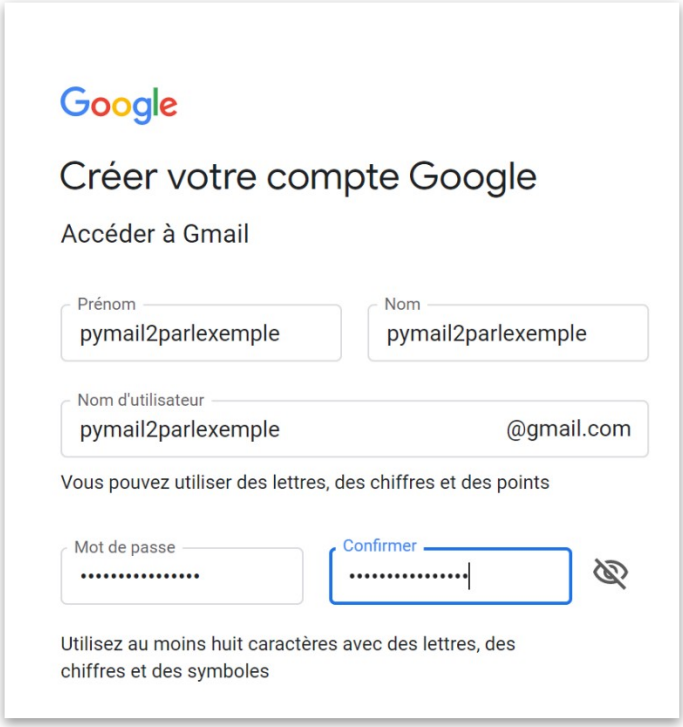
- en **[32]** : on relève le courrier des différents comptes ;

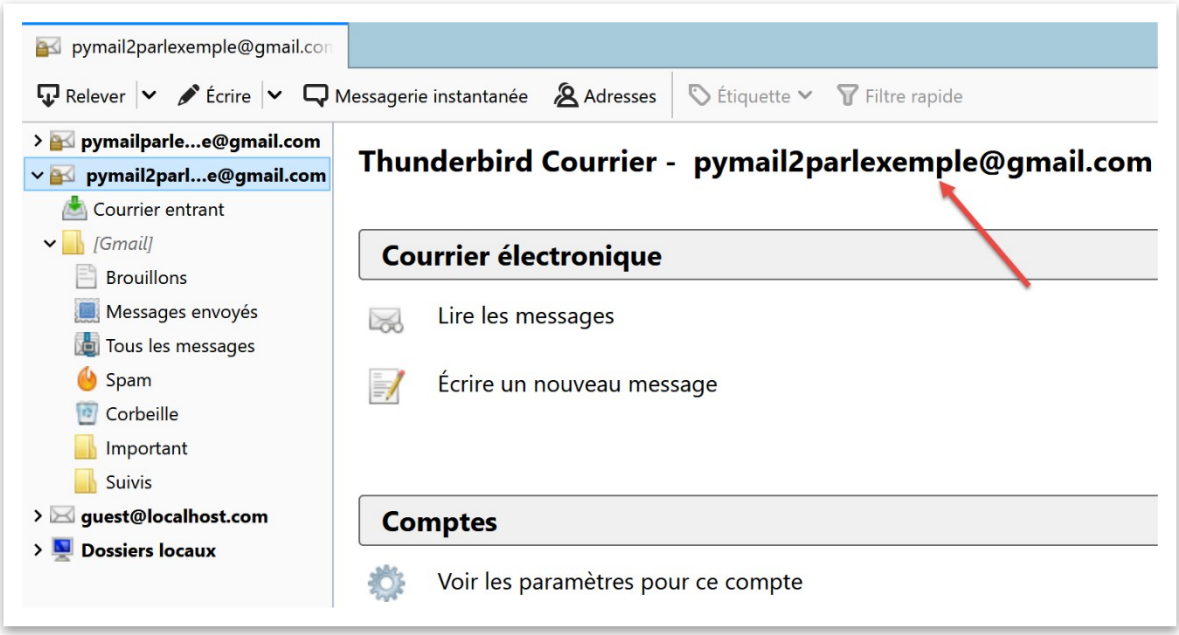


- en `[33-36]` : le courrier reçu par l'utilisateur `[pymailparlexemple@gmail.com]`

Nous créons de même :

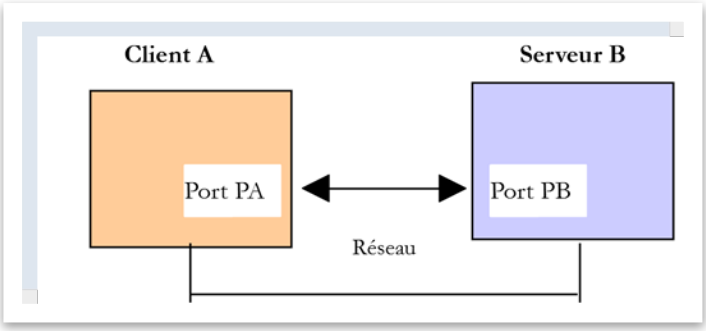
- un nouveau compte Gmail `[pymail2parlexemple@gmail.com]` ;
- un nouverau compte Thunderbird `[pymail2parlexemple@gmail.com]` pour relever les messages de l'utilisateur de même nom :



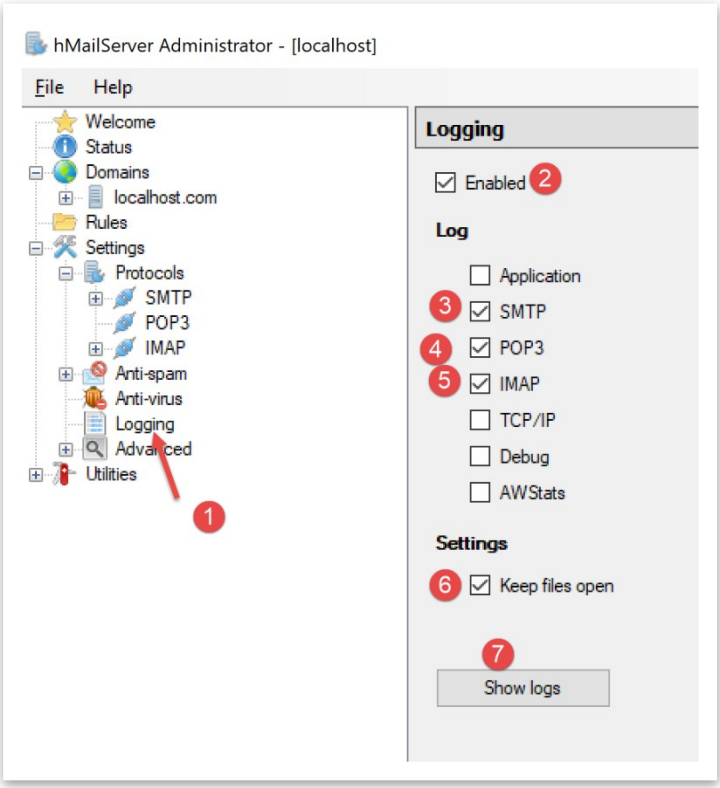


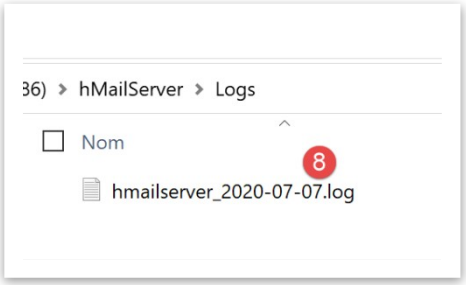
Nous avons désormais les outils pour explorer les protocoles SMTP, POP3 et IMAP. Nous commençons par le protocole SMTP.

21.5.5 Le protocole SMTP

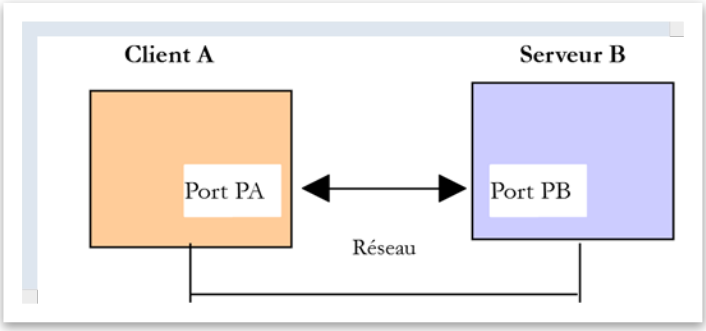


Nous allons découvrir le protocole SMTP en examinant les logs du serveur **[hMailServer]**. Pour cela, nous les activons avec l'outl **[hmailServerAdministrator]** :





- en **[2]**, les logs sont activés ;
- en **[3-5]** : on les active pour les protocoles SMTP, POP3, IMAP ;
- en **[7]**, on demande à les voir ;
- en **[8]**, ouvre le fichier de logs avec un éditeur de texte quelconque ;



Dans l'exemple qui suit, le client sera **[Thunderbird]** et le serveur sera **[hMailServer]**. Avec Thunderbird, faites en sorte que l'utilisateur **[guest@localhost.com]** s'envoie un message à lui-même :

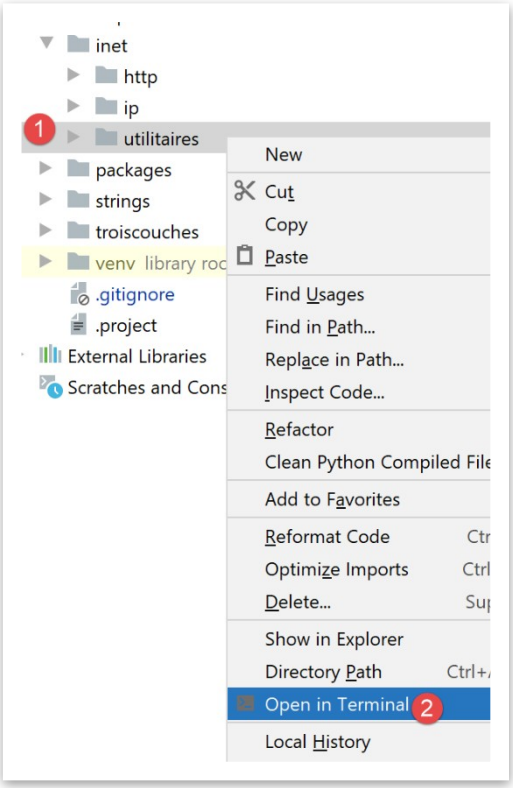
Les logs sont alors les suivants :

```
1.  "SMTPD"  5828   22   "2020-07-07 10:02:54.263"  "127.0.0.1"  "SENT: 220 Bienvenue sur le serveur SMTP
    localhost.com"
2.  "SMTPD"  21956  22   "2020-07-07 10:02:54.360"  "127.0.0.1"  "RECEIVED: EHLO [127.0.0.1]"
3.  "SMTPD"  21956  22   "2020-07-07 10:02:54.362"  "127.0.0.1"  "SENT: 250-DESKTOP-30FF5FB[nl]250-SIZE
    20480000[nl]250-AUTH LOGIN[nl]250 HELP"
4.  "SMTPD"  5828   22   "2020-07-07 10:02:54.381"  "127.0.0.1"  "RECEIVED: MAIL FROM:<guest@localhost.com>
    SIZE=433"
5.  "SMTPD"  5828   22   "2020-07-07 10:02:54.386"  "127.0.0.1"  "SENT: 250 OK"
6.  "SMTPD"  21956  22   "2020-07-07 10:02:54.470"  "127.0.0.1"  "RECEIVED: RCPT TO:<guest@localhost.com>"
7.  "SMTPD"  21956  22   "2020-07-07 10:02:54.473"  "127.0.0.1"  "SENT: 250 OK"
8.  "SMTPD"  21956  22   "2020-07-07 10:02:54.478"  "127.0.0.1"  "RECEIVED: DATA"
9.  "SMTPD"  21956  22   "2020-07-07 10:02:54.479"  "127.0.0.1"  "SENT: 354 OK, send."
10. "SMTPD"  21860  22   "2020-07-07 10:02:54.496"  "127.0.0.1"  "SENT: 250 Queued (0.016 seconds)"
11. "SMTPD"  21568  22   "2020-07-07 10:02:54.505"  "127.0.0.1"  "RECEIVED: QUIT"
12. "SMTPD"  21568  22   "2020-07-07 10:02:54.506"  "127.0.0.1"  "SENT: 221 goodbye"
```

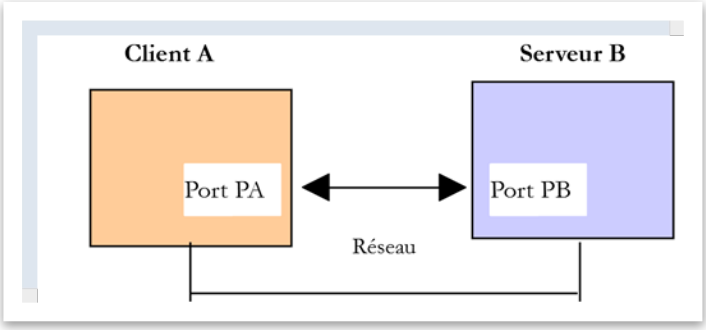
Les lignes ci-dessus décrivent le dialogue qui a eu lieu entre le client SMTP (le gestionnaire de courrier Thunderbird) et le serveur SMTP (hMailServer). Les lignes **[SENT]** indiquent ce que le serveur SMTP a envoyé à son client. Les lignes **[RECEIVED]** indiquent ce que le serveur SMTP a reçu de son client.

- ligne 1 : juste après la connexion du client au serveur SMTP, celui-ci envoie le message de bienvenue à son client ;
- ligne 2 : le client envoie la commande **[EHLO]** pour d'identifier. Ici, il donne son adresse IP **[127.0.0.1]** qui désigne la machine **[localhost]**, ç-à-d la machine qui exécute le client SMTP ;
- ligne 3 : le serveur envoie une série de réponses **[250]**. **[nl]** signifie **[newline]** ç-à-d le caractère \n. Les réponses ont la forme **[250-]** sauf la dernière qui a la forme **[250]**. C'est ainsi que le client SMTP sait que la réponse du serveur SMTP est terminée et qu'il peut envoyer une commande. La série de commandes **[250]** avait pour but d'indiquer au client SMTP une série de commandes qu'il pouvait utiliser ;
- ligne 4 : le client SMTP envoie la commande **[MAIL FROM : adresse_mail_expéditeur]** qui indique qui envoie le message ;
- ligne 5 : le serveur SMTP répond par **[250 OK]** indiquant qu'il a compris la commande ;
- ligne 6 : le client SMTP envoie la commande **[RCPT TO : adresse_mail_destinataire]** pour indiquer l'adresse du destinataire ;
- ligne 7 : de nouveau le serveur SMTP indique qu'il a compris la commande ;
- ligne 8 : le serveur SMTP envoie la commande **[DATA]**. Cela veut dire qu'il va envoyer le contenu du message ;
- ligne 9 : le serveur SMTP indique par la réponse **[354 OK]** qu'il est prêt à recevoir le message. Le texte **[send .]** indique que le client SMTP doit terminer son message par une ligne ne contenant qu'un unique point ;
- ce qu'on ne vois pas ensuite, c'est que le client SMTP envoie son message. Les logs ne l'affichent pas ;
- ligne 10 : le client SMTP a envoyé le point qui indique la fin du message. Le serveur SMTP lui répond qu'il a mis le message en file d'attente (queued) ;
- le client SMTP lui envoie la comande **[QUIT]** pour indiquer qu'il va fermer la connexion ;
- ligne 12 : le serveur lui répond ;

Maintenant que nous connaissons le dialogue client / serveur du protocole SMTP, essayons de le reproduire avec notre client **[RawTcpClient]**. Nous utilisons un terminal PyCharm :



Etudions un nouvel exemple :



- le client A sera le client TCP générique **[RawTcpClient]** ;
- le serveur B sera le serveur de mails **[hMailServer]** ;
- le client A demandera au serveur B de distribuer un courrier envoyé par l'utilisateur **[guest@localhost.com]** pour lui-même ;
- nous vérifierons que le destinataire a bien reçu le mail envoyé ;

Nous lançons le client de la façon suivante :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost 25 --quit bye
2.  Client [DESKTOP-30FF5FB:53122] connecté au serveur [localhost-25]
3.  Tapez vos commandes (quit pour arrêter) :
4.  <-- [220 Bienvenue sur le serveur SMTP localhost.com]
```

- ligne **[1]**, on se connecte sur le port 25 de la machine locale, là où opère le service SMTP de **[hMailServer]**. l'argument **[--quit bye]** indique que l'utilisateur quittera le programme en tapant la commande **[bye]**. Sans cet argument, la commande de fin du programme est **[quit]**. Or **[quit]** est également une commande du protocole SMTP. Il nous faut donc éviter cette ambiguïté ;
- ligne **[2]**, le client est bien connecté ;
- ligne **[3]**, le client attend des commandes tapées au clavier ;
- ligne **[4]**, le serveur lui envoie son message de bienvenue ;

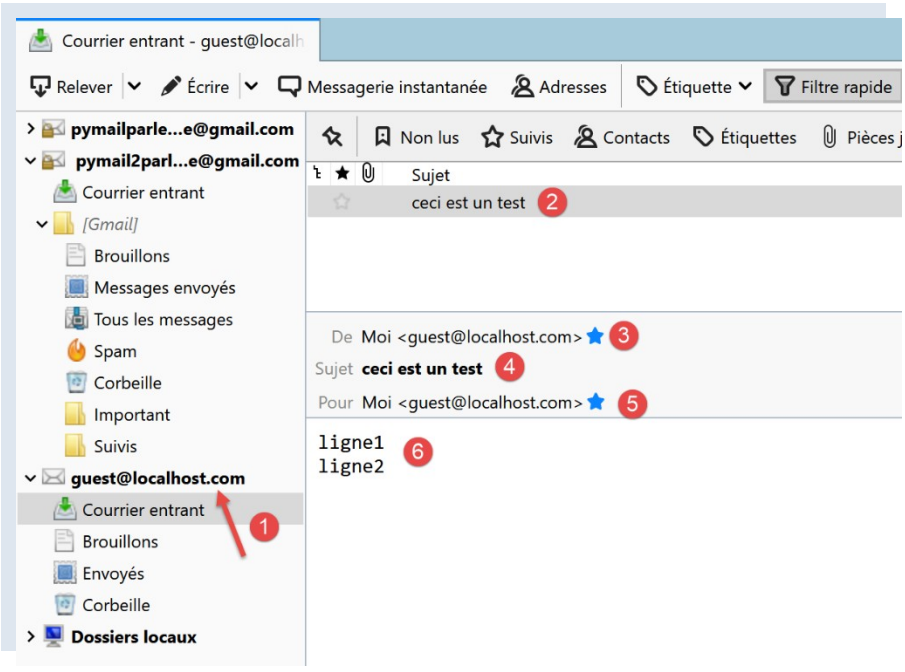
Nous continuons le dialogue de la façon suivante :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost 25
```

```
2.  Client [DESKTOP-30FF5FB:53155] connecté au serveur [localhost-25]
3.  Tapez vos commandes (quit pour arrêter) :
4.  <-- [220 Bienvenue sur le serveur SMTP localhost.com]
5.  EHLO localhost
6.  <-- [250-DESKTOP-30FF5FB]
7.  <-- [250-SIZE 20480000]
8.  <-- [250-AUTH LOGIN]
9.  <-- [250 HELP]
10. MAIL FROM: guest@localhost.com
11. <-- [250 OK]
12. RCPT TO: guest@localhost.com
13. <-- [250 OK]
14. DATA
15. <-- [354 OK, send.]
16. from: guest@localhost.com
17. to: guest@localhost.com
18. subject: ceci est un test
19.
20. ligne1
21. ligne2
22. .
23. <-- [250 Queued (37.824 seconds)]
24. QUIT
25. Fin de la connexion avec le serveur
```

- en **[5]**, le client envoie la commande **[EHLO nom-de-la-machine-client]**. Le serveur lui répond par une suite de messages de la forme **[250-xx]** (6). Le code **[250]** indique le succès de la commande envoyée par le client ;
- en **[10]**, le client indique l'expéditeur du message, ici **[guest@localhost.com]** ;
- en **[11]**, la réponse du serveur ;
- en **[12]**, on indique le destinataire du message, ici l'utilisateur **[guest@localhost.com]** ;
- en **[13]**, la réponse du serveur ;
- en **[14]**, la commande **[DATA]** indique au serveur que le client va envoyer le contenu du message ;
- en **[15]**, la réponse du serveur ;
- en **[16-22]**, le client doit envoyer une liste de lignes de texte terminée par une ligne ne contenant qu'un unique point. Le message peut contenir des lignes **[Subject:, From:, To:]** (16-18) pour définir respectivement le sujet du message, l'expéditeur, le destinataire ;
- en **[19]**, les entêtes précédents doivent être suivis d'une ligne vide ;
- en **[20-21]**, le texte du message ;
- en **[22]**, la ligne ne contenant qu'un unique point qui indique la fin du message ;
- en **[23]**, une fois que le serveur a reçu la ligne ne contenant qu'un unique point, il met le message en file d'attente ;
- en **[24]**, le client indique au serveur qu'il a fini ;
- en **[25]**, on constate que le serveur a fermé la connexion qui le liait au client ;

Maintenant vérifions avec Thunderbird que l'utilisateur **[guest@localhost.com]** a bien reçu le message :



- en **[1-6]**, on voit que l'utilisateur **[guest@localhost.com]** a bien reçu le message ;

Finalement, notre client **[RawTcpClient]** a réussi à envoyer un message via le serveur SMTP **[localhost]**. Maintenant, utilisons la même méthode pour envoyer un message à **[pymailparlexemple@gmail.com]** :

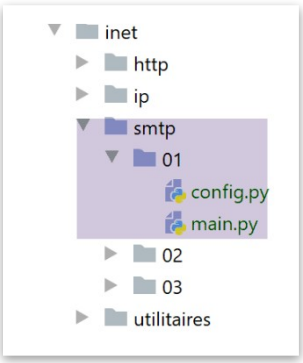
```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe smtp.gmail.com 587
2.  Client [DESKTOP-30FF5FB:53210] connecté au serveur [smtp.gmail.com-587]
3.  Tapez vos commandes (quit pour arrêter) :
4.  <-- [220 smtp.gmail.com ESMTP w13sm643278wrr.67 - gsmtp]
5.  EHLO localhost
6.  <-- [250-smtp.gmail.com at your service, [2a01:cb05:80e8:b500:3c4b:2203:91fa:9b00]]
7.  <-- [250-SIZE 35882577]
8.  <-- [250-8BITMIME]
9.  <-- [250-STARTTLS]
10. <-- [250-ENHANCEDSTATUSCODES]
11. <-- [250-PIPELINING]
12. <-- [250-CHUNKING]
13. <-- [250 SMTPUTF8]
14. MAIL FROM: pymailparlexemple@gmail.com
15. <-- [530 5.7.0 Must issue a STARTTLS command first. w13sm643278wrr.67 - gsmtp]
16. QUIT
17. Fin de la connexion avec le serveur
```

- ligne 1 : on utilise le serveur SMTP de Gmail qui opère sur le port 587 ;
- ligne 15 : on est bloqués parce que le serveur SMTP nous demande de démarrer une connexion sécurisée ce qu'on ne sait pas faire. Contrairement à l'exemple précédent, le serveur **[smtp.gmail.com]** (ligne 1) demande une authentification. Il n'accepte comme clients que les utilisateurs enregistrés dans le domaine **[gmail.com]**. Cette authentification est sécurisée et a lieu au sein d'une connexion cryptée.

Le premier exemple nous a donné les bases pour construire un client SMTP basique en Python. Le deuxième nous a montré que certains serveurs SMTP (la plupart en fait) nécessitent une authentification faite avec une connexion chiffrée.

21.5.6 scripts [smtp/01] : un client SMTP basique

Nous allons reproduire en Python ce que nous avons appris précédemment du protocole SMTP.



Le fichier **[smtp/01/config]** configure l'application de la façon suivante :

```
1.  def configure() -> dict:
2.      return {
3.          # description : description du mail envoyé
4.          # smtp-server : serveur SMTP
5.          # smtp-port : port du serveur SMTP
6.          # from : expéditeur
7.          # to : destinataire
8.          # subject : sujet du mail
9.          # message : message du mail
10.         "mails": [
11.             {
12.                 "description": "mail to localhost via localhost",
13.                 "smtp-server": "localhost",
14.                 "smtp-port": "25",
15.                 "from": "guest@localhost.com",
16.                 "to": "guest@localhost.com",
17.                 "subject": "to localhost via localhost",
18.                 # on envoie de l'UTF-8
19.                 "content-type": 'text/plain; charset="utf-8"',
20.                 # on teste les caractères accentués
21.                 "message": "aglaë séléné\nva au marché\nacheter des fleurs"
22.             },
```

```
23.             {
24.                 "description": "mail to gmail via gmail",
25.                 "smtp-server": "smtp.gmail.com",
26.                 "smtp-port": "587",
27.                 "from": "pymailparlexemple@gmail.com",
28.                 "to": "pymailparlexemple@gmail.com",
29.                 "subject": "to gmail via gmail",
30.                 # on envoie de l'UTF-8
31.                 "Content-type": 'text/plain; charset="utf-8"',
32.                 # on teste les caractères accentués
33.                 "message": "aglaë séléné\nva au marché\nacheter des fleurs"
34.             }
35.         ]
36.     }
```

- lignes 10-35 : une liste de mails à envoyer. Pour chacun d'eux on précise les informations suivantes :
  - **[description]** : un texte décrivant le mail ;
  - **[smtp-server]** : le serveur SMTP à utiliser ;
  - **[smtp-port]** : son port de service ;
  - **[from]** : l'expéditeur du mail ;
  - **[to]** : le destinataire du mail ;
  - **[subject]** : le sujet du mail ;
  - **[content-type]** : l'encodage du mail ;
  - **[message]** : le message du mail ;

Le code **[01/main]** du client SMTP est le suivant :

```
1.  # imports
2.  import socket
3.
4.
5.  # ---------------------------------------------------------------------
6.  def sendmail(mail: dict, verbose: bool):
7.      # envoie message au serveur smtp smtpserver de la part de expéditeur
8.      # pour destinataire. Si verbose=True, fait un suivi des échanges client-serveur
9.
10.     # on laisse remonter les erreurs système
11.     connexion = None
12.     try:
13.         # nom de la machine locale (nécessaire au protocole SMTP)
14.         client = socket.gethostbyaddr(socket.gethostbyname("localhost"))[0]
15.         # ouverture d'une connexion sur le port 25 de smtpServer
16.         connexion = socket.create_connection((mail["smtp-server"], 25))
17.
18.         # connexion représente un flux de communication bidirectionnel
19.         # entre le client (ce programme) et le serveur smtp contacté
20.         # ce canal est utilisé pour les échanges de commandes et d'informations
21.
22.         # après la connexion le serveur envoie un message de bienvenue qu'on lit
23.         send_command(connexion, "", verbose, True)
24.         # cmde ehlo:
25.         send_command(connexion, f"EHLO {client}", verbose, True)
26.         # cmde mail from:
27.         send_command(connexion, f"MAIL FROM: <{mail['from']}>", verbose, True)
28.         # cmde rcpt to:
29.         send_command(connexion, f"RCPT TO: <{mail['to']}>", verbose, True)
30.         # cmde data
31.         send_command(connexion, "DATA", verbose, True)
32.         # préparation message à envoyer
33.         # il doit contenir les lignes
34.         # From: expéditeur
35.         # To: destinataire
36.         # ligne vide
37.         # Message
38.         # .
39.         data = f"{mail['message']}"
40.         # envoi message
41.         send_command(connexion, data, verbose, False)
42.         # envoi .
43.         send_command(connexion, "\r\n.\r\n", verbose, False)
44.         # cmde quit
45.         send_command(connexion, "QUIT", verbose, True)
46.         # fin
47.     finally:
48.         # fermeture connexion
```

```
        if connexion:
            connexion.close()


# ------------------------------------------------------------------------
def send_command(connexion: socket, commande: str, verbose: bool, with_rclf: bool):
    # envoie commande dans le canal connexion
    # mode verbeux si verbose=True
    # si with_rclf=True, ajoute la séquence rclf à commande

    # données
    rclf = "\r\n" if with_rclf else ""
    # envoi cmde si commande non vide
    if commande:
        # on laisse remonter les erreurs système
        #
        # envoi commande
        connexion.send(bytearray(f"{commande}{rclf}", 'utf-8'))
        # écho éventuel
        if verbose:
            affiche(commande, 1)
        # lecture réponse de moins de 1000 caractères
        reponse = str(connexion.recv(1000), 'utf-8')
        # écho éventuel
        if verbose:
            affiche(reponse, 2)
        # récupération code erreur
        codeErreur = int(reponse[0:3])
        # erreur renvoyée par le serveur ?
        if codeErreur >= 500:
            # on lance une exception avec l'erreur
            raise BaseException(reponse[4:])
        # retour sans erreur


# ------------------------------------------------------------------------
def affiche(echange: str, sens: int):
    # affiche échange ? l'écran
    # si sens=1 affiche -->echange
    # si sens=2 affiche <-- échange sans les 2 derniers caractères rclf
    if sens == 1:
        print(f"--> [{echange}]")
        return
    elif sens == 2:
        l = len(echange)
        print(f"<-- [{echange[0:l - 2]}]")
        return


# main ----------------------------------------------------------------

# client SMTP (SendMail Transfer Protocol) permettant d'envoyer un message
# les infos sont prises dans un fichier config contenant les informations suivantes pour chaque serveur

# description : description du mail envoyé
# smtp-server : serveur SMTP
# smtp-port : port du serveur SMTP
# from : expéditeur
# to : destinataire
# subject : sujet du mail
# message : message du mail


# protocole de communication SMTP client-serveur
# -> client se connecte sur le port 25 du serveur smtp
# <- serveur lui envoie un message de bienvenue
# -> client envoie la commande EHLO: nom de sa machine
# <- serveur répond OK ou non
# -> client envoie la commande mail from: <exp?diteur>
# <- serveur répond OK ou non
# -> client envoie la commande rcpt to: <destinataire>
# <- serveur répond OK ou non
# -> client envoie la commande data
# <- serveur répond OK ou non
# -> client envoie ttes les lignes de son message et termine avec une ligne contenant le seul caractère .
# <- serveur répond OK ou non
# -> client envoie la commande quit
```

```
126.# <- serveur répond OK ou non
127.
128.# les réponses du serveur ont la forme xxx texte où xxx est un nombre à 3 chiffres. Tout nombre xxx >=500
129.# signale une erreur. La réponse peut comporter plusieurs lignes commençant toutes par xxx- sauf la
    dernière
130.# de la forme xxx(espace)
131.
132.# les lignes de texte échangées doivent se terminer par les caractéres RC(#13) et LF(#10)
133.
134.# configuration de l'application
135.import config
136.config = config.configure()
137.
138.# on traite les mails un par un
139.for mail in config['mails']:
140.    try:
141.        # logs
142.        print("----------------------------------")
143.        print(f"Envoi du message [{mail['description']}]")
144.        # préparation du message à envoyer
145.        mail[
146.            "message"] = f"From: {mail['from']}\nTo: {mail['to']}\n" \
147.                         f"Subject: {mail['subject']}\n" \
148.                         f"Content-type: {mail['content-type']}" \
149.                         f"\n\n{mail['message']}"
150.        # envoi du message en mode verbeux
151.        sendmail(mail, True)
152.        # fin
153.        print("Message envoyé...")
154.    except BaseException as erreur:
155.        # on affiche l'erreur
156.        print(f"L'erreur suivante s'est produite : {erreur}")
157.    finally:
158.        pass
159.    # mail suivant
```

Commentaires

- lignes 134-136 : on configure l'application ;
- lignes 139-151 : on tarite tous les mails trouvés dans la configuration ;
- lignes 141-143 : on affiche ce qu'on va faire ;
- lignes 144-149 : on définit le message à envoyer. Le message **[message]** est précédé des entêtes **[From, To, Subject, Content-type]** ;
- ligne 151 : l'envoi du mail est assuré par la fonction **[sendmail]** qui admet deux paramètres :
  - **[mail]** : le dictionnaire contenant les informations nécessaires à l'envoi du mail ;
  - **[verbose]** : un booléen indiquant si les échanges client / serveur doivent être ou non logués sur la console ;
- lignes 154-156 : on arrête toutes les exceptions qui sortent de la fonction **[sendmail]**. Elles sont affichées ;
- ligne 6 : **[mail]** est le dictionnaire décrivant le mail à envoyer ;
- ligne 14 : dans le protocole SMTP, le client doit envoyer son non. On récupère ici le nom de la machine locale qui va servir de client ;
- ligne 16 : connexion au serveur SMTP à qui le message va être envoyé ;
- lignes 22-23 : si la connexion s'est faite avec le serveur SMTP, celui-ci va envoyer un message de bienvenue qu'on lit ici ;
- la fonction **[sendmail]** envoie ensuite les différentes commandes que doit envoyer un client SMTP :
  - lignes 24-25 : la commande EHLO ;
  - lignes 26-27 : la commande MAIL FROM: ;
  - lignes 28-29 : la commande RCPT TO: ;
  - lignes 30-31 : la commande DATA ;
  - lignes 32-41 : envoi du message (From, To, Subject, Content-type, texte) ;
  - lignes 42-43 : envoi du point final ;
  - lignes 44-457 : la commande QUIT qui termine le dialogue du client avec le serveur SMTP ;
- l'exécution de **[sendmail]** s'exécute dans un **[try / finally]** qui laisse remonter toutes les exceptions au code appelant. On sait que celui-ci les arrête toutes pour les afficher ;
- lignes 48-50 : libération des ressources ;
- ligne 54 : la fonction **[send_command]** est chargée d'envoyer les commandes du client au serveur SMTP. Elle admet quatre paramètres :
  - **[connexion]** : la connexion qui relie le client au serveur ;
  - **[commande]** : la commande à envoyer ;
  - **[verbose]** : si TRUE alors les échanges client / serveur sont logués sur la console ;
  - **[with_rclf]** : si TRUE, envoie la commande terminée par la séquence \r\n. C'est nécessaire pour toutes les commandes du protocole SMTP, mais **[send_command]** sert aussi à envoyer le message. Là on n'ajoute pas la séquence \r\n ;

- ligne 62 : la commande n'est envoyée que si elle est non vide ;
- lignes 65-66 : la commande est envoyée au serveur sous la forme d'une chaîne d'octets UTF-8 ;
- lignes 70-71 : lecture de de l'ensemble des lignes de la réponse. On suppose qu'elle fait moins de 1000 caractères. La réponse peut comporter plusieurs lignes. Chaque ligne a la forme XXX-YYY où XXX est un code numérique sauf la dernière ligne de la réponse qui a la forme XXX YYY (absence du caractère -) ;
- lignes 76 : lecture du code d'erreur XXX de la 1re ligne ;
- lignes 78-80 : si le code numérique XXX est supérieur à 500, alors le serveur a renvoyé une erreur. On lance alors une exception ;

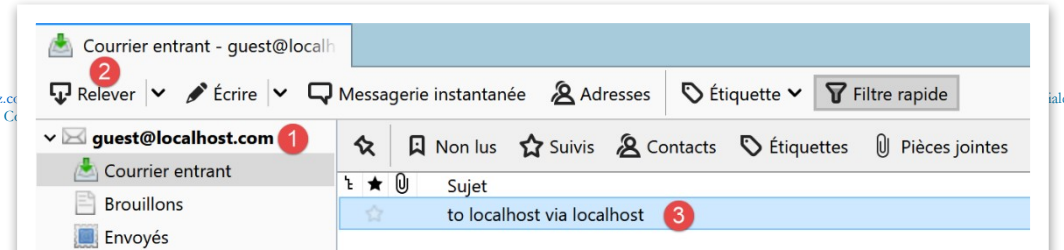
Résultats

L'exécution du script donne les résultats console suivants :

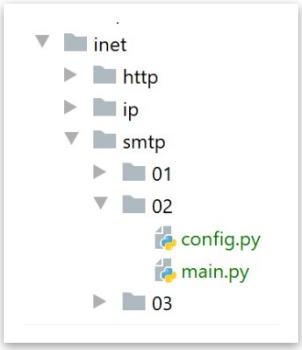
```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/inet/smtp/01/main.py
2.  ----------------------------------
3.  Envoi du message [mail to localhost via localhost]
4.  --> [EHLO DESKTOP-30FF5FB]
5.  <-- [220 Bienvenue sur le serveur SMTP localhost.com]
6.  --> [MAIL FROM: <guest@localhost.com>]
7.  <-- [250-DESKTOP-30FF5FB
8.  250-SIZE 20480000
9.  250-AUTH LOGIN
10. 250 HELP]
11. --> [RCPT TO: <guest@localhost.com>]
12. <-- [250 OK]
13. --> [DATA]
14. <-- [250 OK]
15. --> [From: guest@localhost.com
16. To: guest@localhost.com
17. Subject: to localhost via localhost
18. Content-type: text/plain; charset="utf-8"
19.
20. aglaë séléné
21. va au marché
22. acheter des fleurs]
23. <-- [354 OK, send.]
24. --> [
25. .
26. ]
27. <-- [250 Queued (0.000 seconds)]
28. --> [QUIT]
29. <-- [221 goodbye]
30. Message envoyé...
31. ----------------------------------
32. Envoi du message [mail to gmail via gmail]
33. --> [EHLO DESKTOP-30FF5FB]
34. <-- [220 smtp.gmail.com ESMTP u1sm1364433wrb.78 - gsmtp]
35. --> [MAIL FROM: <pymailparlexemple@gmail.com>]
36. <-- [250-smtp.gmail.com at your service, [2a01:cb05:80e8:b500:3c4b:2203:91fa:9b00]
37. 250-SIZE 35882577
38. 250-8BITMIME
39. 250-STARTTLS
40. 250-ENHANCEDSTATUSCODES
41. 250-PIPELINING
42. 250-CHUNKING
43. 250 SMTPUTF8]
44. --> [RCPT TO: <pymailparlexemple@gmail.com>]
45. <-- [530 5.7.0 Must issue a STARTTLS command first. u1sm1364433wrb.78 - gsmtp]
46. L'erreur suivante s'est produite : 5.7.0 Must issue a STARTTLS command first. u1sm1364433wrb.78 - gsmtp
47.
48.
49. Process finished with exit code 0
```

- lignes 3-30 : l'utilisation du serveur SMTP **[hMailServer]** pour envoyer un mail à **[guest@localhost]** se passe bien ;
- lignes 32-46 : l'utilisation du serveur SMTP **[smtp.gmail.com]** pour envoyer un mail à **[pymailparlexemple@gmail.com]** ne se passe pas bien : en ligne 45, le serveur SMTP envoie un code d'erreur 530 avec un message d'erreur. Celui-ci indique que le client SMTP doit au préalable s'authentifier via une connexion sécurisée. Notre client ne l'a pas fait et est donc refusé ;

Les résultats dans Thunderbird sont les suivants :



21.5.7 scripts `[smtp/02]` : un lient SMTP écrit avec la bibliothèque `[smtplib]`



Le client précédent souffre d'au moins deux insuffisances :

1. il ne sait pas utiliser une connexion sécurisée si le serveur la réclame ;
2. il ne sait pas joindre des attachements au message ;

Nous allons traiter la première insuffisance dans le script **`[smtp/02]`**. Dans notre nouveau script nous allons utiliser le module Python **`[smtplib]`**.

Le script **`[smtp/02/main]`** utilisera le fichier de configuration jSON **`[smtp/02/config]`** suivant :

```
1.  def configure() -> dict:
2.      return {
3.          # description : description du mail envoyé
4.          # smtp-server : serveur SMTP
5.          # smtp-port : port du serveur SMTP
6.          # from : expéditeur
7.          # to : destinataire
8.          # subject : sujet du mail
9.          # message : message du mail
10.         "mails": [
11.             {
12.                 "description": "mail to localhost via localhost avec smtplib",
13.                 "smtp-server": "localhost",
14.                 "smtp-port": "25",
15.                 "from": "guest@localhost.com",
16.                 "to": "guest@localhost.com",
17.                 "subject": "to localhost via localhost avec smtplib",
18.                 # on teste les caractères accentués
19.                 "message": "aglaë séléné\nva au marché\nacheter des fleurs",
20.             },
21.             {
22.                 "description": "mail to gmail via gmail avec smtplib",
23.                 "smtp-server": "smtp.gmail.com",
24.                 "smtp-port": "587",
25.                 "from": "pymail2parlexemple@gmail.com",
26.                 "to": "pymail2parlexemple@gmail.com",
27.                 "subject": "to gmail via gmail avec smtplib",
28.                 # on teste les caractères accentués
29.                 "message": "aglaë séléné\nva au marché\nacheter des fleurs",
30.                 # smtp avec authentification
31.                 "user": "pymail2parlexemple@gmail.com",
32.                 "password": "#6prIlh@1QZ3TG",
33.             }
34.         ]
35.     }
```

On retrouve les mêmes rubriques que dans le fichier **`[smtp/01/config]`** avec deux rubriques supplémentaires lorsque el serveur SMTP demande une authentification :

- ligne 31, **`[user]`** : le nom de l'utilisateur qui authentifie la connexion ;
- ligne 32, **`[password]`** : son mot de passe ;

Ces deux rubriques ne sont présentes que si le serveur SMTP contacté exige une authentification. Celle-ci se fait alors au-travers d'une connexion sécurisée.

Le code du script **[smtp/02/main.py]** est le suivant :

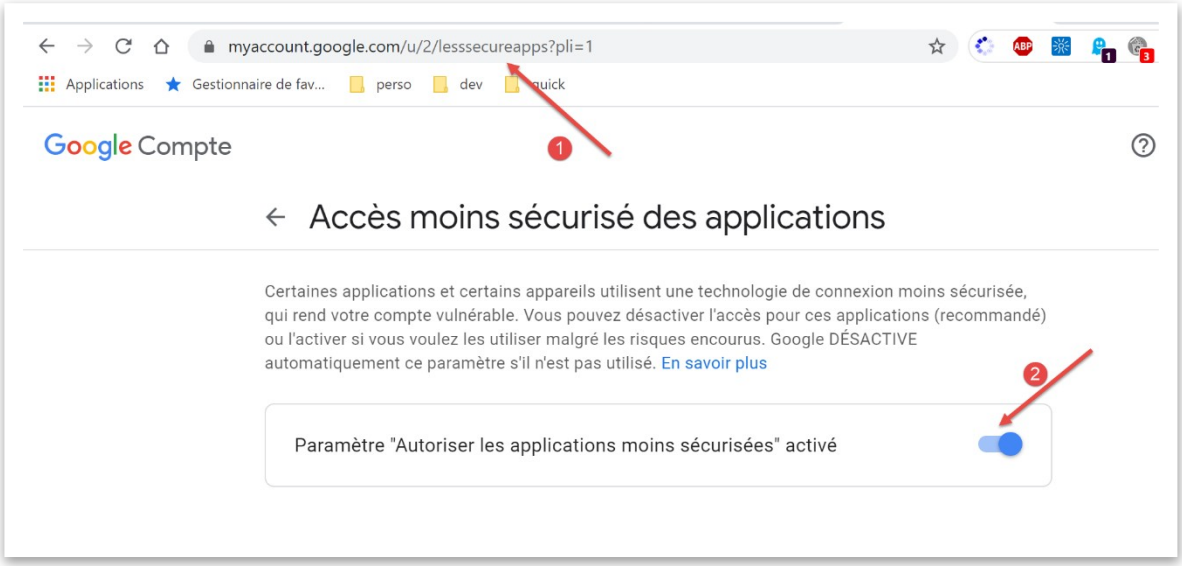
```
1.  # imports
2.  import smtplib
3.  from email.mime.text import MIMEText
4.  from email.utils import formatdate
5.
6.
7.  # ---------------------------------------------------------------------
8.  def sendmail(mail: dict, verbose: True):
9.      # envoie message au serveur smtp smtpserver de la part de expéditeur
10.     # pour destinataire. Si verbose=True, fait un suivi des échanges client-serveur
11.
12.     # on utilise la bibliothéque smtplib
13.     # on laisse remonter les exceptions
14.     #
15.     # le serveur SMTP
16.     server = smtplib.SMTP(mail["smtp-server"])
17.     # mode verbose
18.     server.set_debuglevel(verbose)
19.     # connexion sécurisée ?
20.     if "user" in mail:
21.         # connexion sécurisée
22.         server.starttls()
23.         # EHLO commande + authentification
24.         server.login(mail["user"], mail["password"])
25.
26.    # construction d'un message Multipart - c'est ce message qui Multipart sera envoyé
27.     msg = MIMEText(mail["message"])
28.     msg['from'] = mail["from"]
29.     msg['to'] = mail["to"]
30.     msg['date'] = formatdate(localtime=True)
31.     msg['subject'] = mail["subject"]
32.     # on envoie le message
33.     server.send_message(msg)
34.     # on quitte
35.     server.quit()
36.
37.
38. # main ----------------------------------------------------------------
39.
40. # les infos sont prises dans un fichier config contenant les informations suivantes pour chaque serveur
41.
42. # description : description du mail envoyé
43. # smtp-server : serveur SMTP
44. # smtp-port : port du serveur SMTP
45. # from : expéditeur
46. # to : destinataire
47. # subject : sujet du mail
48. # content-type : encodage du mail
49. # message : message du mail
50.
51.
52. # configurationn de l'application
53. import config
54. config = config.configure()
55.
56. # on traite les mails un par un
57. for mail in config['mails']:
58.     try:
59.         # logs
60.         print("----------------------------------")
61.         print(f"Envoi du message [{mail['description']}]")
62.         # envoi du message en mode verbeux
63.         sendmail(mail, True)
64.         # fin
65.         print("Message envoyé...")
66.     except BaseException as erreur:
67.         # on affiche l'erreur
68.         print(f"L'erreur suivante s'est produite : {erreur}")
69.     finally:
70.         pass
71.     # mail suivant
```

Commentaires

- lignes 8-35 : seule la fonction **[sendmail]** est utilisée. Elle va désormais utiliser le module **[smtplib]** (ligne 2) ;
- ligne 16 : connexion au serveur SMTP ;
- ligne 18 : si **[verbose=True]**, les échanges client / serveur seront affichés sur la console ;
- lignes 20-24 : on fait l'éventuelle authentification si le serveur SMTP l'exige ;
- ligne 22 : l'authentification se fait au travers d'une connexion sécurisée ;
- ligne 24 : authentification ;
- lignes 26-33 : envoi du message. Le dialogue vu avec le script **[smtp/01/main]** va alors se dérouler. S'il y a eu authentification, il se déroulera au sein d'une connexion sécurisée ;
- ligne 35 : on termine le dialogue client / serveur ;

Avant d'exécuter le script **[smtp/02/main]**, vous devez modifier la configuration du compte Gmail **[pymailparlexemple@gmail.com]** :

- connectez-vous au compte Gmail **[pymailparlexemple@gmail.com]** ;
- modifiez la configuration suivante :



- en **[2]**, autorisez les applications moins sécurisées à accéder au compte ;

Faites la même chose avec le second compte Gmail **[pymail2parlexemple@gmail.com]**.

Résultats

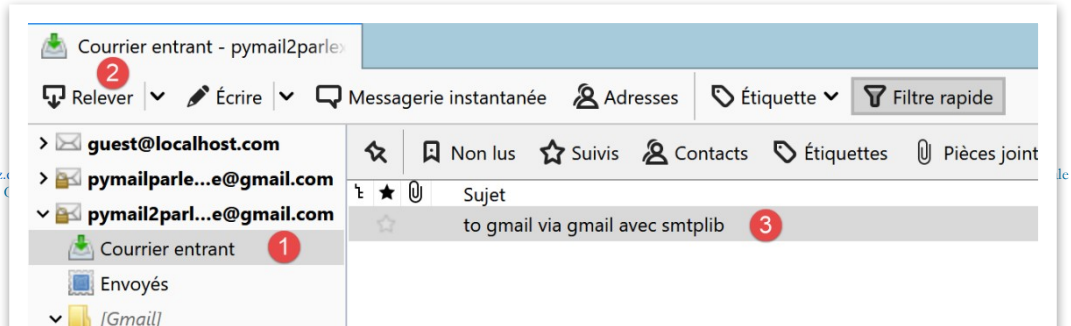
Lorsqu'on exécute le script **[smtp/02/main]** on obtient les résultats console suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/inet/smtp/02/main.py
2.  ----------------------------------
3.  Envoi du message [mail to localhost via localhost avec smtplib]
4.  send: 'ehlo [192.168.43.163]\r\n'
5.  reply: b'250-DESKTOP-30FF5FB\r\n'
6.  reply: b'250-SIZE 20480000\r\n'
7.  reply: b'250-AUTH LOGIN\r\n'
8.  reply: b'250 HELP\r\n'
9.  reply: retcode (250); Msg: b'DESKTOP-30FF5FB\nSIZE 20480000\nAUTH LOGIN\nHELP'
10. send: 'mail FROM:<guest@localhost.com> size=310\r\n'
11. reply: b'250 OK\r\n'
12. reply: retcode (250); Msg: b'OK'
13. send: 'rcpt TO:<guest@localhost.com>\r\n'
14. reply: b'250 OK\r\n'
15. reply: retcode (250); Msg: b'OK'
16. send: 'data\r\n'
17. reply: b'354 OK, send.\r\n'
18. reply: retcode (354); Msg: b'OK, send.'
19. data: (354, b'OK, send.')
```

```
20. send: b'Content-Type: text/plain; charset="utf-8"\r\nMIME-Version: 1.0\r\nContent-Transfer-Encoding:
    base64\r\nfrom: guest@localhost.com\r\nto: guest@localhost.com\r\ndate: Wed, 08 Jul 2020 08:35:39
    +0200\r\nsubject: to localhost via localhost avec
    smtplib\r\n\r\nYWdsYcOrIHPDqWzDqW7DqQp2YSBhdSBtYXJjaMOpCmFjaGV0ZXIgZGVzIGZsZXVycw==\r\n.\r\n'
21. reply: b'250 Queued (0.000 seconds)\r\n'
22. reply: retcode (250); Msg: b'Queued (0.000 seconds)'
23. data: (250, b'Queued (0.000 seconds)')
24. send: 'quit\r\n'
25. reply: b'221 goodbye\r\n'
26. reply: retcode (221); Msg: b'goodbye'
27. Message envoyé...
28. ---------------------------------
29. Envoi du message [mail to gmail via gmail avec smtplib]
30. send: 'ehlo [192.168.43.163]\r\n'
31. reply: b'250-smtp.gmail.com at your service, [37.172.118.130]\r\n'
32. reply: b'250-SIZE 35882577\r\n'
33. reply: b'250-8BITMIME\r\n'
34. reply: b'250-STARTTLS\r\n'
35. reply: b'250-ENHANCEDSTATUSCODES\r\n'
36. reply: b'250-PIPELINING\r\n'
37. reply: b'250-CHUNKING\r\n'
38. reply: b'250 SMTPUTF8\r\n'
39. reply: retcode (250); Msg: b'smtp.gmail.com at your service, [37.172.118.130]\nSIZE
    35882577\n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nPIPELINING\nCHUNKING\nSMTPUTF8'
40. send: 'STARTTLS\r\n'
41. reply: b'220 2.0.0 Ready to start TLS\r\n'
42. reply: retcode (220); Msg: b'2.0.0 Ready to start TLS'
43. send: 'ehlo [192.168.43.163]\r\n'
44. reply: b'250-smtp.gmail.com at your service, [37.172.118.130]\r\n'
45. reply: b'250-SIZE 35882577\r\n'
46. reply: b'250-8BITMIME\r\n'
47. reply: b'250-AUTH LOGIN PLAIN XOAUTH2 PLAIN-CLIENTTOKEN OAUTHBEARER XOAUTH\r\n'
48. reply: b'250-ENHANCEDSTATUSCODES\r\n'
49. reply: b'250-PIPELINING\r\n'
50. reply: b'250-CHUNKING\r\n'
51. reply: b'250 SMTPUTF8\r\n'
52. reply: retcode (250); Msg: b'smtp.gmail.com at your service, [37.172.118.130]\nSIZE
    35882577\n8BITMIME\nAUTH LOGIN PLAIN XOAUTH2 PLAIN-CLIENTTOKEN OAUTHBEARER
    XOAUTH\nENHANCEDSTATUSCODES\nPIPELINING\nCHUNKING\nSMTPUTF8'
53. send: 'AUTH PLAIN AHB5bWFpbDJwYXJsZXhlbXBsZUBnbWFpbC5jb20AIzZwcklsaEQmQDFRWjNURw==\r\n'
54. reply: b'235 2.7.0 Accepted\r\n'
55. reply: retcode (235); Msg: b'2.7.0 Accepted'
56. send: 'mail FROM:<pymail2parlexemple@gmail.com> size=320\r\n'
57. reply: b'250 2.1.0 OK e5sm4132618wrs.33 - gsmtp\r\n'
58. reply: retcode (250); Msg: b'2.1.0 OK e5sm4132618wrs.33 - gsmtp'
59. send: 'rcpt TO:<pymail2parlexemple@gmail.com>\r\n'
60. reply: b'250 2.1.5 OK e5sm4132618wrs.33 - gsmtp\r\n'
61. reply: retcode (250); Msg: b'2.1.5 OK e5sm4132618wrs.33 - gsmtp'
62. send: 'data\r\n'
63. reply: b'354  Go ahead e5sm4132618wrs.33 - gsmtp\r\n'
64. reply: retcode (354); Msg: b'Go ahead e5sm4132618wrs.33 - gsmtp'
65. data: (354, b'Go ahead e5sm4132618wrs.33 - gsmtp')
66. send: b'Content-Type: text/plain; charset="utf-8"\r\nMIME-Version: 1.0\r\nContent-Transfer-Encoding:
    base64\r\nfrom: pymail2parlexemple@gmail.com\r\nto: pymail2parlexemple@gmail.com\r\ndate: Wed, 08 Jul 2020
    08:35:40 +0200\r\nsubject: to gmail via gmail avec
    smtplib\r\n\r\nYWdsYcOrIHPDqWzDqW7DqQp2YSBhdSBtYXJjaMOpCmFjaGV0ZXIgZGVzIGZsZXVycw==\r\n.\r\n'
67. reply: b'250 2.0.0 OK  1594190139 e5sm4132618wrs.33 - gsmtp\r\n'
68. reply: retcode (250); Msg: b'2.0.0 OK  1594190139 e5sm4132618wrs.33 - gsmtp'
69. data: (250, b'2.0.0 OK  1594190139 e5sm4132618wrs.33 - gsmtp')
70. send: 'quit\r\n'
71. Message envoyé...
72. reply: b'221 2.0.0 closing connection e5sm4132618wrs.33 - gsmtp\r\n'
73. reply: retcode (221); Msg: b'2.0.0 closing connection e5sm4132618wrs.33 - gsmtp'
74.
75. Process finished with exit code 0
```

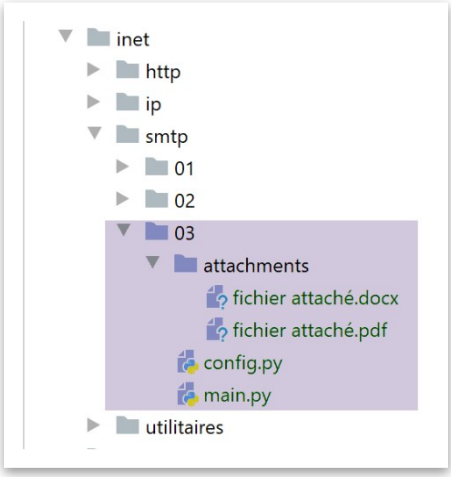
- ligne 40 : le client **[smtplib]** commence le dialogue pour établir une liaison cryptée avec le serveur SMTP, ce qu'on n'avait pas su faire dans le script **[smtp/main/01]** ;
- sinon on retouve les commandes connues du protocole SMTP ;

Si on consulte le compte Gmail de l'utilisateur **[pymail2parlexemple]** on a la chose suivante :



21.5.8 scripts [smtp/03] : gestion des fichiers attachés

Nous complétons le script **[smtp/02/main]** afin que le mail envoyé puisse avoir des fichiers attachés.



Le script **[smtp/03/main]** est configuré par le script **[smtp/03/config]** suivant :

```
1.  import os
2.
3.
4.  def configure() -> dict:
5.      # configuration de l'application
6.      script_dir = os.path.dirname(os.path.abspath(__file__))
7.
8.      return {
9.          # description : description du mail envoyé
10.         # smtp-server : serveur SMTP
11.         # smtp-port : port du serveur SMTP
12.         # from : expéditeur
13.         # to : destinataire
14.         # subject : sujet du mail
15.         # message : message du mail
16.         "mails": [
17.             {
18.                 "description": "mail to gmail via gmail avec smtplib",
19.                 "smtp-server": "smtp.gmail.com",
20.                 "smtp-port": "587",
21.                 "from": "pymail2parlexemple@gmail.com",
22.                 "to": "pymail2parlexemple@gmail.com",
23.                 "subject": "to gmail via gmail avec smtplib",
24.                 # on teste les caractères accentués
25.                 "message": "aglaë séléné\nva au marché\nacheter des fleurs",
26.                 # smtp avec authentification
27.                 "user": "pymail2parlexemple@gmail.com",
28.                 "password": "#6prIlhD&@1QZ3TG",
29.                 # ici, il faut mettre des chemins absolus pour les fichiers attachés
30.                 "attachments": [
31.                     f"{script_dir}/attachments/fichier attaché.docx",
32.                     f"{script_dir}/attachments/fichier attaché.pdf",
33.                 ]
34.             }
35.         ]
36.     }
```

Le fichier **[smtp/03/config]** ne diffère du fichier **[smtp/02/config]** utilisé précédemment que par la présence facultative d'une liste **[attachments]** (lignes 30-32) qui désigne la liste des fichiers à attacher au message à envoyer.

Le script **[smtp/03/main]** est le suivant :

```
1.  # imports
2.  import email
3.  import mimetypes
```

```
import os
import smtplib
from email import encoders
from email.mime.audio import MIMEAudio
from email.mime.base import MIMEBase
from email.mime.image import MIMEImage
from email.mime.message import MIMEMessage
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.utils import formatdate



# ---------------------------------------------------------------------
def sendmail(mail: dict, verbose: True):
    # envoie mail[message] au serveur smtp mail[smtp-server] de la part de mail[from]
    # pour mail[to]. Si verbose=True, fait un suivi des échanges client-serveur

    # on utilise la bibliothéque smtplib
    # on laisse remonter les exceptions
    #
    # le serveur SMTP
    server = smtplib.SMTP(mail["smtp-server"])
    # mode verbose
    server.set_debuglevel(verbose)
    # connexion sécurisée ?
    if "user" in mail:
        server.starttls()
        server.login(mail["user"], mail["password"])

    # construction d'un message Multipart - c'est le message qui sera envoyé
    # credit : https://docs.python.org/3.4/library/email-examples.html
    msg = MIMEMultipart()
    msg['From'] = mail["from"]
    msg['To'] = mail["to"]
    msg['Date'] = formatdate(localtime=True)
    msg['Subject'] = mail["subject"]
    # on attache le message texte au format MIMEText
    msg.attach(MIMEText(mail["message"]))
    # on parcourt les attachements
    for path in mail["attachments"]:
        # path doit être un chemin absolu
        # on devine le type du fichier attaché
        ctype, encoding = mimetypes.guess_type(path)
        # si on n'a pas deviné
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        # on décompose le type en maintype/subtype
        maintype, subtype = ctype.split('/', 1)
        # on traite les différents cas
        if maintype == 'text':
            with open(path) as fp:
                # Note: we should handle calculating the charset
                part = MIMEText(fp.read(), _subtype=subtype)
        elif maintype == 'image':
            with open(path, 'rb') as fp:
                part = MIMEImage(fp.read(), _subtype=subtype)
        elif maintype == 'audio':
            with open(path, 'rb') as fp:
                part = MIMEAudio(fp.read(), _subtype=subtype)
        # cas du type message / rfc822
        elif maintype == 'message':
            with open(path, 'rb') as fp:
                part = MIMEMessage(email.message_from_bytes(fp.read()))
        else:
            # autres cas
            with open(path, 'rb') as fp:
                part = MIMEBase(maintype, subtype)
                part.set_payload(fp.read())
            # Encode the payload using Base64
            encoders.encode_base64(part)
        # Set the filename parameter
        basename = os.path.basename(path)
        part.add_header('Content-Disposition', 'attachment', filename=basename)
        # on attache le fichier au message à envoyer
```

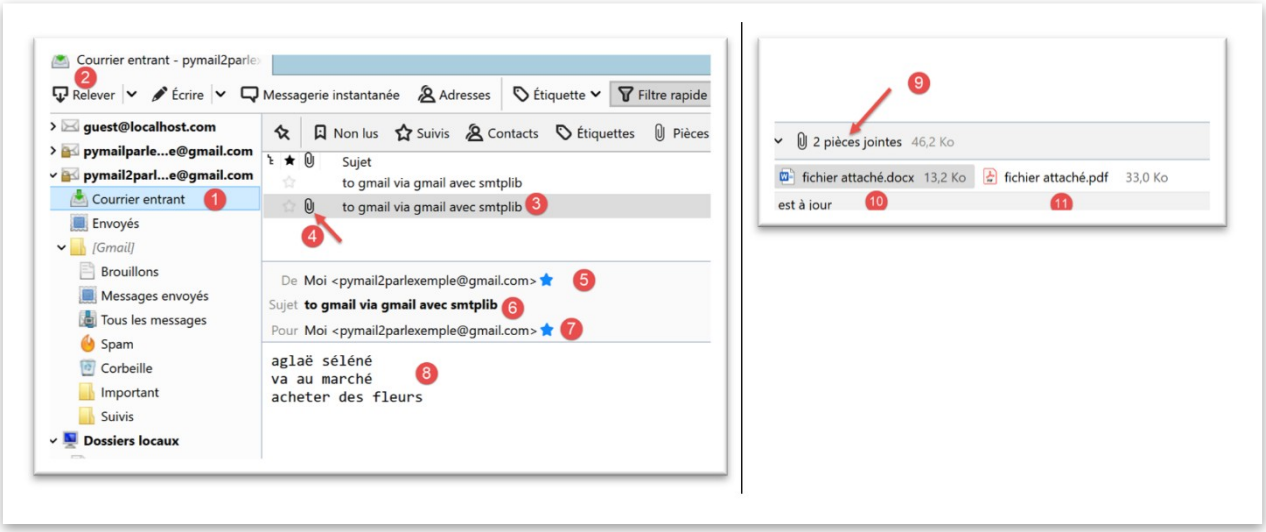
```
81.         msg.attach(part)
82.     # tous les attachements ont été faits - on envoie le message en tant que chaîne de caractères
83.     server.send_message(msg)
84.
85.
86. # main ----------------------------------------------------------------
87.
88. ..
```

Commentaires

- lignes 18-32 : la fonction **[sendmail]** reste ce qu'elle était lorsqu'il n'y avait pas d'attachements ;
- ligne 35 : le code qui suit est tiré d'une documentation officielle de Python ;
- ligne 36 : le message qui va être envoyé va comprendre plusieurs parties : du texte et des fichiers attachés. On appelle cela un message **[Multipart]** ;
- lignes 37-40 : on trouve dans le message **[Multipart]** les champs habituels de tout mail ;
- ligne 42 : les différentes parties du message **[Multipart] [msg]** sont attachées au message par la méthode **[msg.attach]** (ligne 81). Les parties attachées peuvent être de toute nature. Celles-ci sont caractérisées par un type MIME. Le type MIME d'un texte ordinaire est le type **[MIMEText]** ;
- lignes 44-81 : on va attacher au message **[msg Multipart]** tous les attachements du message à envoyer (ligne 81) ;
- ligne 44 : **[path]** représente le chemin absolu du fichier à attacher ;
- ligne 47 : pour trouver le type MIME à utiliser pour la partie à attacher, on va utiliser le suffixe (.docx, .php…) du fichier à attacher. La méthode **[mimetypes.guess_type]** fait ce travail. Elle rend deux informations :
  - **[ctype]** : le type MIME du fichier ;
  - **[encoding]** : une information sur son encodage ;
- lignes 49-52 : au cas où on ne peut pas déterminer le type MIME du fichier, on dit que c'est un fichier binaire (ligne 52) ;
- ligne 54 : le type MIME d'un fichier se décompose en type principal / type secondaire, par exemple **[application/pdf]**. On sépare ces deux éléments ;
- lignes 56-76 : on traite différents cas selon la valeur du type MIME principal. Par exemple, dans le cas **[application/pdf]** d'un fichier PDF, on va exécuter les lignes 70-76 :
  - lignes 56-59 : le cas où le fichier attaché est un fichier texte. Dans ce cas on crée un élément de type **[MIMEText]** de contenu **[fp.read]** ;
  - lignes 60-62 : le cas où le fichier contient une image. Dans ce cas on crée un élément de type **[MIMEImage]** de contenu **[fp.read]** ;
  - lignes 63-65 : le cas où le fichier est un fichier audio. Dans ce cas on crée un élément de type **[MIMEAudio]** de contenu **[fp.read]** ;
  - lignes 66-69 : le cas où le fichier est un mail. Dans ce cas on crée un élément de type **[MIMEMessage]** (ligne 69) de contenu **[email.message_from_bytes(fp.read())]**. Contrairement aux cas précédents où le contenu de l'élément MIME était le contenu binaire du fichier associé, ici le contenu de l'élément MIMEMessage est de type **[email.message.Message]** ;
  - lignes 70-76 : les autres cas. Cela comprend par exemple les fichiers Word et PDF de notre exemple ;
- ligne 72 : le fichier à attacher est ouvert en mode binaire (rb=read binary) ;
- ligne 74 : **[fp.read]** lit la totalité du fichier binaire ;
- lignes 72-74 : la structure **[with open(…) as file]** fait deux choses :
  - elle ouvre le fichier et lui donne le descripteur **[file]** ;
  - elle assure qu'à la sortie du **[with]**, erreur ou pas, le descripteur **[file]** sera fermé. C'est donc une alternative à la structure **[try file=open(…)/ finally]** ;
- ligne 73 : on crée un nouvel élément **[part]** à incorporer au message Multipart. On utilise ici la classe **[MIMEBase]** et on passe au constructeur les éléments **[maintype, subtype]** déterminés ligne 54 ;
- ligne 74 : l'élement à incorporer dans le message Multipart doit avoir un contenu. Celui-ci peut être initialisé avec la méthode **[set_payload]** ;
- lignes 75-76 : les fichiers attachés doivent subir un encodage 7 bits. En effet, historiquement certains serveurs SMTP ne supportaient que des caractères codés sur 7 bits. Ici c'est le codage appelé 'Base64' qui est utilisé ;
- ligne 77 : à partir de cette ligne, le traitement est comment à tous les types MIME que nous avons créés aux lignes 56-76 **[MIMEMessage, MIMEImage, MIMEAudio, MIMEBase, MIMEText]** ;
- ligne 79 : l'élément à ajouter dans le message Multipart a un entête le décrivant. On indique ici que l'élément ajouté correspond à un fichier attaché. Le nom de ce fichier est le troisième paramètre passé à la méthode **[add_header]**. Le nom de ce fichier est souvent utilisé par les lecteurs de courrier pour enregistrer, sous ce nom, le fichier attaché dans le système de fichiers du lecteur. On a pour l'instant travaillé avec le nom absolu du fichie attaché. Ici on passe simplement son nom sans son chemin (ligne 78) ;
- ligne 81 : le binaire du fichier est incorporé dans le message **[msg Multipart]** ;
- ligne 83 : lorsque tous les parties du message ont été attachées au **[msg Multipart]**, celui-ci est envoyé ;

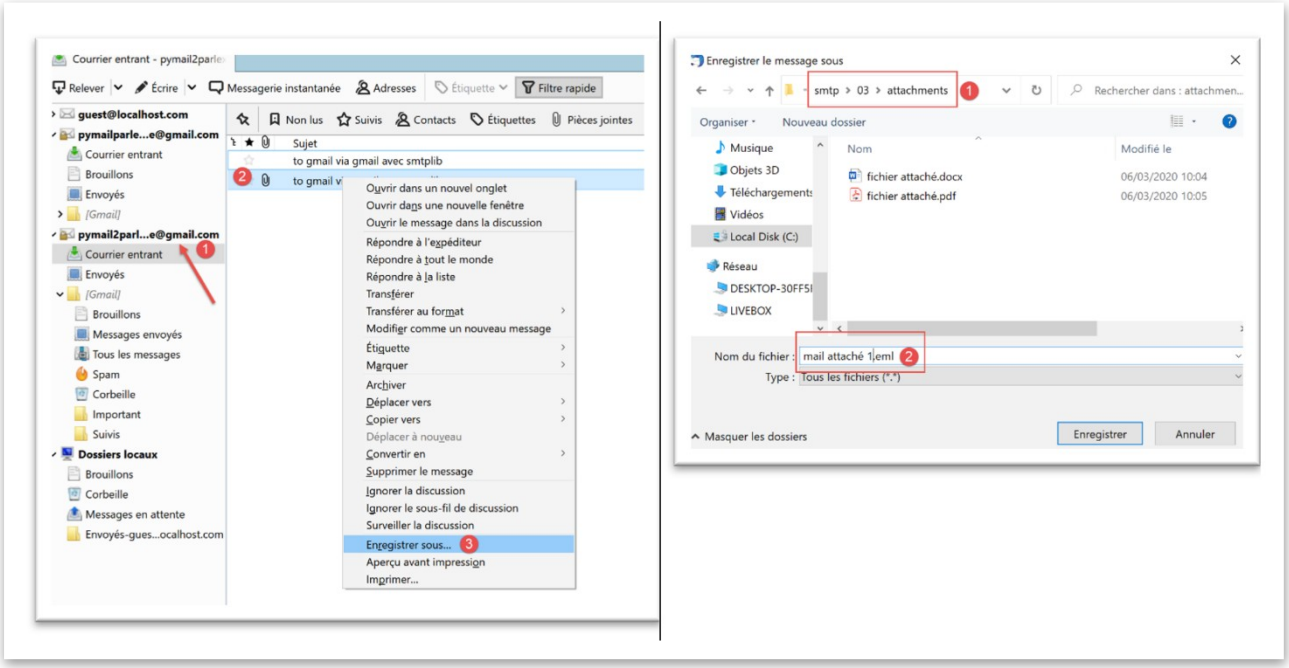
Résultats

Si on exécute le script **[smtp/03/main]** avec le fichier **[smtp/02/config]** déjà présenté, le compte **[pymail2parlexemple@gmail.com]** reçoit ceci :



On voit les fichiers attachés en **[4, 9-11]**.

Montrons un exemple maintenant avec un mail attaché. Nous allons sauvegarder le mail reçu en **[3]** ci-dessus :



Nous sauvegardons le mail sous le nom **[mail attaché 1.eml]** dans le dossier **[smtp/03/attachments]**.

Nous modifions maintenant le fichier **[smtp/03/config]** de la façon suivante :

```
import os

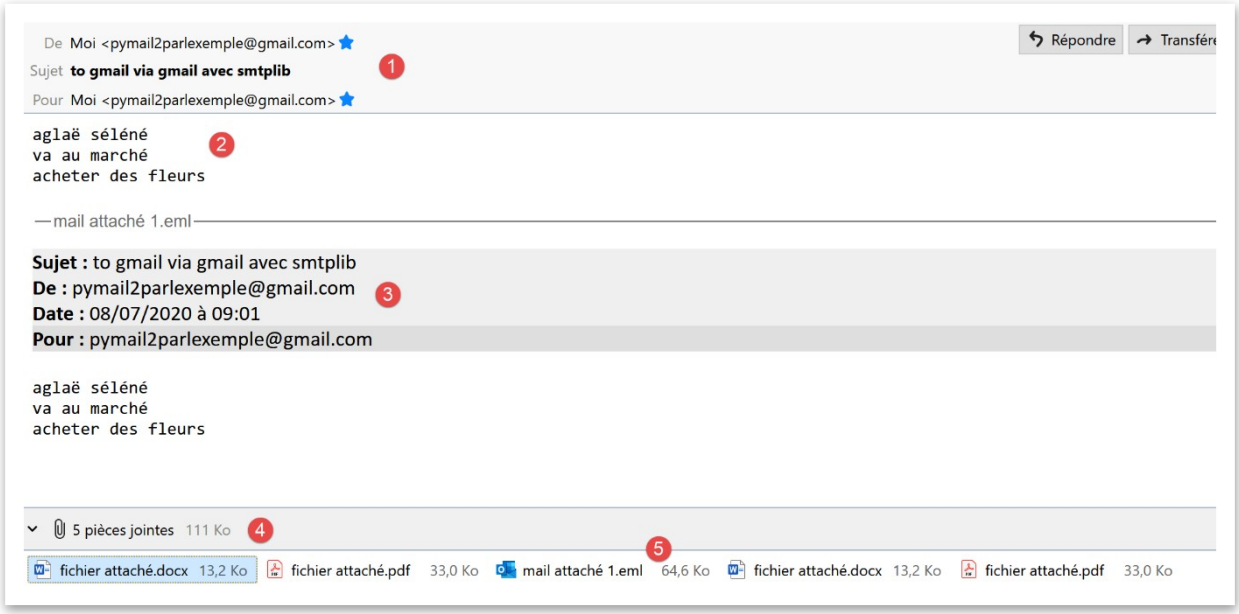

def configure() -> dict:
    # configuration de l'application
    script_dir = os.path.dirname(os.path.abspath(__file__))

    return {
        # description : description du mail envoyé
        # smtp-server : serveur SMTP
        # smtp-port : port du serveur SMTP
        # from : expéditeur
```

```
13.         # to : destinataire
14.         # subject : sujet du mail
15.         # message : message du mail
16.         "mails": [
17.             {
18.                 "description": "mail to gmail via gmail avec smtplib",
19.                 "smtp-server": "smtp.gmail.com",
20.                 "smtp-port": "587",
21.                 "from": "pymail2parlexemple@gmail.com",
22.                 "to": "pymail2parlexemple@gmail.com",
23.                 "subject": "to gmail via gmail avec smtplib",
24.                 # on teste les caractères accentués
25.                 "message": "aglaë séléné\nva au marché\nacheter des fleurs",
26.                 # smtp avec authentification
27.                 "user": "pymail2parlexemple@gmail.com",
28.                 "password": "#6prIlhD&@1QZ3TG",
29.                 # ici, il faut mettre des chemins absolus pour les fichiers attachés
30.                 "attachments": [
31.                     f"{script_dir}/attachments/fichier attaché.docx",
32.                     f"{script_dir}/attachments/fichier attaché.pdf",
33.                     f"{script_dir}/attachments/mail attaché 1.eml",
34.                 ]
35.             }
36.         ]
37.     }
```

- ligne 33, nous avons ajouté un attachement ;

Maintenant nous exécutons de nouveau le script **[smtp/03/main]**. Cela donne le résultat suivant dans la boîte à lettres de l'utilisateur **[pymail2parlexemple@gmail.com]** :



- en **[1]**, le mail reçu ;
- en **[2]** : le texte du message ;
- en **[3]** : le texte du mail attaché ;
- en **[4]** : Thunderbird a trouvé 5 pièces jointes :
  - **[fichier attaché.docx]** ;
  - **[fichier attaché.pdf]** ;
  - **[mail attaché 1.eml]**. Cette pièce jointe est elle-même un mail contenant deux pièces jointes :
    - **[fichier attaché.docx]** ;
    - **[fichier attaché.pdf]** ;

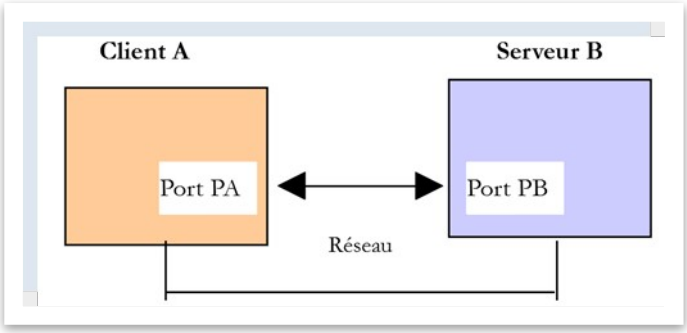
21.6 Le protocole POP3

21.6.1 Introduction

Pour lire les mails entreposés dans un serveur de mails, deux protocoles existent :

- le protocole POP3 (Post Office Protocol) historiquement le 1er protocole mais peu utilisé maintenant ;
- le protocole IMAP (Internet Message Access Protocol) protocole plus récent que POP3 et le plus utilisé actuellement ;

Pour découvrir le protocole POP3, nous allons utiliser l'architecture suivante :

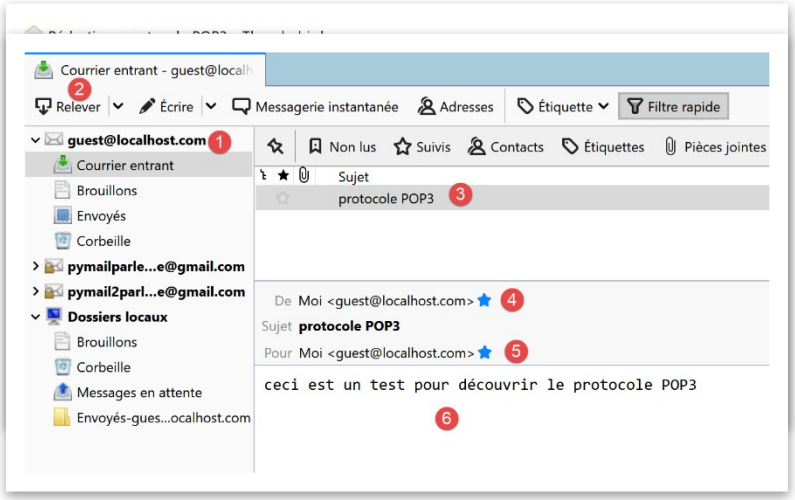


- **[Serveur B]** sera selon les cas :
  - un serveur POP3 local, implémenté par le serveur de mail **[hMailServer]** ;
  - le serveur **[pop.gmail.com]** qui est le serveur POP3 du gestionnaire de mails **[gmail.com]** ;
- **[Client A]** sera un client POP3 de diverses formes :
  - le client **[RawTcpClient]** pour découvrir le protocole POP3 ;
  - un script Python rejouant le protocole POP3 du client **[RawTcpClient]** ;
  - un script Python utilisant des modules Python permettant de gérer les pièces attachées ainsi que l'utilisation d'une connexion chiffrée et authentifiée lorsque le serveur POP3 l'exige ;

21.6.2 Découverte du protocole POP3

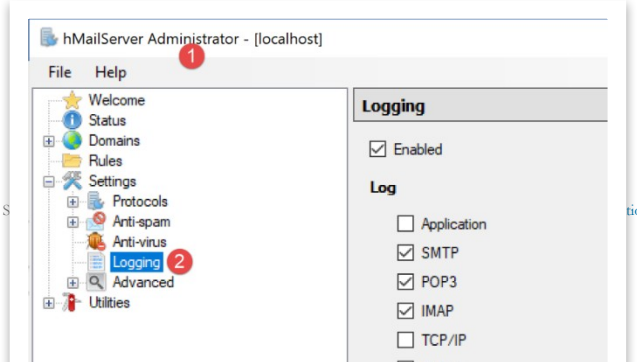
Comme nous l'avons fait avec le protocole SMTP, nous allons découvrir le protocole POP3 à l'aide des logs du serveur de mails **[hMailServer]**. Il faut ici lancer ce serveur.

Avec Thunderbird, nous allons :
- envoyer un mail à l'utilisateur **[guest@localhost.com]** ;
- lire la boîte à lettre de cet utilisateur ;



En **[3-6]** ci-dessus, le message reçu par l'utilisateur **[guest@localhost.com]**.

Nous examinons maintenant les logs du serveur **[hMailServer]**. Pour cela nous utilisons l'outil d'administration **[hMailServer Administrator]** :

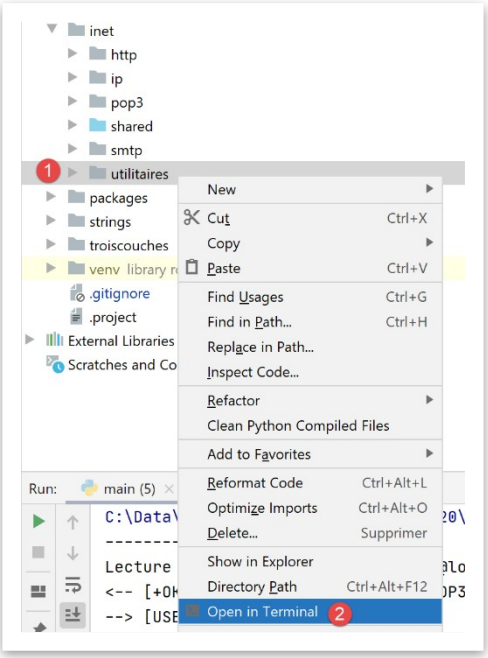


Les logs POP3 sont les suivants (les dernières lignes dans le fichier de logs du jour) :

```
1.  "POP3D" 35084   5       "2020-07-08 14:19:46.392"       "127.0.0.1"     "SENT: +OK Bienvenue sur le
    serveur POP3 localhost.com"
2.  "POP3D" 34968   5       "2020-07-08 14:19:46.405"       "127.0.0.1"     "RECEIVED: CAPA"
3.  "POP3D" 34968   5       "2020-07-08 14:19:46.407"       "127.0.0.1"     "SENT: +OK CAPA list
    follows[nl]USER[nl]UIDL[nl]TOP[nl]."
4.  "POP3D" 35076   5       "2020-07-08 14:19:46.410"       "127.0.0.1"     "RECEIVED: USER guest"
5.  "POP3D" 35076   5       "2020-07-08 14:19:46.411"       "127.0.0.1"     "SENT: +OK Send your password"
6.  "POP3D" 34968   5       "2020-07-08 14:19:46.418"       "127.0.0.1"     "RECEIVED: PASS ***"
7.  "POP3D" 34968   5       "2020-07-08 14:19:46.421"       "127.0.0.1"     "SENT: +OK Mailbox locked and
    ready"
8.  "POP3D" 34968   5       "2020-07-08 14:19:46.423"       "127.0.0.1"     "RECEIVED: STAT"
9.  "POP3D" 34968   5       "2020-07-08 14:19:46.423"       "127.0.0.1"     "SENT: +OK 1 612"
10. "POP3D" 34968   5       "2020-07-08 14:19:46.426"       "127.0.0.1"     "RECEIVED: LIST"
11. "POP3D" 34968   5       "2020-07-08 14:19:46.426"       "127.0.0.1"     "SENT: +OK 1 messages (612
    octets)"
12. "POP3D" 34968   5       "2020-07-08 14:19:46.426"       "127.0.0.1"     "SENT: 1 612[nl]."
13. "POP3D" 35076   5       "2020-07-08 14:19:46.427"       "127.0.0.1"     "RECEIVED: UIDL"
14. "POP3D" 35076   5       "2020-07-08 14:19:46.428"       "127.0.0.1"     "SENT: +OK 1 messages (612 octets)
    [nl]1 42[nl]."
15. "POP3D" 34968   5       "2020-07-08 14:19:46.435"       "127.0.0.1"     "RECEIVED: RETR 1"
16. "POP3D" 34968   5       "2020-07-08 14:19:46.436"       "127.0.0.1"     "SENT: ."
17. "POP3D" 34924   5       "2020-07-08 14:19:46.459"       "127.0.0.1"     "RECEIVED: QUIT"
18. "POP3D" 34924   5       "2020-07-08 14:19:46.459"       "127.0.0.1"     "SENT: +OK POP3 server saying
    goodbye..."
```

- ligne 1 : le serveur POP3 envoie un message de bienvenue au client (Thunderbird) qui vient de se connecter ;
- ligne 2 : le client envoie la commande **[CAPA]** (capabilities) pour demander la listes des commandes qu'il peut utiliser ;
- ligne 3 : le serveur lui répond qu'il peut utiliser les commandes **[USER, UIDL, TOP]**. Le serveur POP commence ses réponses par **[+OK]** ou **[-ERR]** pour indiquer qu'il a réussi ou échoué à exécuter la commande du client ;
- ligne 4 : le client envoie la commande **[USER guest]** pour indiquer qu'il veut consulter la boîte à lettres de l'utilisateur **[guest]** ;
- ligne 5 : le serveur lui répond **[+OK]** et demande le mote de passe de **[guest]** ;
- ligne 6 : le client envoie la commande **[PASS password]** pour envoyer le mot de passe de l'utilisateur **[guest]**. Ici le mot de passe est en clair car le serveur POP3 n'a pas imposé de connexion sécurisée. Nous verrons que ce sera différent avec le serveur POP3 de Gmail ;
- ligne 7 : le serveur a validé l'ensemble login / mot de passe. Il indique qu'il bloque la boîte à lettres de l'utilisateur **[guest]** ;
- ligne 8 : le client lui envoie la commande **[STAT]** qui demande des informations sur la boîte à lettres ;
- ligne 9 : le serveur lui répond qu'il y a un message de 612 octets. De façon générale, il répond qu'il y a N messages et donne la taille totale de ces messages ;
- ligne 10 : le client envoie la commande **[LIST]**. Cette commande demande la liste des messages ;
- ligne 11 : le serveur lui envoie la liste des messages sous la forme suivante :
  - une ligne récapitulative avec le nombre de messages et leur taille totale ;
  - une ligne par message indiquant le n° du message et sa taille ;
- ligne 13 : le client envoie la commande **[UIDL]** qui demande la liste des messages avec leurs identifiants. En effet, chaque message est repéré par un n° unique au sein du service de mails ;
- ligne 14 : la réponse du serveur. On voit ainsi que le message n° 1 dans la liste a l'identifiant 42 ;
- ligne 15 : le client envoie la commande **[RETR 1]** qui demande à ce qu'on lui transfère le message n° 1 de la liste ;
- ligne 16 : le serveur POP3 le fait ;
- ligne 17 : le client envoie la commande **[QUIT]** pour indiquer qu'il va se déconnecter du serveur POP3 ;
- ligne 18 : le serveur va lui également fermer sa connexion avec le client mais auparavant il lui envoie un message d'au-revoir ;

Nous allons reproduire maintenant des éléments du dialogue ci-dessus en utilisant le client **[RawTcpClient]** exécuté dans une fenêtre PyCharm :



Le dialogue est le suivant :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>RawTcpClient.exe localhost
    110
2.  Client [DESKTOP-30FF5FB:63762] connecté au serveur [localhost-110]
3.  Tapez vos commandes (quit pour arrêter) :
4.  <-- [+OK Bienvenue sur le serveur POP3 localhost.com]
5.  USER guest
6.  <-- [+OK Send your password]
7.  PASS guest
8.  <-- [+OK Mailbox locked and ready]
9.  LIST
10. <-- [+OK 1 messages (612 octets)]
11. <-- [1 612]
12. <-- [.]
13. RETR 1
14. <-- [+OK 612 octets]
15. <-- [Return-Path: guest@localhost.com]
16. <-- [Received: from [127.0.0.1] (DESKTOP-30FF5FB [127.0.0.1])]
17. <-- [   by DESKTOP-30FF5FB with ESMTP]
18. <-- [   ; Wed, 8 Jul 2020 14:19:36 +0200]
19. <-- [To: guest@localhost.com]
20. <-- [From: "guest@localhost.com" <guest@localhost.com>]
21. <-- [Subject: protocole POP3]
22. <-- [Message-ID: <ca895136-25c5-411e-373a-a68cbd0eca51@localhost.com>]
23. <-- [Date: Wed, 8 Jul 2020 14:19:33 +0200]
24. <-- [User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:68.0) Gecko/20100101]
25. <-- [ Thunderbird/68.10.0]
26. <-- [MIME-Version: 1.0]
27. <-- [Content-Type: text/plain; charset=utf-8; format=flowed]
28. <-- [Content-Transfer-Encoding: 8bit]
29. <-- [Content-Language: fr]
30. <-- []
31. <-- [ceci est un test pour découvrir le protocole POP3]
32. <-- []
33. <-- [.]
34. QUIT
35. Fin de la connexion avec le serveur
```

- ligne 1 : on ouvre une connexion avec le port 110 de la machine **[localhost]**. C'est là qu'opère le service POP3 de **[hMailServer]** ;
- aux lignes 5, 7, 9, 13, 34, nous utilisons les commandes **[USER, PASS, LIST, RETR, QUIT]** ;
- ligne 4 : le message de bienvenue du serveur POP3 ;
- ligne 5 : on indique qu'on veut accéder à la boîte à lettres de l'utilisateur **[guest]** ;
- ligne 7 : on envoie le mot de passe de l'utilisateur **[guest]** en clair ;
- ligne 9 : on demande la liste des messages de la boîte à lettres ;

- ligne 13 : on demande le message n° 1 ;
- lignes 14-33 : le serveur POP3 envoie le message n° 1 ;
- ligne 34 : on termine la session ;

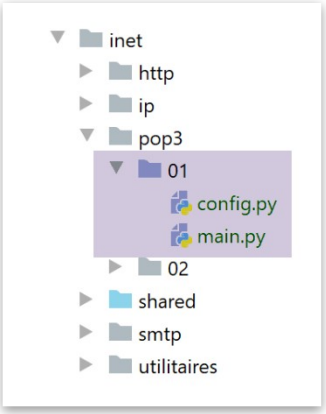
Voici un résumé de quelques commandes courantes acceptées par un serveur POP3 :

- la commande **[USER]** sert à définir l'utilisateur dont on veut lire la boîte mail ;
- la commande **[PASS]** sert à définir son mot de passe ;
- la commande **[LIST]** demande la liste des messages présents dans la boîte à lettres de l'utilisateur ;
- la commande **[RETR]** demande à voir le message dont on passe le n° ;
- la commande **[DELE]** demande la suppression du message dont on passe le n° ;
- la commande **[QUIT]** indique au serveur qu'on a terminé ;

La réponse du serveur peut prendre plusieurs formes :

- une ligne unique commençant par **[+OK]** pour indiquer que la commande précédente du client a réussi ;
- une ligne unique commençant par **[-ERR]** pour indiquer que la commande précédente du client a échoué ;
- plusieurs lignes où :
  - la 1re ligne commence par **[+OK]** ;
  - la dernière ligne est constituée d'un unique point ;

21.6.3 scripts [pop3/01] : un client POP3 basique



Comme le procole POP3 a la même structure que le protocole SMTP, le script **[pop3/01/main.py]** est un portage du script **[smtp/01/main.py]**. Il aura le fichier de configuration **[pop3/01/config.py]** suivant :

```
1.  def configure() -> dict:
2.      # les boîtes à lettres dont on relève les mails
3.      mailboxes = [
4.          # server : serveur POP3
5.          # port : port du serveur POP3
6.          # user : utilisateur dont on veut lire les messages
7.          # password : son mot de passe
8.          # maxmails : le nombre maximum de mails à télécharger
9.          # timeout : délai d'attente maximal d'une réponse du serveur
10.         # encoding : encodage des mails reçus
11.         # delete : si True, alors les mails sont supprimés de la boîte à lettres
12.         # une fois qu'ils ont été téléchargés localement
13.
14.         {
15.             "server": "localhost",
16.             "port": "110",
17.             "user": "guest",
18.             "password": "guest",
19.             "maxmails": 10,
20.             "timeout": 1.0,
21.             "encoding": "utf-8",
22.             "delete": False
23.         }
24.     ]
25.     # on rend la configuration
26.     return {
27.         "mailboxes": mailboxes
```

```
28.     }
```

- lignes 3-24 : la liste des boîtes à lettres à consulter. Ici il n'y en a qu'une ;
- lignes 4-12 : significations des éléments du dictionnaire définissant chacune des boîtes à lettres ;
- ligne 15 : le serveur POP3 interrogé est le serveur local **[hMailServer]** ;
- lignes 17-18 : on veut lire la boîte à lettres de l'utilisateur **[guest@localhost]** ;
- ligne 19 : on lira au plus 10 mails ;
- ligne 20 : le client aura un délai d'attente d'une réponse du serveur d'au plus 1 seconde ;
- ligne 21 : le type d'encodage des messages lus ;
- ligne 22 : on ne supprimera pas les messages téléchargés ;

Le script **[pop3/01/main.py]** est le suivant :

```
1.  # imports
2.  import re
3.  import socket
4.
5.
6.  # ------------------------------------------------------------------------
7.  def readmails(mailbox: dict, verbose: bool):
8.      # lit la boîte mail décrite par le dictionnaire [mailbox]
9.      # si verbose=True, fait un suivi des échanges client-serveur
10. …
11.
12.
13. # ------------------------------------------------------------------------
14. def send_command(mailbox: dict, connexion: socket, commande: str, verbose: bool, with_rclf: bool) -> str:
15.     # envoie commande dans le canal connexion
16.     # mode verbeux si verbose=True
17.     # si with_rclf=True, ajoute la séquence rclf à échange
18.     # rend la 1ère ligne de la réponse
19. …
20.
21.
22. # ------------------------------------------------------------------------
23. def affiche(echange: str, sens: int):
24.     …
25.
26.
27. # main ------------------------------------------------------------------
28.
29. # client POP3 (Post Office Protocol) permettant de lire des messages d'une boîte à lettres
30. # protocole de communication POP3 client-serveur
31. # -> client se connecte sur le port 110 du serveur smtp
32. # <- serveur lui envoie un message de bienvenue
33. # -> client envoie la commande USER utilisateur
34. # <- serveur répond OK ou non
35. # -> client envoie la commande PASS mot_de_passe
36. # <- serveur répond OK ou non
37. # -> client envoie la commande LIST
38. # <- serveur répond OK ou non
39. # -> client envoie la commande RETR n° pour chacun des mails
40. # <- serveur répond OK ou non. Si OK envoie le contenu du mail demandé
41. # -> serveur envoie ttes les lignes du mail et termine avec une ligne contenant le
42. # seul caractère .
43. # -> client envoie la commande DELE n° pour supprimer un mail
44. # <- serveur répond OK ou non
45. # # -> client envoie la commande QUIT pour terminer le dialogue avec le serveur
46. # <- serveur répond OK ou non
47. # les réponses du serveur ont la forme +OK texte où -ERR texte
48. # La réponse peut comporter plusieurs lignes. Alors la dernière est constituée d'un unique point
49. # les lignes de texte échangées doivent se terminer par les caractères RC(#13) et LF(#10)
50. #
51.
52. # on récupère la configuration de l'application
53. import config
54. config = config.configure()
55.
56. # on traite les boîtes mail une par une
57. for mailbox in config['mailboxes']:
58.     try:
59.         # affichage console
60.         print("---------------------------------")
61.         print(
62.             f"Lecture de la boîte mail POP3 {mailbox['user']}@{mailbox['server']}:{mailbox['port']}")
```

```
63.         # lecture de la boîte mail en mode verbeux
64.         readmails(mailbox, True)
65.         # fin
66.         print("Lecture terminée...")
67.     except BaseException as erreur:
68.         # on affiche l'erreur
69.         print(f"L'erreur suivante s'est produite : {erreur}")
70.     finally:
71.         pass
```

Commentaires

Comme nous l'avons dit, **[pop3/01/main.py]** est un portage du script **[smtp/01/main.py]** que nous avons déjà commenté. Nous ne commenterons que les principales différences :

- ligne 64 : la fonction **[readmails]** est chargée de lire les mails d'une boîte aux lettres. Les informations pour se connecter à cette boîte à lettres sont dans le dictionnaire **[mailbox]**. Le second paramètre **[True]** est le paramètre **[Verbose]** qui demande ici un suivi des échanges client / serveur ;

La fonction **[readmails]** est la suivante :

```
1.  # ---------------------------------------------------------------------
2.  def readmails(mailbox: dict, verbose: bool):
3.      # lit les mails de la boîte mail décrite par le dictionnaire [mailbox]
4.      # si verbose=True, fait un suivi des échanges client-serveur
5.
6.      # on isole les paramètres de la boîte mail
7.      # on suppose que le dictionnaire [mailbox] est valide
8.      server = mailbox['server']
9.      port = int(mailbox['port'])
10.     user = mailbox['user']
11.     password = mailbox['password']
12.     maxmails = mailbox['maxmails']
13.     delete = mailbox['delete']
14.     timeout = mailbox['timeout']
15.
16.     # on laisse remonter les erreurs système
17.     connexion = None
18.     try:
19.         # ouverture d'une connexion sur le port [port] de [server] avec un timeout d'une seconde
20.         connexion = socket.create_connection((server, port), timeout=timeout)
21.
22.         # connexion représente un flux de communication bidirectionnel
23.         # entre le client (ce programme) et le serveur pop3 contacté
24.         # ce canal est utilisé pour les échanges de commandes et d'informations
25.
26.         # lecture msg de bienvenue
27.         send_command(mailbox, connexion, "", verbose, True)
28.         # cmde USER
29.         send_command(mailbox, connexion, f"USER {user}", verbose, True)
30.         # cmde PASS
31.         send_command(mailbox, connexion, f"PASS {password}", verbose, True)
32.         # cmde LIST
33.         première_ligne = send_command(mailbox, connexion, "LIST", verbose, True)
34.         # analyse de la 1ère ligne pour connaître le nbre de messages
35.         match = re.match(r"^\+OK (\d+)", première_ligne)
36.         nbmessages = int(match.groups()[0])
37.         # on boucle sur les messages
38.         imessage = 0
39.         while imessage < nbmessages and imessage < maxmails:
40.             # cmde RETR
41.             send_command(mailbox, connexion, f"RETR {imessage + 1}", verbose, True)
42.             # cmde DELE
43.             if delete:
44.                 send_command(mailbox, connexion, f"DELE {imessage + 1}", verbose, True)
45.             # msg suivant
46.             imessage += 1
47.         # cmde QUIT
48.         send_command(mailbox, connexion, "QUIT", verbose, True)
49.         # fin
50.     finally:
51.         # fermeture connexion
52.         if connexion:
53.             connexion.close()
```

Commentaires

- lignes 8-14 : on récupère les informations de configuration de la boîte à lettres à consulter ;
- lignes 19-20 : ouverture d'une connexion avec le serveur POP3 ;
- lignes 26-27 : lecture du message de bienvenue envoyé par le serveur ;
- lignes 28-29 : on envoie la commande **[USER]** pour identifier l'utilisateur dont on veut les mails ;
- lignes 30-31 : on envoie la commande **[PASS]** pour donner le mot de passe de cet utilisateur ;
- lignes 32-33 : on envoie la commande **[LIST]** pour savoir combien il y a de mails dans la boîte à lettres de cet utilisateur. La fonction **[sendCommand]** renvoie la première ligne de la réponse du serveur. Dans celle-ci le serveur indique combien il y a de messages dans la boîte à lettres ;
- lignes 34-36 : on récupère le nombre de messages dans la 1re ligne de la réponse ;
- lignes 39-46 : on boucle sur chacun des messages. Pour chacun d'eux on émet deux commandes :
  - RETR i : pour récupérer le message n° i (lignes 40-41) ;
  - DELE i : pour le supprimer si la configuration demande que les messages lus soient supprimés du serveur (lignes 43-44) ;
- lignes 47-48 : on envoie la commande **[QUIT]** pour dire au serveur qu'on a terminé ;

La fonction **[send_command]** est la suivante :

```
1.  # ---------------------------------------------------------------------------
2.  def send_command(mailbox: dict, connexion: socket, commande: str, verbose: bool, with_rclf: bool) -> str:
3.      # envoie commande dans le canal connexion
4.      # mode verbeux si verbose=True
5.      # si with_rclf=True, ajoute la séquence rclf à échange
6.      # rend la 1ère ligne de la réponse
7.
8.      # marque de fin de ligne
9.      if with_rclf:
10.         rclf = "\r\n"
11.     else:
12.         rclf = ""
13.     # envoi commande si non vide
14.     if commande:
15.         connexion.send(bytearray(f"{commande}{rclf}", 'utf-8'))
16.         # écho éventuel
17.         if verbose:
18.             affiche(commande, 1)
19.     # lecture de la socket comme si elle était un fichier texte
20.     encoding = f"{mailbox['encoding']}" if mailbox['encoding'] else None
21.     file = connexion.makefile(encoding=encoding)
22.     # on exploite ce fichier ligne par ligne
23.     # lecture 1ère ligne
24.     première_ligne = réponse = file.readline().strip()
25.     # mode verbeux ?
26.     if verbose:
27.         affiche(première_ligne, 2)
28.     # récupération code erreur
29.     code_erreur = réponse[0]
30.     if code_erreur == "-":
31.         # il y a eu une erreur
32.         raise BaseException(réponse[5:])
33.     # cas particulier des réponses à plusieurs lignes LIST, RETR
34.     cmd = commande.lower()[0:4]
35.     if cmd == "list" or cmd == "retr":
36.         # dernière ligne de la réponse ?
37.         dernière_ligne = False
38.         while not dernière_ligne:
39.             # lecture ligne suivante
40.             ligne_suivante = file.readline().strip()
41.             # mode verbeux ?
42.             if verbose:
43.                 affiche(ligne_suivante, 2)
44.             # dernière ligne ?
45.             dernière_ligne = ligne_suivante == "."
46.     # fini - on rend la 1ère ligne
47.     return première_ligne
```

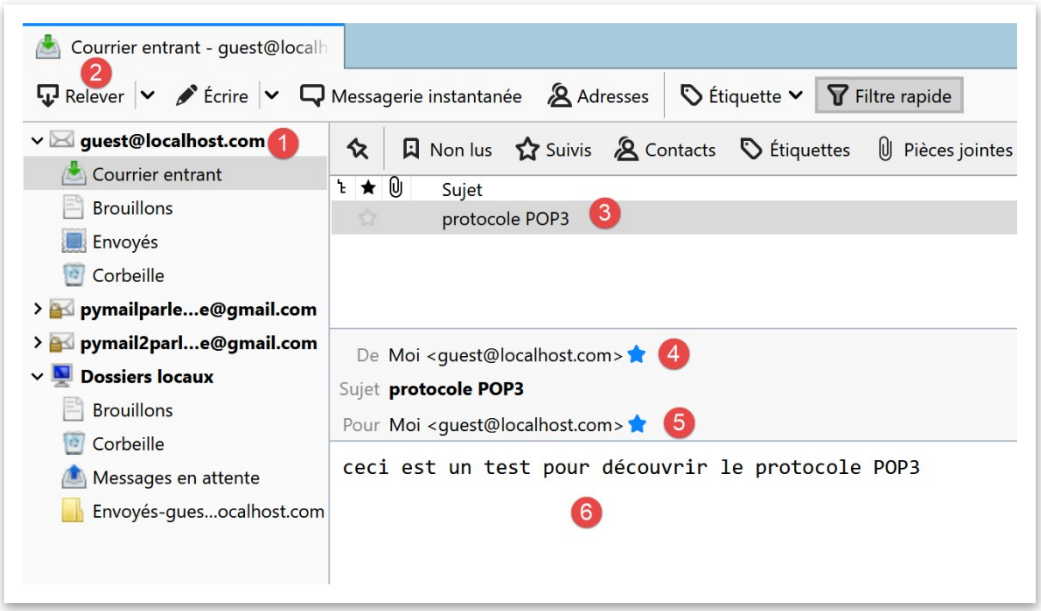
Commentaires

- lignes 13-18 : la commande **[command]** n'est envoyée au serveur POP3 que si elle est non vide. Ce cas est nécessaire pour lire le message de bienvenue du serveur POP3 qu'il envoie alors même que le client n'a pas encore envoyé de commandes ;

- lignes 19-21 : on lit la socket comme si elle était un fichier texte. Cela va nous permettre d'utiliser la méthode **[readline]** (ligne 24) et de lire ainsi le message ligne par ligne. On utilise la clé **[encoding]** du dictionnaire **[mailbox]** pour indiquer le codage des lignes qui vont être lues ;
- ligne 24 : on lit la 1re ligne de la réponse ;
- lignes 28-32 : on gère le cas d'une éventuelle erreur. Celles-ci sont du type **[-ERR invalid password, -ERR mailbox unknown, -ERR unable to lock mailbox…]** ;
- ligne 32 : on lance une exception avec le message de l'erreur ;
- ligne 35 : seules les commandes **[list, retr]** peuvent avoir des réponses à plusieurs lignes ;
- lignes 36-45 : dans le cas d'une réponse à plusieurs lignes, on affiche toutes les lignes reçues (lignes 42-43) jusqu'à recevoir la dernière ligne (ligne 45) ;
- ligne 46 : on rend la 1re ligne lue car dans le cas de la commande **[LIST]**, elle comporte le nombre de messages présents dans la boîte à lettres ;

Résultats

Prenons l'exemple précédent. Avec Thunderbird, nous avions envoyé le message suivant à l'utilisateur **[guest@localhost]** (il faut que le serveur hMailServer soit lancé) :



A l'exécution, on obtient les résultats suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/inet/pop3/01/main.py
2.  ----------------------------------
3.  Lecture de la boîte mail POP3 guest@localhost:110
4.  <-- [+OK Bienvenue sur le serveur POP3 localhost.com]
5.  --> [USER guest]
6.  <-- [+OK Send your password]
7.  --> [PASS guest]
8.  <-- [+OK Mailbox locked and ready]
9.  --> [LIST]
10. <-- [+OK 1 messages (612 octets)]
11. <-- [1 612]
12. <-- [.]
13. --> [RETR 1]
14. <-- [+OK 612 octets]
15. <-- [Return-Path: guest@localhost.com]
16. <-- [Received: from [127.0.0.1] (DESKTOP-30FF5FB [127.0.0.1])]
17. <-- [by DESKTOP-30FF5FB with ESMTP]
18. <-- [; Wed, 8 Jul 2020 14:19:36 +0200]
19. <-- [To: guest@localhost.com]
20. <-- [From: "guest@localhost.com" <guest@localhost.com>]
21. <-- [Subject: protocole POP3]
22. <-- [Message-ID: <ca895136-25c5-411e-373a-a68cbd0eca51@localhost.com>]
23. <-- [Date: Wed, 8 Jul 2020 14:19:33 +0200]
24. <-- [User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:68.0) Gecko/20100101]
25. <-- [Thunderbird/68.10.0]
26. <-- [MIME-Version: 1.0]
```

```
27. <-- [Content-Type: text/plain; charset=utf-8; format=flowed]
28. <-- [Content-Transfer-Encoding: 8bit]
29. <-- [Content-Language: fr]
30. <-- []
31. <-- [ceci est un test pour découvrir le protocole POP3]
32. <-- []
33. <-- [.]
34. --> [QUIT]
35. <-- [+OK POP3 server saying goodbye...]
36. Lecture terminée...
37.
38. Process finished with exit code 0
```

- lignes 15-31 : on récupère correctement le message envoyé à **[guest@localhost]**.

Nous avons là un client POP3 basique auquel il manque certaines capacités :

1. la possibilité de dialoguer avec un serveur POP3 sécurisé ;
2. la possibilité de lire les pièces attachées à un message ;

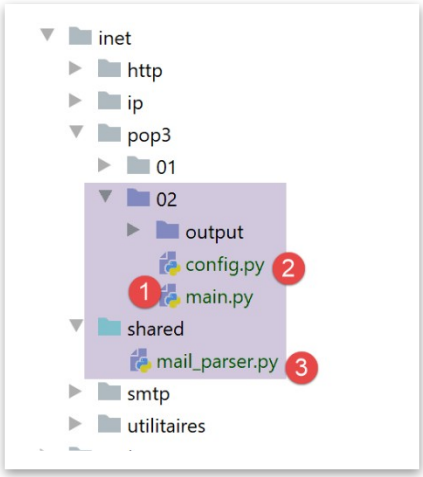
Nous allons implémenter ces deux possibilités avec un nouveau script qui sera cette fois plus complexe.

21.6.4 scripts [pop3/02] : client POP3 avec les modules [poplib] et [email]

Nous allons écrire un client POP3 permettant de gérer les pièces attachées ainsi que la communication avec des serveurs sécurisés. Par ailleurs, nous sauvegarderons dans des fichiers, les messages et leurs pièces attachées.

Nous allons utiliser deux modules Python :

- **[poplib]** : qui va assurer le protocole POP3 ;
- **[email]** : qui regroupe de nombreux sous-modules qui vont nous permettre d'analyser les messages reçus. Chaque message est une chaîne de caractères structurée dans laquelle on peut retrouver :
  - les entêtes du message **[From, To, Subject, Return-Path…]** ;
  - le message dans ses versions texte et éventuellement HTML ;
  - les pièces attachées ;



Le script **[inet/pop3/02/main] [1]** est configuré par le fichier **[inet/pop3/02/config] [2]** et utilise le module **[inet/shared/mail_parser] [3]**.

Le fichier **[pop3/02/config]** est le suivant :

```
1.  import os
2.
3.
4.  def configure() -> dict:
5.      # configuration de l'appli
6.      config = {
7.          # liste des boîtes à lettres à gérer
8.          "mailboxes": [
9.              # server : serveur POP3
```

```
            # port : port du serveur POP3
            # user : utilisateur dont on veut lire les messages
            # password : son mot de passe
            # maxmails : le nombre maximum de mail à télécharger
            # timeout : délai d'attente maximal d'une réponse du serveur
            # delete : à vrai s'il faut supprimer du serveur les messages téléchargés
            # ssl : à vrai si la lecture des mails se fait au travers d'une liaison sécurisée
            # output : le dossier de rangement des messages téléchargés

            {
                "server": "pop.gmail.com",
                "port": "995",
                "user": "pymail2parlexemple@gmail.com",
                "password": "#6prIlhD&@1QZ3TG",
                "maxmails": 10,
                "delete": False,
                "ssl": True,
                "timeout": 2.0,
                "output": "output"
            }
        ]
    }
    # chemin absolu du dossier du script
    script_dir = os.path.dirname(os.path.abspath(__file__))

    # chemins absolus des dossiers à inclure dans le syspath
    absolute_dependencies = [
        # dossier local
        f"{script_dir}/../../shared",
    ]

    # configuration du syspath
    from myutils import set_syspath
    set_syspath(absolute_dependencies)

    # on rend la configuration
    return config
```

Le fichier définit la liste des boîtes à lettres à consulter et fixe le Python Path de l'application.

Il n'y a ici qu'une unique boîte à lettres :

- lignes 22-23 : l'utilisateur dont on veut lire les mails ;
- lignes 20-21 : le nom et le port du serveur POP3 qui stocke les mails de cet utilisateur ;
- ligne 24 : le nombre maximum de mails à récupérer. En effet, si vous essayez ce script sur votre propre boîte mail, vous ne voudrez sans doute pas récupérer les centaines de mails qui s'y trouvent ;
- ligne 25 : booléen qui indique si après la lecture d'un mail, on doit supprimer celui-ci (delete=True) ;
- ligne 26 : l'attribut **[ssl]** à True signifie que le serveur POP3 défini aux lignes 20-21 utilise une connexion cryptée ;
- ligne 27 : le temps d'attente maximal des réponses du serveur exprimé en secondes ;
- ligne 28 : le dossier dans lequel ranger les mails lus. Il sera créé s'il n'existe pas. On a ici un nom relatif. A l'exécution, il sera relatif au dossier à partir duquel vous lancez le script. Avec **[Pycharm]**, ce dossier sera celui du script **[pop3/02]** ;

Le script **[pop3/02/main]** est le suivant :

```
# imports
import email
import os
import poplib
import shutil


# lecture d'une boîte mail
def readmails(mailbox: dict, verbose: bool):
    # lit la boîte mail décrite par le dictionnaire [mailbox]
    # si verbose=True, fait un suivi des échanges client-serveur
…

# main ----------------------------------------------------------------
#  client POP3 (Post Office Protocol) permettant de lire des mails

# on récupère la configuration de l'application
import config
config = config.configure()

```

```
21. # on traite les boîtes mail une par une
22. for mailbox in config['mailboxes']:
23.     try:
24.         # affichage console
25.         print("----------------------------------")
26.         print(
27.             f"Lecture de la boîte mail POP3 {mailbox['user']}@{mailbox['server']}:{mailbox['port']}")
28.         # lecture de la boîte mail en mode verbeux
29.         readmails(mailbox, True)
30.         # fin
31.         print("Lecture terminée...")
32.     except BaseException as erreur:
33.         # on affiche l'erreur
34.         print(f"L'erreur suivante s'est produite : {erreur}")
35.     finally:
36.         pass
```

- lignes 17-36 : la partie **[main]** du script est analogue à celle du script **[pop3/01]** ;

La fonction **[readmails]** est la suivante :

```
1.  # lecture d'une boîte mail
2.  def readmails(mailbox: dict, verbose: bool):
3.      # lit la boîte mail décrite par le dictionnaire [mailbox]
4.      # si verbose=True, fait un suivi des échanges client-serveur
5.
6.      # import de mail_parser
7.      from mail_parser import save_message
8.
9.      # on isole les paramètres de la boîte mail
10.     # on suppose que le dictionnaire [mailbox] est valide
11.     server = mailbox['server']
12.     port = int(mailbox['port'])
13.     user = mailbox['user']
14.     password = mailbox['password']
15.     maxmails = mailbox['maxmails']
16.     ssl = mailbox['ssl']
17.     timeout = mailbox['timeout']
18.     output = mailbox['output']
19.
20.     # on laisse remonter les erreurs système
21.     pop3 = None
22.     try:
23.         # on crée les dossiers de stockage s'ils n'existent pas
24.         if not os.path.isdir(output):
25.             os.mkdir(output)
26.         # user
27.         dir2 = f"{output}/{user}"
28.         # on supprime le dossier [dir2] s'il existe puis on le recrée
29.         if os.path.isdir(dir2):
30.             # suppression
31.             shutil.rmtree(dir2)
32.         # création
33.         os.mkdir(dir2)
34.         # ouverture d'une connexion sur le port [port] de [server]
35.         if ssl:
36.             pop3 = poplib.POP3_SSL(server, port, timeout=timeout)
37.         else:
38.             pop3 = poplib.POP3(server, port, timeout=timeout)
39.
40.         # connexion représente un flux de communication bidirectionnel
41.         # entre le client (ce programme) et le serveur pop3 contacté
42.         # ce canal est utilisé pour les échanges de commandes et d'informations
43.
44.         # mode verbose
45.         pop3.set_debuglevel(2 if verbose else 0)
46.         # lecture msg de bienvenue
47.         pop3.getwelcome( )
48.         # cmde USER
49.         réponse = pop3.user(user)
50.         # cmde PASS
51.         réponse = pop3.pass_(password)
52.         # cmde LIST
53.         liste = pop3.list()
54.         # les mails sont dans liste[1]
55.         imail = 0
```

```
56.         nb_mails = len(liste[1])
57.         fini = imail == maxmails or imail == nb_mails
58.         éléments = liste[1]
59.         while not fini:
60.             # élément courant
61.             élément = éléments[imail]
62.             # élément est une liste d'octets qu'on décode en string
63.             desc = élément.decode()
64.             # on a une chaîne séparée par des blancs
65.             # le 1er élément est le n° du message
66.             num = desc.split()[0]
67.             # on récupère le message
68.             message = pop3.retr(int(num))
69.             # les lignes du message sont dans message [1]
70.             str_message = ""
71.             for ligne in message[1]:
72.                 # ligne est une suite d'octets qu'on décode en string
73.                 str_message += f"{ligne.decode()}\r\n"
74.             # dossier du message
75.             dir3 = f"{dir2}/message_{num}"
76.             # si le dossier n'existe pas, on le crée
77.             if not os.path.isdir(dir3):
78.                 os.mkdir(dir3)
79.             # objet email.message.Message
80.             save_message(dir3, email.message_from_string(str_message), 0)
81.             # un mail de +
82.             imail += 1
83.             # a-t-on atteint le max ?
84.             fini = imail == maxmails or imail == nb_mails
85.
86.         # cmde QUIT
87.         pop3.quit()
88.     finally:
89.         # fermeture connexion
90.         if pop3:
91.             pop3.close()
```

Commentaires

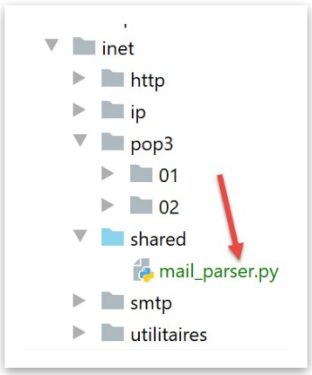
- lignes 6-7 : on importe la fonction **[mail_parser.save_message]** utilisée ligne 80 ;
- le code de la fonction est encapsulée dans un try (ligne 22)/ finally (ligne 88). Ainsi toutes les exceptions remontent au code principal qui les arrêtent et les affichent ;
- lignes 11-18 : on récupère les informations de configuration de la boîte à lettres ;
- lignes 23-33 : tous les messages seront stockés dans le dossier **[output/user]** où **[output]** et **[user]** sont définis dans la configuration. On crée donc successivement les dossiers **[output]** puis **[output/user]**. Pour créer ce dernier, on le supprime d'abord ligne 31. **[shutil]** est un module qu'il faut importer. **[shutil.rmtree(dir)]** supprime le dossier **[dir]** et tout ce qu'il contient ;
- pour toutes les opérations sur les fichiers système on utilise le module **[os]** qu'il faut également importer ;
- lignes 34-38 : on ouvre une connexion avec le serveur POP3. Si le serveur est sécurisé, on utilise la classe **[poplib.POP3_SSL]** sinon la classe **[poplib.POP3]**. L'attribut **[ssl]** utilisé ligne 35 provient de la configuration de la boîte à lettres ;
- ligne 45 : on fixe un niveau de logs :
  - 0 : pas de logs ;
  - 1 : les commandes émises par le client POP3 sont loguées ;
  - 2 : logs détaillés. On voit également ce que reçoit le client POP3 ;
- ligne 47 : après la connexion, leserveur POP3 envoie un message de bienvenue. On lit celui-ci ;
- lignes 48-49 : commande USER du protocole POP3 ;
- lignes 50-51 : commande PASS du protocole POP3 ;
- lignes 52-53 : commande LIST du protocole POP3. La réponse est un tuple (response, **['mesg_num octets'...]**, octets), par exemple liste=(b'+OK 3 messages (3859 octets)', **[b'1 584', b'2 550', b'3 2725']**, 22). On voit que les deux premiers éléments du tuple sont des bytes (préfixe b). liste**[1]** est un tableau où chaque élément est une suite d'octets contenant deux informations : le n° du message et sa taille en octets ;
- ligne 56 : de ce qui précède on déduit que le nombre de messages dans la boîte à lettres peut être obtenu par **[len[liste1]]** ;
- lignes 59-84 : on boucle sur chacun des messages. On s'arrête lorsque tous ont été lus ou qu'on a atteint le nombre maximal de mails fixé par configuration ;
- ligne 61 : élément courant du tableau liste**[1]**, donc quelque chose comme b'1 584', une suite d'octets ;
- ligne 63 : on passe de la suite d'octets à une chaîne de caractères. On a maintenant la chaîne '1 584' ;
- ligne 66 : on récupère le n° du message, ici la chaîne '1' ;
- ligne 68 : on émet la commande POP3 RETR num. On récupère une réponse du genre :

```
[message=(b'+OK 584 octets', [b'Return-Path: guest@localhost', b'Received: from [127.0.0.1] (localhost
[127.0.0.1])', b'\tby DESKTOP-528I5CU with ESMTPA', b'\t; Tue, 17 Mar 2020 09:41:50 +0100', b'To:
guest@localhost', b'From: "guest@localhost" <guest@localhost>', b'Subject: test', b'Message-ID:
<2572d0f0-5b7c-2c31-5a70-c628293d5709@localhost>', b'Date: Tue, 17 Mar 2020 09:41:48 +0100', b'User-
Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:68.0) Gecko/20100101', b' Thunderbird/68.6.0', b'MIME-
Version: 1.0', b'Content-Type: text/plain; charset=utf-8; format=flowed', b'Content-Transfer-Encoding:
8bit', b'Content-Language: fr', b'', b'h\xc3\xa9l\xc3\xa8ne est all\xc3\xa9e au march\xc3\xa9 acheter
des l\xc3\xa9gumes.', b''], 614)]
```

- message est un tuple de trois éléments ;
- message**[1]** est un tableau de lignes. Chaque ligne est une suite d'octets (préfixe b). Le message complet est formé par cet ensemble de lignes ;
- **[Return-Path, Received, To, Subject, Message-ID, Content-Type, Content-Transfer-Encoding, Content-Language]** sont les entêtes du message. Chacun donne une information sur le message reçu. Ces informations vont permettre de récupérer le corps du message (avant-dernier élément du tableau message**[1]**) ;

- lignes 71-73 : on crée la chaîne **[strMessage]** formée de toutes les lignes du message. On a maintenant le message sous forme d'une chaîne de caractères. Ce message peut contenir d'autres messages ainsi que des pièces attachées. Car les pièces attachées le sont sous la forme d'une chaîne de caractères. Donc un point à retenir, c'est qu'un mail est au départ une chaîne de caractères et c'est cette chaîne de caractères qu'il faut analyser pour en extraire les pièces attachées, les éventuels autres messages encapsulés et bien sûr le corps du message, ce qu'a écrit l'expéditeur ;
- lignes 74-78 : on va ranger le corps du message et les pièces attachées message dans le dossier **[dir3]** ;
- lignes 79-80 : on va déléguer l'analyse du message à une fonction **[save_message]** :
  - le 1er paramètre est **[dir3]**, le dossier dans lequel le contenu du message doit être rangé ;
  - le second paramètre est un type **[email.message.Message]**. Cet objet a les méthodes pour récupérer les différentes parties du message (corps, pièces attachées) ainsi que tous ses entêtes. Il faut importer le module **[email]** pour disposer de cet objet. La fonction **[email.message_from_string]** permet de construire un objet **[email.message.Message]** à partir de la chaîne de caractères du message ;

La fonction **[save_message]** fait partie du module **[mail_parser]** :



Le module **[mail_parser]** a été importé aux lignes 6-7 de la fonction **[readmails]** ;

Dans **[mail_parser.py]** la fonction **[save_message]** est la suivante :

```
1.  # imports
2.  import codecs
3.  import email.contentmanager
4.  import email.header
5.  import email.iterators
6.  import email.message
7.  import os
8.
9.
10. # sauvegarde d'un message de type email.message.Message
11. # cette fonction peut être appelée de façon récursive
12. def save_message(output: str, email_message: email.message.Message, irfc822=0) -> int:
13.     # output : dossier de sauvegarde des messages
14.     # email_message : le message à sauvegarder
15.     # irfc822 : n° courant de la numérotation des mails attachés
16.     #
17.     # partie du message
18.     part = email_message
19.     # les entêtes [From, To, Subject] sont trouvés dans une des parties multipart
20.     # ou bien dans une partie [text/*] lorsqu'il n'y a pas de partie [multipart]
```

```
21.     keys = part.keys()
22.     # From doit faire partie des entêtes, sinon la partie n'a pas les entêtes qu'on cherche
23.     if "From" in keys:
24.         # on récupère certains entêtes
25.         headers = [f"From: {decode_header(part.get('From'))}",
26.                    f"To: {decode_header(part.get('To'))}",
27.                    f"Subject: {decode_header(part.get('Subject'))}",
28.                    f"Return-Path: {decode_header(part.get('Return-Path'))}",
29.                    f"User-Agent: {decode_header(part.get('User-Agent'))}",
30.                    f"Date: {decode_header(part.get('Date'))}"]
31.         # sauvegarde des entêtes dans un fichier texte
32.         with codecs.open(f"{output}/headers.txt", "w", "utf-8") as file:
33.             # écriture dans fichier
34.             string = '\r\n'.join(headers)
35.             file.write(f"{string}\r\n")
36.
37.     # type de la partie [part]
38.     main_type = part.get_content_maintype()
39.     …
```

Commentaires

- ligne 12 : la fonction reçoit au plus trois paramètres :
  - **[output]** : le dossier où enregistrer le message (2ième paramètre) ;
  - **[email_message]** : un message de type **[email.message.Message]**. Ce type est un type structuré. Il contient le texte du mail ainsi que tous les fichiers attachés et offre des méthodes pour récupérer ses différents éléments ;
  - **[irfc822]** : ce paramètre est utilisé pour numéroter les mails encapsulés dans **[email_message] ;**
- ligne 18 : l'objet **[email_message]** est mis dans **[part]**. Le type **[email.message.Message]** contient des parties **[part]** (corps du message, pièces attachées, mails encapsulés) qui ont également le type **[email.message.Message]**. Chaque partie **[part]** peut avoir des sous-parties. Ainsi le type **[email.message.Message]** est un arbre d'éléments de type **[email.message.Message]** :
  - **[part.ismultipart()]** vaut **[True]** si la partie **[part]** contient des sous-parties. Celles-ci sont alors disponibles via **[part.get_payload()]** ;
  - lorsque **[part.ismultipart()]** vaut **[False]**, c'est qu'on est arrivé à une feuille de l'arbre du message initial : il peut s'agir :
    - du corps du message sous la forme d'un texte normal ;
    - du corps du message sous la forme d'un texte HTML ;
    - d'une pièce attachée (à l'exception d'un message encapsulé pour lequel **[part.ismultipart()]** vaut **[True]**) ;
- de par la nature en arbre du paramètre **[email.message.Message]**, la fonction **[save_message]** sera appelée de façon récursive. La récursivité cesse lorsqu'on atteint les feuilles de l'arbre, ç-à-d une partie **[part]** pour laquelle **[part.ismultipart()]** vaut **[False]** ;
- ligne 21 : nous demandons à voir les clés (ou entêtes) du message couramment analysé (qui de par la récursivité peut être une sous-partie du message initial) ;
- lignes 23-35 : on veut enregistrer les entêtes :
  - **[From]** : l'expéditeur du message ;
  - **[To]** : le destinataire du message ;
  - **[Subject]** : le sujet du message ;
  - **[Return-Path]** : le destinataire à qui on doit répondre si on veut répondre. En effet, cette information n'est pas toujours dans le **[From]** ;
  - **[User-Agent]** : le client POP3 qui dialogue avec le serveur POP3 ;
  - **[Date]** : date d'envoi du mail ;
- ligne 23 : seule l'une des parties d'un message contient ces entêtes. Pour les autres parties, le code des lignes 23-35 sera ignoré ;
- lignes 25-30 : on crée une liste avec les six entêtes ;
- ligne 25 : analysons le 1er entête :
  - **[part.get(key)]** permet d'avoir l'entête associé à la clé **[key]** ;
  - cet entête peut être encodé. Si cet encodage n'est pas utf-8, on décode l'entête pour le réencoder en utf-8 à l'aide de la fonction **[decode_header]** ;
  - le 1er entête sera de la forme **[From: pymail2lexemple@gmail.com]** ;
- lignes 31-35 : on sauve les entêtes dans le fichier **[output/headers.txt]** ;

La fonction **[decode_header]** est la suivante (toujours dans **[mail_parser.py]**) :

```
1.  # décodage de headers
2.  def decode_header(header: object) -> str:
3.      # on décode l'entête
4.      header = email.header.decode_header(f"{header}")
5.      # le résultat est un tableau - ici il n'aura qu'un élément de type (header, encoding)
```

```
6.      # si encoding==None, alors header est une chaîne de caractères
7.      # sinon c'est une liste d'octets codés par encoding
8.      header, encoding = header[0]
9.      if not encoding:
10.         # si pas d'encodage
11.         return header
12.     else:
13.         # si encodage, on décode
14.         return header.decode(encoding)
```

Commentaires

- ligne 4 : on décode l'entête :
  - il faut importer le module **[email.header]** ;
  - on obtient une liste de tuples **[(header1,encoding1) , (header2, encoding2)…]** ;
  - pour les entêtes **[From, To, Subject, Return-Path, Date]** la liste n'aura qu'un élément ;
  - ligne 8 : on récupère l'entête unique et son encodage :
    - si **[encoding==None]** alors **[header]** est l'entête sous la forme d'une chaîne de caractères ;
    - sinon, **[header]** est une suite d'octets représentant l'entête encodé ;
- lignes 10-11 : s'il n'y avait pas d'encodage, alors on rend l'entête ;
- lignes 12-14 : s'il y avait un encodage, alors on décode, dans une chaîne de caractères, la suite d'octets qu'on a récupérée et on rend celle-ci ;

Revenons à la fonction **[save_message]** :

```
1.  # sauvegarde d'un message de type email.message.Message
2.  # cette fonction peut être appelée de façon récursive
3.  def save_message(output: str, email_message: email.message.Message, irfc822=0) -> int:
4.      # output : dossier de sauvegarde des messages
5.      # email_message : le message à sauvegarder
6.      # irfc822 : n° courant de la numérotation des mails attachés
7.      #
8.      # partie du message
9.      part = email_message
10.     # les entêtes [From, To, Subject] sont trouvés dans une des parties multipart
11.     # ou bien dans une partie [text/*] lorsqu'il n'y a pas de partie [multipart]
12.     keys = part.keys()
13.     # From doit faire partie des entêtes, sinon la partie n'a pas les entêtes qu'on cherche
14.     if "From" in keys:
15.         # on récupère certains entêtes
16.         headers = [f"From: {decode_header(part.get('From'))}",
17.                    f"To: {decode_header(part.get('To'))}",
18.                    f"Subject: {decode_header(part.get('Subject'))}",
19.                    f"Return-Path: {decode_header(part.get('Return-Path'))}",
20.                    f"User-Agent: {decode_header(part.get('User-Agent'))}",
21.                    f"Date: {decode_header(part.get('Date'))}"]
22.         # sauvegarde des entêtes dans un fichier texte
23.         with codecs.open(f"{output}/headers.txt", "w", "utf-8") as file:
24.             # écriture dans fichier
25.             string = '\r\n'.join(headers)
26.             file.write(f"{string}\r\n")
27.
28.     # type de la partie [part]
29.     main_type = part.get_content_maintype()
30.     sub_type = part.get_content_subtype()
31.     type_of_part = f"{main_type}/{sub_type}"
32.     # si le message est de type text/plain
33.     if type_of_part == "text/plain":
34.         # message texte
35.         save_textmessage(output, part, 0)
36.
37.     # si le message est de type text/html
38.     elif type_of_part == "text/html":
39.         # message HTML
40.         save_textmessage(output, part, 1)
41.
42.     # si le message est un conteneur de parties
43.     elif part.is_multipart():
44.         …
45.     else:
46.         …
47.     # on ignore les autres parties (pas text/plain, pas text/html, pas attachment)
48.     # on rend la valeur actuelle de irfc822 (numérotation des mails attachés rangés dans le dossier output)
49.     return irfc822
```

Commentaires

- lignes 1-26 : on a traité les entêtes du message initial ;
- lignes 28-31 : les parties d'un message de type **[email.message.Message]** ont un type principal et un sous-type. On les récupère ;
- lignes 32-35 : si la partie traitée a le type **[text/plain]** alors on est arrivés à une feuille de l'arbre du message initial. C'est le texte qu'a écrit l'expéditeur dans son message ;
- ligne 35 : ce texte est écrit dans un fichier :
  - le 1er paramètre **[output]** est le dossier dans lequel le texte doit être sauvegardé ;
  - le second paramètre est la partie du message qui contient le texte à sauvegarder ;
  - le troisième paramètre vaut 0 pour sauvegarder un texte normal, 1 pour un texte HTML ;
- lignes 37-40 : si la partie a le type **[text/html]** alors on est également arrivés à une feuille de l'arbre du message initial. C'est le texte qu'a écrit l'expéditeur dans son message, cette fois-ci au format HTML. Les gestionnaires de courrier ne proposent pas tous ce format ;

La fonction **[save_textmessage]** est la suivante :

```
1.  # sauvegarde d'un message texte
2.  def save_textmessage(output: str, part: email.message.Message, type_of_text: int):
3.      # entêtes
4.      headers = []
5.      # charset du message
6.      charset = part.get_content_charset()
7.      if charset is not None:
8.          charset = part.get_content_charset().lower()
9.          headers.append(f"Charset: {charset}")
10.     # mode de codage du contenu
11.     content_transfer_encoding = part.get("Content-Transfer-Encoding")
12.     if content_transfer_encoding is not None:
13.         headers.append(f"Transfer-Content-Encoding: {content_transfer_encoding}")
14.     # le mode 8bit a posé problème
15.     if content_transfer_encoding == "8bit":
16.         # on récupère le message du mail
17.         msg = part.get_payload()
18.     else:
19.         # on récupère le message du mail
20.         msg = email.contentmanager.raw_data_manager.get_content(part)
21.     # selon les types de texte
22.     filename = None
23.     if type_of_text == 0:
24.         # sauvegarde des entêtes
25.         with codecs.open(f"{output}/headers.txt", "a", "utf-8") as file:
26.             # écriture dans fichier
27.             string = '\r\n'.join(headers)
28.             file.write(f"{string}\r\n")
29.         # fichier texte pour le contenu
30.         filename = f"{output}/mail.txt"
31.     elif type_of_text == 1:
32.         # fichier html pour le contenu
33.         filename = f"{output}/mail.html"
34.     # sauvegarde du message
35.     with codecs.open(filename, "w", "utf-8") as file:
36.         # écriture dans fichier
37.         file.write(msg)
```

Commentaires

- comme les entêtes, le texte du message peut être encodé. Il peut y avoir deux encodages :
  - l'encodage initial du texte (utf-8, iso-8859-1…). C'est l'encodage utilisé par le gestionnaire de courrier qui a envoyé le message. Il est connu avec l'entête **[Content-Type]** du message reçu ;
  - un second encodage qu'a pu subir le texte précédent pour être envoyé. Il est connu avec l'entête **[Transfer-Content-Encoding]** du message reçu ;
- ligne 6 : l'encodage initial du texte ;
- ligne 11 : le second encodage que le texte a subi pour son transfert vers le destinataire ;
- lignes 9, 13 : ces deux informations sont mises dans la liste **[headers]**. Elles seront rajoutées aux informations du fichier **[headers.txt]** qui enregistre certains entêtes du message ;
- ligne 20 : **[email.contentmanager.raw_data_manager.get_content]** permet d'avoir le message avec son encodage initial 1. On s'est débarrassé de l'encodage 2. Seulement l'objet **[email.contentmanager.raw_data_manager]** ne gère que deux types de **[Transfer-Content-Encoding]** :

    - **[quoted-printable]** ;
    - **[base64]** ;

    Il ignore les autres. Or Thunderbird par exemple utilise le **[Transfer-Content-Encoding]** nommé "8bit". Cet encodage est ignoré et les messages avec des caractères accentués sont dénaturés. Le message peut alors être obtenu par la méthode **[part.get_payload()]** (lignes 15-17) ;
- ligne 21 : lorsqu'on est là, on a le message débarrassé de son encodage de transfert, donc le message tel qu'il a été écrit par l'expéditeur ;
- lignes 22-37 : on est dans le cas où on doit sauvegarder un message texte ;
    - lignes 24-28 : on sauvegarde les deux entêtes construits lignes 9, 13 dans le fichier **[headers.txt]**. Celui-ci existe déjà et contient des entêtes. Aussi utilise-t-on le mode "a" (ligne 25) pour ouvrir ce fichier. "a" signifie "append" et les nouveaux entêtes sont ajoutés (en fin de fichier) au contenu existant du fichier **[headers.txt]** ;
    - ligne 30 : le nom du fichier dans lequel sauvegarder le message texte ;
    - ligne 33 : le nom du fichier dans lequel sauvegarder le message HTML ;
    - lignes 34-37 : on sauvegarde le texte utf-8 dans un fichier ;

Revenons à la fonction **[save_message]** :

```
1.  # sauvegarde d'un message de type email.message.Message
2.  # cette fonction peut être appelée de façon récursive
3.  def save_message(output: str, email_message: email.message.Message, irfc822=0) -> int:
4.      # output : dossier de sauvegarde des messages
5.      # email_message : le message à sauvegarder
6.      # irfc822 : n° courant de la numérotation des mails attachés
7.      #
8.      # partie du message
9.      part = email_message
10.     # les entêtes [From, To, Subject] sont trouvés dans une des parties multipart
11.     # ou bien dans une partie [text/*] lorsqu'il n'y a pas de partie [multipart]
12.     keys = part.keys()
13.     # From doit faire partie des entêtes, sinon la partie n'a pas les entêtes qu'on cherche
14.     if "From" in keys:
15.         # on récupère certains entêtes
16.         headers = [f"From: {decode_header(part.get('From'))}",
17.                    f"To: {decode_header(part.get('To'))}",
18.                    f"Subject: {decode_header(part.get('Subject'))}",
19.                    f"Return-Path: {decode_header(part.get('Return-Path'))}",
20.                    f"User-Agent: {decode_header(part.get('User-Agent'))}",
21.                    f"Date: {decode_header(part.get('Date'))}"]
22.         # sauvegarde des entêtes dans un fichier texte
23.         with codecs.open(f"{output}/headers.txt", "w", "utf-8") as file:
24.             # écriture dans fichier
25.             string = '\r\n'.join(headers)
26.             file.write(f"{string}\r\n")
27.
28.     # type de la partie [part]
29.     main_type = part.get_content_maintype()
30.     sub_type = part.get_content_subtype()
31.     type_of_part = f"{main_type}/{sub_type}"
32.     # si le message est de type text/plain
33.     if type_of_part == "text/plain":
34.         # message texte
35.         save_textmessage(output, part, 0)
36.
37.     # si le message est de type text/html
38.     elif type_of_part == "text/html":
39.         # message HTML
40.         save_textmessage(output, part, 1)
41.
42.     # si le message est un conteneur de parties
43.     elif part.is_multipart():
44.         # cas particulier du mail attaché
45.         if type_of_part == "message/rfc822":
46.             # création d'un nouveau dossier output2 pour le mail attaché
47.             irfc822 += 1
48.             output2 = f"{output}/rfc822_{irfc822}"
49.             os.mkdir(output2)
50.             # sauvegarde des sous-parties du message irfc822 dans output2
51.             for subpart in part.get_payload():
52.                 # dans le nouveau dossier irfc822 redémarre à 0
53.                 save_message(output2, subpart, 0)
54.
55.         else:
56.             # on n'a pas affaire à un mail attaché
57.             # sauvegarde des sous-parties dans le dossier courant output
```

```
58.             # irfc822 doit alors être incrémenté pour chaque sous-partie message/rfc822
59.             for subpart in part.get_payload():
60.                 # save_message rend la dernière valeur de irfc822
61.                 # incrémentée de 1 si subpart="message/rfc822", pas incrémentée sinon
62.                 irfc822 = save_message(output, subpart, irfc822)
63.     else:
64.         # autres cas (pas text/plain, pas text/html, pas multipart)
65.         # attachement ?
66.         disposition = part.get('Content-Disposition')
67.         if disposition and disposition.startswith('attachment'):
68.             save_attachment(output, part)
69.     # on ignore les autres parties (pas text/plain, pas text/html, pas attachment)
70.     # on rend la valeur actuelle de irfc822 (numérotation des mails attachés rangés dans le dossier output)
71.     return irfc822
```

Commentaires

- lignes 33-40 : nous avons traité deux cas possibles d'un message à une extrémité de l'arbre du message initial (pas de sous-parties). Il nous reste encore deux cas à traiter :
  - lignes 43-62 : le cas où la partie analysée contient elle-même des sous-parties (part.ismultipart()==True) ;
  - lignes 63-68 : pour les cas restants, on ne traite que le cas où la partie analysée est un attachement ;

Nous traitons ce dernier cas. Nous sommes là encore à une extrémité du message initial (pas de sous-parties). Nous avons déjà rencontré deux cas de cette espèce : les types **text/plain** et **text/html**. Nous traitons maintenant le cas du fichier attaché.

- ligne 66 : l'attachement est repéré par la clé **[Content-Disposition]** ;
- ligne 67 : si cette clé existe et que celle-ci commence par la chaîne **[attachment]**, alors on a affaire à une pièce attachée au message ;
- ligne 68 : l'attachement est sauvegardé dans le dossier **[output]** ;

La fonction **[save_attachment]** est la suivante :

```
1.  # sauvegarde d'un attachement
2.  def save_attachment(output: str, part: email.message.Message):
3.      # nom du fichier attaché
4.      filename = os.path.basename(part.get_filename())
5.
6.      # le nom du fichier peut être encodé
7.      # par exemple =?utf-8?Q?Cours-Tutoriels-Serge-Tah=C3=A9-1568x268=2Ep
8.      filename = decode_header(filename)
9.      # on sauvegarde le fichier attaché
10.     with open(f"{output}/{filename}", "wb") as file:
11.         file.write(part.get_payload(decode=True))
```

- ligne 4 : si **[part]** est un attachement, alors le nom du fichier attaché est obtenu par **[part.get_filename]**. On ne garde que le nom du fichier pas son chemin ;
- ligne 8 : les noms des fichiers sont généralement encodés et ce de la même façon que les entêtes du message. Aussi utilise-t-on la fonction **[decode_header]** pour le décoder ;
- ligne 11 : le contenu du fichier attaché est pour l'instant une chaîne de caractères produite par l'encodage (souvent base64) en texte du contenu initial du fichier. Pour obtenir ce contenu initial on utilise la fonction **[part.get_payload(decode=True)]**. Le paramètre **[decode=True]** indique que le contenu de la pièce attachée doit être décodé. On obtient alors une suite d'octets ;
- ligne 10 : cette suite d'octets est sauvegardée dans le fichier **[output/filename]**. Le mode "wb" d'ouverture du fichier signifie *write binary* ;

Revenons au code de la fonction **[save_message]** :

```
1.  def save_message(output: str, email_message: email.message.Message, irfc822=0) -> int:
2.      # output : dossier de sauvegarde des messages
3.      # email_message : le message à sauvegarder
4.      # irfc822 : n° courant de la numérotation des mails attachés
5.      #
6.      # partie du message
7.      part = email_message
8.      # les entêtes [From, To, Subject] sont trouvés dans une des parties multipart
9.      # ou bien dans une partie [text/*] lorsqu'il n'y a pas de partie [multipart]
10.     keys = part.keys()
11.     # From doit faire partie des entêtes, sinon la partie n'a pas les entêtes qu'on cherche
12.     if "From" in keys:
13.         # on récupère certains entêtes
14.         headers = [f"From: {decode_header(part.get('From'))}",
```

```
                    f"To: {decode_header(part.get('To'))}",
                    f"Subject: {decode_header(part.get('Subject'))}",
                    f"Return-Path: {decode_header(part.get('Return-Path'))}",
                    f"User-Agent: {decode_header(part.get('User-Agent'))}",
                    f"Date: {decode_header(part.get('Date'))}"]
        # sauvegarde des entêtes dans un fichier texte
        with codecs.open(f"{output}/headers.txt", "w", "utf-8") as file:
            # écriture dans fichier
            string = '\r\n'.join(headers)
            file.write(f"{string}\r\n")

    # type de la partie [part]
    main_type = part.get_content_maintype()
    sub_type = part.get_content_subtype()
    type_of_part = f"{main_type}/{sub_type}"
    # si le message est de type text/plain
    if type_of_part == "text/plain":
        # message texte
        save_textmessage(output, part, 0)

    # si le message est de type text/html
    elif type_of_part == "text/html":
        # message HTML
        save_textmessage(output, part, 1)

    # si le message est un conteneur de parties
    elif part.is_multipart():
        # cas particulier du mail attaché
        if type_of_part == "message/rfc822":
            # création d'un nouveau dossier output2 pour le mail attaché
            irfc822 += 1
            output2 = f"{output}/rfc822_{irfc822}"
            os.mkdir(output2)
            # sauvegarde des sous-parties du message irfc822 dans output2
            for subpart in part.get_payload():
                # dans le nouveau dossier irfc822 redémarre à 0
                save_message(output2, subpart, 0)

        else:
            # on n'a pas affaire à un mail attaché
            # sauvegarde des sous-parties dans le dossier courant output
            # irfc822 doit alors être incrémenté pour chaque sous-partie message/rfc822
            for subpart in part.get_payload():
                # save_message rend la dernière valeur de irfc822
                # incrémentée de 1 si subpart="message/rfc822", pas incrémentée sinon
                irfc822 = save_message(output, subpart, irfc822)
    else:
        # autres cas (pas text/plain, pas text/html, pas multipart)
        # attachement ?
        disposition = part.get('Content-Disposition')
        if disposition and disposition.startswith('attachment'):
            save_attachment(output, part)
    # on ignore les autres parties (pas text/plain, pas text/html, pas attachment)
    # on rend la valeur actuelle de irfc822 (numérotation des mails attachés rangés dans le dossier output)
    return irfc822
```

Commentaires

- nous avons traité les cas des terminaisons de l'arbre du message initial : les parties **[text/plain, text/html et Content-Disposition=attachment;…]** Il nous reste à traiter le cas où la partie analysée est un conteneur de parties, ç-à-d qu'elle contient des sous-parties **[part.is_multipart()==True]**, ligne 41. Pour arriver aux terminaisons de l'arbre du message, il faut donc analyser ces sous-parties ;
- ligne 43 : on traite de façon particulière le cas où la partie analysée a un type **[message/rfc822]**. C'est le type d'un mail. C'est donc le cas où un mail a comme pièce attachée un autre mail ;

Le code est le suivant :

```
    # si le message est un conteneur de parties
    elif part.is_multipart():
        # cas particulier du mail attaché
        if type_of_part == "message/rfc822":
            # création d'un nouveau dossier output2 pour le mail attaché
            irfc822 += 1
            output2 = f"{output}/rfc822_{irfc822}"
            os.mkdir(output2)
```

```
9.              # sauvegarde des sous-parties du message irfc822 dans output2
10.             for subpart in part.get_payload():
11.                 # dans le nouveau dossier irfc822 redémarre à 0
12.                 save_message(output2, subpart, 0)
13.
14.         else:
15.             # on n'a pas affaire à un mail attaché
16.             # sauvegarde des sous-parties dans le dossier courant output
17.             # irfc822 doit alors être incrémenté pour chaque sous-partie message/rfc822
18.             for subpart in part.get_payload():
19.                 # save_message rend la dernière valeur de irfc822
20.                 # incrémentée de 1 si subpart="message/rfc822", pas incrémentée sinon
21.                 irfc822 = save_message(output, subpart, irfc822)
22. …
23.     return irfc822
```

- la différence entre une partie **[message/rfc822]** et les autres parties multipart est que le dossier de sauvegarde change ;
  - lignes 6-8 : pour la partie **[message/rfc822]**, le dossier de sauvegarde devient celui de la ligne 7 **[output/rfc822_x]** où x est le n° du mail attaché, 1 pour le premier, 2 pour le deuxième… ;
  - ligne 21 : pour les autres parties multipart, le dossier de sauvegarde continue à être le dossier **[output]** du message initial. On ne change pas de dossier ;
- lignes 10-12 : chaque sous-partie est sauvegardée par un appel récursif à **[save_message]**. Le 3e paramètre est l'indice de numérotation des mails encapsulés dans **[subpart]**. Au départ cet indice vaut 0 ;
- ligne 21 : même explication que pour la ligne 12, mais la valeur du 3e paramètre **[irfc822]** change. Si dans la boucle des lignes 18-21, il y a plusieurs mails encapsulés, ils doivent être rangés dans des dossiers **[…/rfc822-1…/rfc822_2…]**. Donc le 3e paramètre de la fonction **[save_message]** doit avoir successivement les valeurs, 1, 2, 3… Pour ce faire, **[save_message]** rend la valeur de **[irfc822]** (ligne 21).

Prenons un exemple et supposons que la liste des sous-parties de la ligne 18 soit **[subpart1, subpart2, subpart3, subpart4, subpart5]** et que **[subpart1, subpart3, subpart5]** soient des mails attachés, **[subpart2]** une partie text/plain et **[subpart4]** un attachement, et qu'on n'a pas encore rencontré de mail attaché dans le message **[irfc822=0]**. Dans ce cas :

- **[subpart1]** est sauvegardé par la ligne 21 : la fonction **[saveMessage]** est exécutée avec irfc822=0 ;
- **[subpart1]** est un mail attaché, donc irfc822 passe à 1 (ligne 6 du code). Un dossier **[output/irfc822_1]** est créé. La valeur rendue par **[saveMessage(ouput,subpart1,0)]** est donc 1 (ligne 23) ;
- **[subpart2]** est sauvegardé par la ligne 21 : la fonction **[saveMessage]** est exécutée avec irfc822=1 ;
- **[subpart2]** n'est pas un mail attaché. Donc irfc822 reste à 1. C'est la valeur récupérée ligne 21 ;
- **[subpart3]** est sauvegardé par la ligne 21 : la fonction **[save_message]** est exécutée avec irfc822=1 ;
- **[subpart3]** est un mail attaché, donc irfc822 passe à 2 (ligne 6 du code). Un dossier **[output/irfc822_2]** est créé. La valeur rendue par **[save_message(ouput,subpart1,1)]** est donc 2 (ligne 21) ;
- **[subpart4]** est sauvegardé par la ligne 21 : la fonction **[save_message]** est exécutée avec irfc822=2 ;
- **[subpart4]** n'est pas un mail attaché. Donc irfc822 reste à 2. C'est la valeur récupérée ligne 21 ;
- **[subpart5]** est sauvegardé par la ligne 21 : la fonction **[save_message]** est exécutée avec irfc822=2 ;
- **[subpart5]** est un mail attaché, donc irfc822 passe à 3 (ligne 6 du code). Un dossier **[output/irfc822_3]** est créé. La valeur rendue par **[save_message(ouput,subpart1,2)]** est donc 3 (ligne 21) ;

Exemples d'exécution

Nous envoyons 4 mails à **[pymail2parlexemple@gmail.com]** à partir de : **[Gmail, Outlook, em Client, Thunderbird]**

- **[Gmail]** : [https://mail.google.com/] ;
- **[Outlook]** : [https://outlook.live.com/owa/] ;
- **[em Client]** : [https://www.emclient.com/] ;
- **[Mozilla Thunderbird]** : **[https://www.thunderbird.net/fr/]** ;

Tous les mails auront le sujet **[hélène va au marché]** et comme texte **[acheter des légumes]**. On veut tester comment sont récupérés les caractères accentués.

Nous les lisons avec le script **[pop3/02/main]** configuré avec le fichier **[pop3/02/config]** suivant :

```
1.  import os
2.
3.
4.  def configure() -> dict:
5.      # configuration de l'appli
6.      config = {
7.          # liste des boîtes à lettres à gérer
```

```
        "mailboxes": [
            # server : serveur POP3
            # port : port du serveur POP3
            # user : utilisateur dont on veut lire les messages
            # password : son mot de passe
            # maxmails : le nombre maximum de mail à télécharger
            # timeout : délai d'attente maximal d'une réponse du serveur
            # delete : à vrai s'il faut supprimer du serveur les messages téléchargés
            # ssl : à vrai si la lecture des mails se fait au travers d'une liaison sécurisée
            # output : le dossier de rangement des messages téléchargés

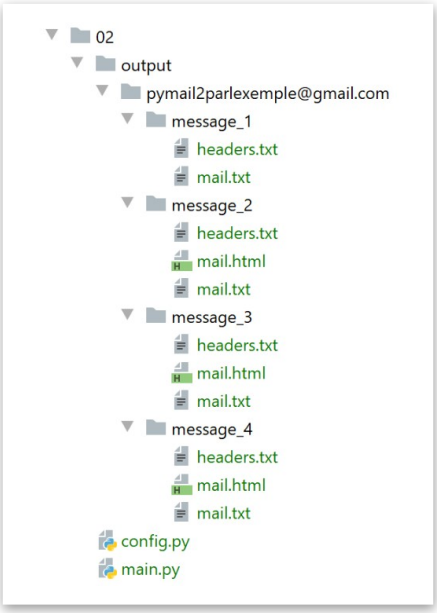
            {
                "server": "pop.gmail.com",
                "port": "995",
                "user": "pymail2parlexemple@gmail.com",
                "password": "#6prD&@1QZ3TG",
                "maxmails": 10,
                "delete": False,
                "ssl": True,
                "timeout": 2.0,
                "output": "output"
            }
        ]
    }
    # chemin absolu du dossier du script
    script_dir = os.path.dirname(os.path.abspath(__file__))

    # chemins absolus des dossiers à inclure dans le syspath
    absolute_dependencies = [
        # dossier local
        f"{script_dir}/../../shared",
    ]

    # configuration du syspath
    from myutils import set_syspath
    set_syspath(absolute_dependencies)

    # on rend la configuration
    return config
```

Le résultat est le suivant :



Le message 1 est celui envoyé par Thunderbird :



- en **[5]**, Thunderbird **[3]** utilise un **[Transfer-Content-Encoding]** de type **[8bit]** ;
- en **[4]** : le message est codé en UTF-8 ;

Le message 2 est celui envoyé par em Client :





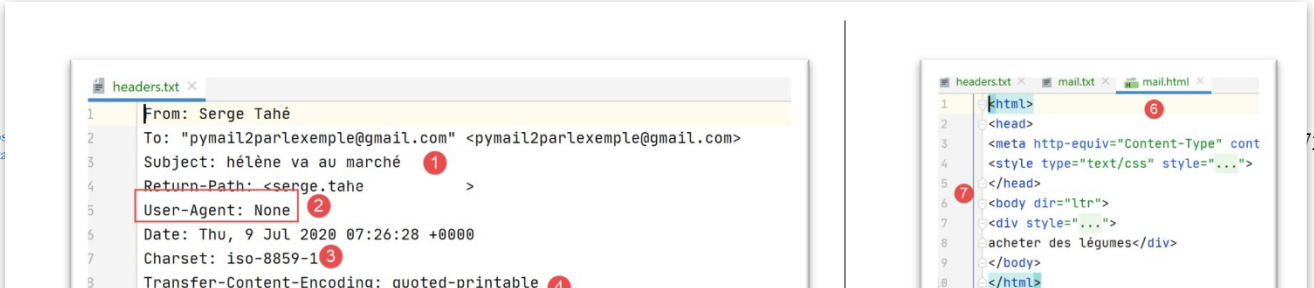
On remarquera que **[em Client]** code les textes en utf-8 **[4]** et qu'il les transfère en **[quoted-printable] [5]**. Il a également envoyé une copie du message en HTML **[7-8]**. Tous les gestionnaires de mail testés ici peuvent faire cela. Il s'agit d'un paramètre de configuration.

Le message 3 est celui envoyé par Gmail :



On remarquera que Gmail code les textes en utf-8 **[3]** et qu'il les transfère en **[quoted-printable] [4]**. En **[6]**, la version HTML du message.

Le message 4 est celui envoyé par Outlook :



On remarquera que Outlook code les textes en iso-8859-1 **[3]** et qu'il les transfère en **[quoted-printable] [4]**.

Les exemples précédents montrent deux choses :

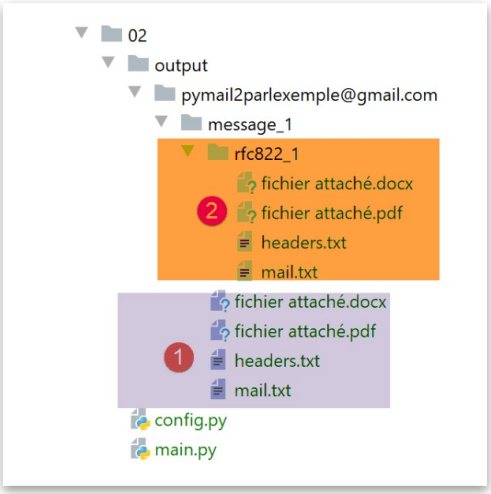
- notre client **[pop3/02]** a été fonctionnel ;
- les gestionnaires de courriers ont des façons différentes d'envoyer un mail ;

Voyons maintenant les fichiers attachés. Avec Thunderbird, nous vidons la boîte mail de l'utilisateur **[pymail2parlexemple@gmail.com]**. Puis nous utilisons le script **[smtp/03/main]** pour envoyer un mail avec la configuration **[smtp/03/config]** suivante :

```
1.  import os
2.
3.
4.  def configure() -> dict:
5.      # configuration de l'application
6.      script_dir = os.path.dirname(os.path.abspath(__file__))
7.
8.      return {
9.          # description : description du mail envoyé
10.         # smtp-server : serveur SMTP
11.         # smtp-port : port du serveur SMTP
12.         # from : expéditeur
13.         # to : destinataire
14.         # subject : sujet du mail
15.         # message : message du mail
16.         "mails": [
17.             {
18.                 "description": "mail to gmail via gmail avec smtplib",
19.                 "smtp-server": "smtp.gmail.com",
20.                 "smtp-port": "587",
21.                 "from": "pymail2parlexemple@gmail.com",
22.                 "to": "pymail2parlexemple@gmail.com",
23.                 "subject": "to gmail via gmail avec smtplib",
24.                 # on teste les caractères accentués
25.                 "message": "aglaë séléné\nva au marché\nacheter des fleurs",
26.                 # smtp avec authentification
27.                 "user": "pymail2parlexemple@gmail.com",
28.                 "password": "#6prIlhD&@1QZ3TG",
29.                 # ici, il faut mettre des chemins absolus pour les fichiers attachés
30.                 "attachments": [
31.                     f"{script_dir}/attachments/fichier attaché.docx",
32.                     f"{script_dir}/attachments/fichier attaché.pdf",
33.                     f"{script_dir}/attachments/mail attaché 1.eml",
34.                 ]
35.             }
36.         ]
37.     }
```

- lignes 31-33 : nous attachons au mail :
  - un fichier Word ;
  - un fichier PDF ;
  - un mail contenant les mêmes deux fichiers attachés ;

Une fois le mail envoyé, nous exécutons le script **[pop3/02]** pour lire la boîte mail de l'utilisateur **[pymail2parlexemple@gmail.com]**. Les résultats sont les suivants :



- en **[1]** : le message avec ses deux fichiers attachés ;
- en **[2]** : le mail attaché lui-même avec ses deux fichiers attachés ;

Conclusion

Le module **[mail_parser.py]** est particulièrement complexe. Cela est dû à la complexité des mails eux-mêmes. Nous allons réutiliser ce module pour le protocole IMAP.

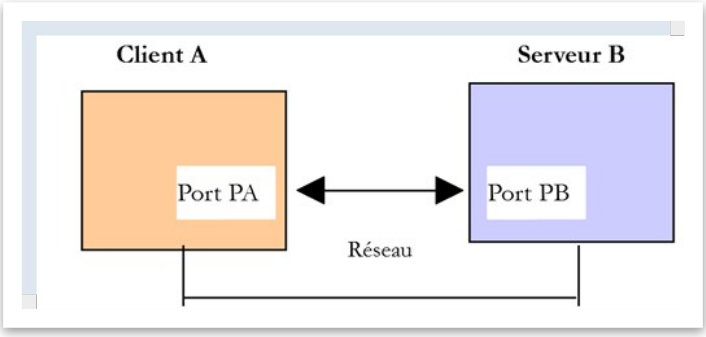
21.7 Le protocole IMAP

21.7.1 Introduction

Pour lire les mails entreposés dans un serveur de mails, deux protocoles existent :

- le protocole POP3 (Post Office Protocol) historiquement le 1er protocole mais peu utilisé maintenant ;
- le protocole IMAP (Internet Message Access Protocol) protocole plus récent que POP3 et le plus utilisé actuellement ;

Pour découvrir le protocole IMAP, nous allons utiliser l'architecture suivante :

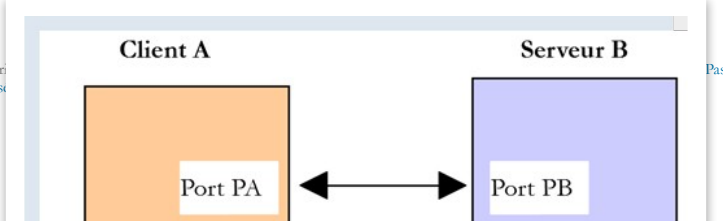


- **[Serveur B]** sera selon les cas :
  - un serveur IMAP local, implémenté par le serveur de mail **[hMailServer]** ;
  - le serveur **[imap.gmail.com:993]** qui est le serveur IMAP du gestionnaire de mails **[Gmail]** ;
- **[Client A]** sera un script Python utilisant des modules Python permettant de gérer les pièces attachées ainsi que l'utilisation d'une connexion chiffrée et authentifiée lorsque le serveur IMAP l'exige ;

Le protocole IMAP va au-delà du protocole POP3 :

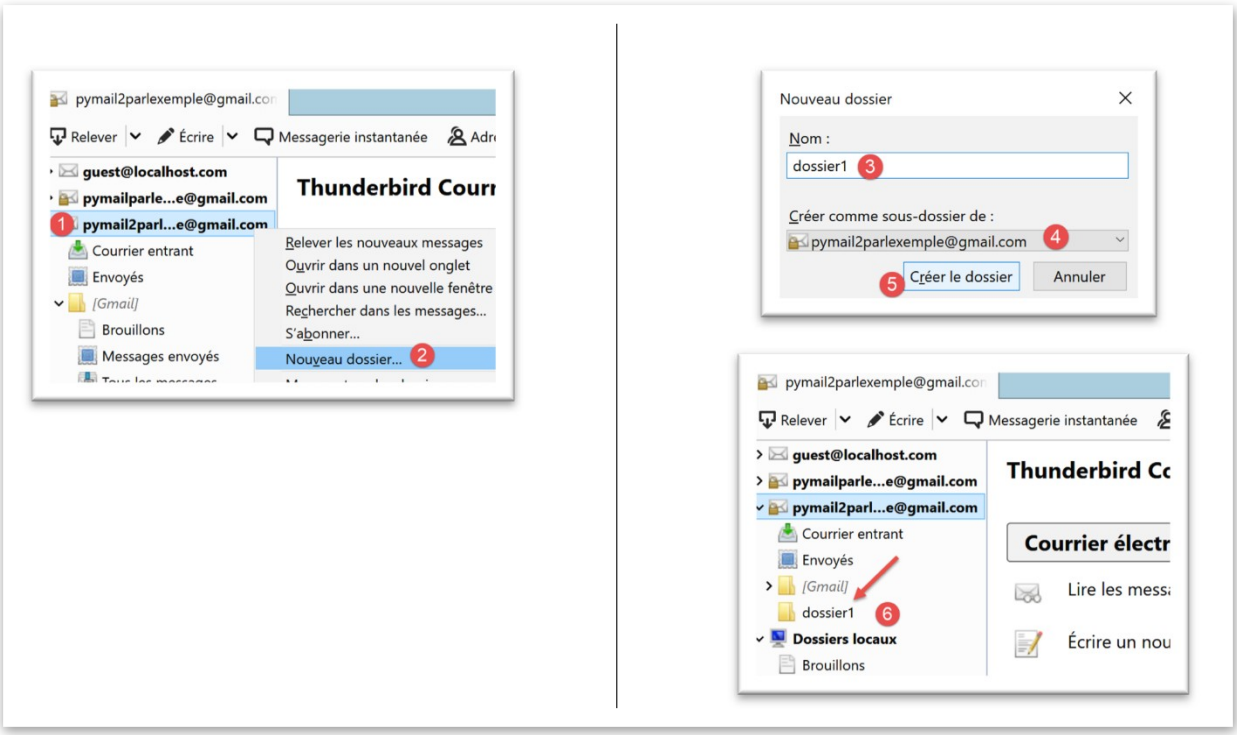
- les mails sont conservés sur le serveur IMAP et peuvent être organisés en dossiers ;
- le client IMAP peut envoyer des commandes de création / modification / suppression de ces dossiers ;

Voyons un exemple avec Thunderbird. Dans l'architecture suivante :

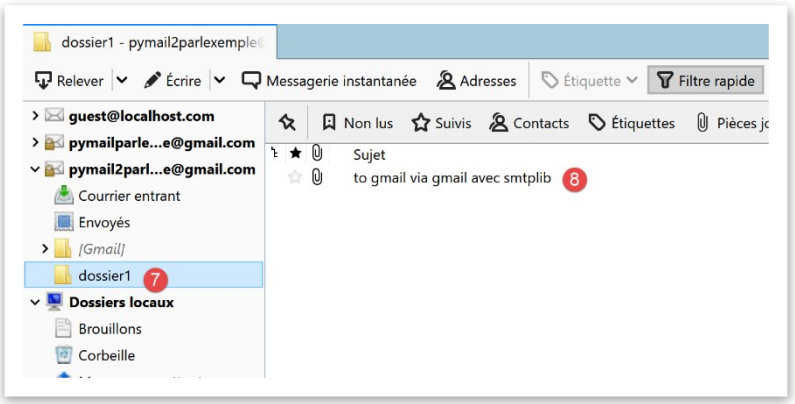


- Thunderbird est le client A ;
- **[imap.gmail.com]** est le serveur B (Gmail) ;

Créons un dossier dans les mails de l'utilisateur **[pymail2parlexemple@gmail.com]** avec Thunderbird :

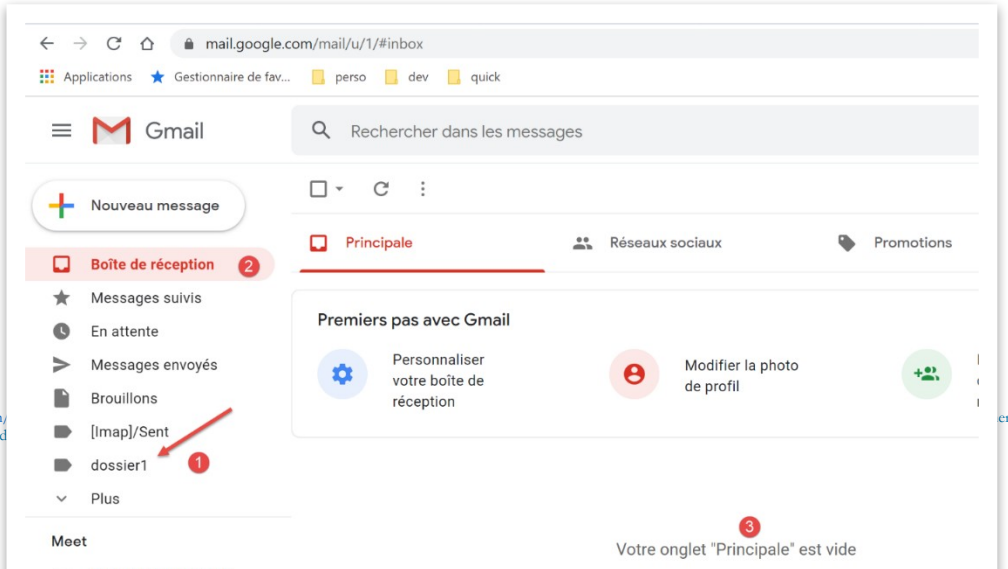


- en **[1-6]**, nous créons le dossier **[dossier1]** ;

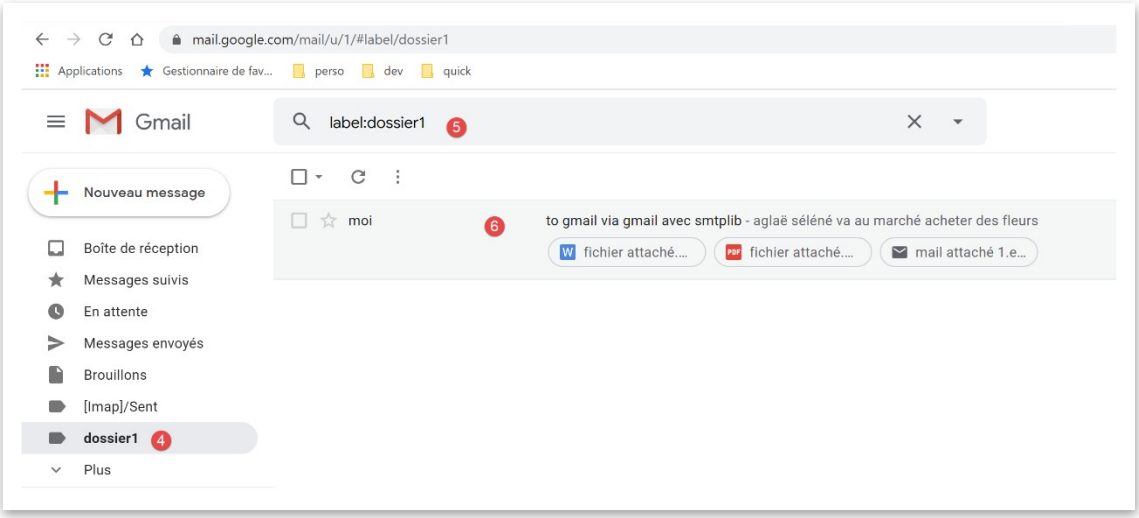


- en **[7-8]**, nous déplaçons (avec la souris) tous les fichiers du dossier **[Courrier entrant]** dans le dossier **[dossier1]** ;

Maintenant connectons-nous au site web de Gmail et identifions-nous comme l'utilisateur **[pymail2parlexemple@gmail.com]** :

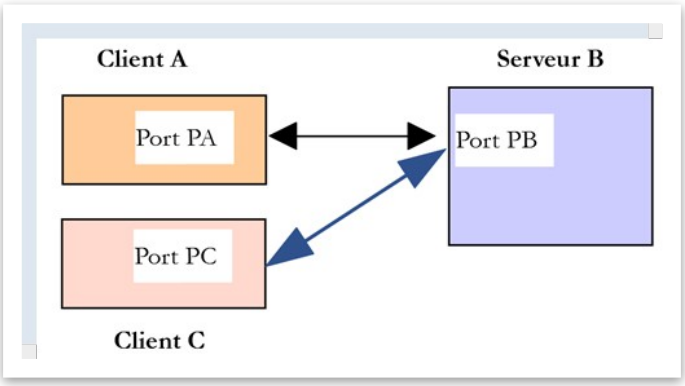


- en **[2-3]**, la boîte de réception est vide ;
- en **[1]**, le dossier **[dossier1]** qui a été créé ;



- en **[4-6]** : les mails qui ont été déplacés dans le dossier **[dossier1]** ;

Nous sommes là devant l'architecture suivante :

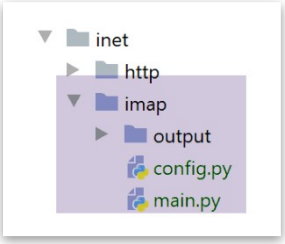


- Client A est l'application Thunderbird ;
- Client C est l'application web de Gmail ;
- Serveur B est le serveur IMAP de Gmail ;

L'arbre des dossiers de l'utilisateur est maintenu par le serveur IMAP. Ensuite tous les clients IMAP se synchronisent sur lui pour présenter à l'utilisateur les dossiers de son compte. Ici Thunderbird a envoyé plusieurs commandes pour :

- créer le dossier **[dossier1]** ;
- transférer des messages dans ce dossier ;

21.7.2 script [imap/main] : client IMAP avec le module [imaplib]



Le script **[imap/main]** est configuré par le script **[imap/config]** suivant :

```
1.  import os
2.
3.
4.  def configure() -> dict:
5.      # configuration de l'appli
6.      config = {
7.          # liste des boîtes à lettres à gérer
8.          "mailboxes": [
9.              # server : serveur IMAP
10.             # port : port du serveur IMAP
11.             # user : utilisateur dont on veut lire les messages
12.             # password : son mot de passe
13.             # maxmails : le nombre maximum de mail à télécharger
14.             # timeout : délai d'attente maximal d'une réponse du serveur
15.             # delete : à vrai s'il faut supprimer du serveur les messages téléchargés
16.             # ssl : à vrai si la lecture des mails se fait au travers d'une liaison sécurisée
17.             # output : le dossier de rangement des messages téléchargés
18.
19.             {
20.                 "server": "imap.gmail.com",
21.                 "port": "993",
22.                 "user": "pymail2parlexemple@gmail.com",
23.                 "password": "#6prIlhD&@1QZ3TG",
24.                 "maxmails": 10,
25.                 "ssl": True,
26.                 "timeout": 2.0,
27.                 "output": "output"
28.             }
29.         ]
30.     }
31.     # chemin absolu du dossier du script
32.     script_dir = os.path.dirname(os.path.abspath(__file__))
33.
34.     # chemins absolus des dossiers à inclure dans le syspath
35.     absolute_dependencies = [
36.         # dossier local
37.         f"{script_dir}/../shared",
38.     ]
39.
40.     # configuration du syspath
41.     from myutils import set_syspath
42.     set_syspath(absolute_dependencies)
43.
44.     # on rend la configuration
45.     return config
```

Commentaires

- lignes 8-29 : la clé **[mailboxes]** est associée à la liste des boîtes à lettres à consulter ;
- ligne 20 : le serveur IMAP ;
- ligne 21 : son port de service ;
- lignes 22-23 : l'utilisateur dont on veut lire les mails ;
- ligne 24 : le nombre maximum de mails qu'on veut lire ;
- ligne 25 : indique s'il faut établir une liaison sécurisée avec le serveur IMAP (True) ou pas (False) ;
- ligne 26 : le délai d'attente maximum d'attente d'une réponse du serveur ;
- ligne 27 : dossier de sauvegarde des mails lus ;

Le script **[imap/main]** est le suivant :

```
1.  # imports
2.  import email
3.  import imaplib
4.  import os
5.  import shutil
6.
7.
8.  # ------------------------------------------------------------------------
9.
10. def readmails(mailbox: dict):
11.     …
12.
13.
14. # main ----------------------------------------------------------------
15. #  client IMAP permettant de lire des mails
16.
```

```
17. # on récupère la configuration de l'application
18. import config
19. config = config.configure()
20.
21. # on traite les boîtes mail une par une
22. for mailbox in config['mailboxes']:
23.     try:
24.         # affichage console
25.         print("----------------------------------")
26.         print(
27.             f"Lecture de la boîte mail POP3 {mailbox['user']} / {mailbox['server']}:{mailbox['port']}")
28.         # lecture de la boîte mail
29.         readmails(mailbox)
30.         # fin
31.         print("Lecture terminée...")
32.     # except BaseException as erreur:
33.     #     # on affiche l'erreur
34.     #     print(f"L'erreur suivante s'est produite : {erreur}")
35.     finally:
36.         pass
```

Commentaires

- lignes 14-36 : nous retrouvons la démarche déjà rencontrée dans le script |pop3/02/main| ;

La fonction **[readmails]** est la suivante :

```
1.  def readmails(mailbox: dict):
2.      # on laisse remonter les exceptions
3.      #
4.      # module du parseur de mail
5.      from mail_parser import save_message
6.
7.      # on récupère des informations de configuration
8.      output = mailbox['output']
9.      user = mailbox['user']
10.     password = mailbox['password']
11.     timeout = mailbox['timeout']
12.     server = mailbox['server']
13.     port = int(mailbox['port'])
14.     maxmails = mailbox['maxmails']
15.     ssl = mailbox['ssl']
16.     #
17.     # c'est parti
18.     imap_resource = None
19.     try:
20.         # on crée les dossiers de stockage s'ils n'existent pas
21.         if not os.path.isdir(output):
22.             os.mkdir(output)
23.         # user
24.         dir2 = f"{output}/{user}"
25.         # on supprime le dossier [dir2] s'il existe puis on le recrée
26.         if os.path.isdir(dir2):
27.             # suppression
28.             shutil.rmtree(dir2)
29.         # création
30.         os.mkdir(dir2)
31.         # connexion au serveur IMAP
32.         if ssl:
33.             imap_resource = imaplib.IMAP4_SSL(server, port)
34.         else:
35.             imap_resource = imaplib.IMAP4(server, port)
36.         # timeout des communications du client
37.         sock = imap_resource.socket()
38.         sock.settimeout(timeout)
39.         # authentification
40.         imap_resource.login(user, password)
41.         # on sélectionne le dossier INBOX (courrier entrant)
42.         imap_resource.select('INBOX')
43.         # on récupère tous les messages de ce dossier : critère ALL
44.         # pas d'encoding particulier : None
45.         typ1, data1 = imap_resource.search(None, 'ALL')
46.         # print(f"typ={typ1}, data={data1}")
47.
48.         # data1[0] est un tableau d'octets réunissant les n°s de tous les messages séparés par un espace
49.         nums = data1[0].split()
```

```
50.         imail = 0
51.         fini = imail >= maxmails or imail >= len(nums)
52.         # on lit les mails un à un
53.         while not fini:
54.             # num est un n° de message en binaire
55.             num = nums[imail]
56.             # print(f"message n° {num}")
57.
58.             # on récupère le msg n° num
59.             typ2, data2 = imap_resource.fetch(num, '(RFC822)')
60.             # print(f"type={typ2}, data={data2}")
61.
62.             # data est une liste qui contient des tuples, ici un seul
63.             # data[0] est le tuple, data[0][1] est le deuxième élément du tuple
64.             # data[0][1] contient une suite d'octets représentant toutes les lignes du message
65.             # par message il faut entendre texte du message + tous les fichiers attachés
66.
67.             # on récupère le message comme type email.message.Message
68.             message = email.message_from_bytes(data2[0][1])
69.             # dossier du message
70.             dir3 = f"{dir2}/message_{int(num)}"
71.             # si le dossier n'existe pas, on le crée
72.             if not os.path.isdir(dir3):
73.                 os.mkdir(dir3)
74.             # on le sauvegarde
75.             save_message(dir3, message)
76.             # message suivant
77.             imail += 1
78.             fini = imail >= maxmails or imail >= len(nums)
79.     finally:
80.         if imap_resource:
81.             # on ferme la connexion avec la mailbox
82.             imap_resource.close()
83.             # on se déconnecte du serveur IMAP
84.             imap_resource.logout()
```

Commentaires

- lignes 7-15 : on récupère les éléments de la configuration ;
- lignes 19, 79 : le code est contrôlé par un try / finally. On n'intercepte donc pas les exceptions (absence de la clause except) qui vont alors remonter au code appelant qui les arrête et les affiche ;
- lignes 23-30 : on crée le dossier de sauvegarde des mails ;
- lignes 31-35 : on se connecte au serveur IMAP. La classe utilisée est différente selon qu'on a affaire à un serveur IMAP sécurisé (IMAP4_SSL) ou pas (IMAP4) ;
- lignes 36-38 : on fixe le timeout des communications client / serveur ;
- lignes 39-40 : on s'authentifie auprès du serveur IMAP ;
- lignes 41-42 : on a vu que la boîte mail d'un utilisateur IMAP pouvait être organisée en dossiers. Le dossier **[INBOX]** est celle du courrier entrant. Pour sélectionner le dossier **[dossier1]** on écrirait **[imapResource.select('dossier1')]** ;
- lignes 43-45 : on demande la liste de tous les messages trouvés dans **[INBOX]** :
  - le 1er paramètre de **[imapResource.search]** est un type d'encodage. **[None]** signifie "pas de filtre sur l'encodage" ;
  - le 2e paramètre est un critère. Il y a différentes façons d'exprimer celui-ci. Le critère **[ALL]** signifie qu'on veut tous les messages du dossier ;

  Le résultat de **[imapResource.search]** ressemble à ceci :

  ```
  typ=OK, data=[b'1 2']
  ```

  [data] est une liste qui contient les n°s des messages obtenus. Ceux-ci sont en binaire. Ci-dessus, deux messages ont été trouvés dans le dossier **[INBOX]** ;
- ligne 49 : on récupère les n°s des messages. Ci-dessus on aura la liste **[b'1' b'2']**, une liste de numéros codés en binaire ;
- lignes 53-78 : on va boucler pour lire les messages du dossier **[INBOX]** ;
- lignes 54-55 : n° du message ;
- lignes 58-59 : le message n° **[num]** est demandé au serveur IMAP ;
  - le 1er paramètre est le n° du message désiré ;
  - le second paramètre est une chaîne "(part1)(part2)…" où **[parti]** est le nom d'une partie du message. Je n'ai pas approfondi ce point. Le nom (RFC822) désigne la totalité du mail ;

  On reçoit quelque chose de la forme suivante :

```
type=OK, data=[(b'1 (RFC822 {614}', b'Return-Path: guest@localhost\r\nReceived: from [127.0.0.1]
(localhost [127.0.0.1])\r\n\tby DESKTOP-528I5CU with ESMTPA\r\n\t; Tue, 17 Mar 2020 09:41:50
```

```
+0100\r\nTo: guest@localhost\r\nFrom: "guest@localhost" <guest@localhost>\r\nSubject: test\r\nMessage-ID: <2572d0f0-5b7c-2c31-5a70-c628293d5709@localhost>\r\nDate: Tue, 17 Mar 2020 09:41:48 +0100\r\nUser-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:68.0) Gecko/20100101\r\n Thunderbird/68.6.0\r\nMIME-Version: 1.0\r\nContent-Type: text/plain; charset=utf-8; format=flowed\r\nContent-Transfer-Encoding: 8bit\r\nContent-Language: fr\r\n\r\nh\xc3\xa9l\xc3\xa8ne est all\xc3\xa9 au march\xc3\xa9 acheter des l\xc3\xa9gumes.\r\n\r\n'), b')']
```

L'élément **[data]** est ici une liste à 1 élément et cet unique élément est un tuple de trois éléments :

```
1.  data = [
2.      (b'1 (RFC822 {614}',
3.       b'Return-Path: guest@localhost\r\nReceived: from [127.0.0.1] (localhost [127.0.0.1])\r\n\tby DESKTOP-528I5CU with ESMTPA\r\n\t; Tue, 17 Mar 2020 09:41:50 +0100\r\nTo: guest@localhost\r\nFrom: "guest@localhost" <guest@localhost>\r\nSubject: test\r\nMessage-ID: <2572d0f0-5b7c-2c31-5a70-c628293d5709@localhost>\r\nDate: Tue, 17 Mar 2020 09:41:48 +0100\r\nUser-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:68.0) Gecko/20100101\r\n Thunderbird/68.6.0\r\nMIME-Version: 1.0\r\nContent-Type: text/plain; charset=utf-8; format=flowed\r\nContent-Transfer-Encoding: 8bit\r\nContent-Language: fr\r\n\r\nh\xc3\xa9l\xc3\xa8ne est all\xc3\xa9 au march\xc3\xa9 acheter des l\xc3\xa9gumes.\r\n\r\n'),
4.      b')'
5.  ]
```

Le second élément de ce tuple est une chaîne binaire représentant la totalité du message demandé. On reconnaît ci-dessus, des éléments déjà présentés lors de l'étude du module **[mail_parser]**.

data**[0]** représente un tuple à deux éléments. data**[0][1]** représente les lignes du message sous une forme binaire.

- ligne 68 : la fonction **[email.message_from_bytes(data2[0][1])]** construit un objet de type **[email.message.Message]** à partir des lignes du message. Le type **[email.message.Message]** est le type du paramètre du module **[mail_parser]** que nous avons écrit précédemment ;
- lignes 69-73 : nous créons le dossier de sauvegarde du message n° **[num]** ;
- ligne 75 : nous faisons appel à la fonction **[save_message]** du module **[mail_parser]** de la ligne 5. Cette fonction a été décrite au paragraphe |pop3/02/main| ;
- lignes 76-78 : on reboucle pour traiter le message suivant ;
- lignes 79-84 : qu'il y ait eu erreur ou pas :
  - ligne 82 : on clôt la connexion avec le dossier interrogé ;
  - ligne 84 : on se déconnecte du serveur IMAP ;

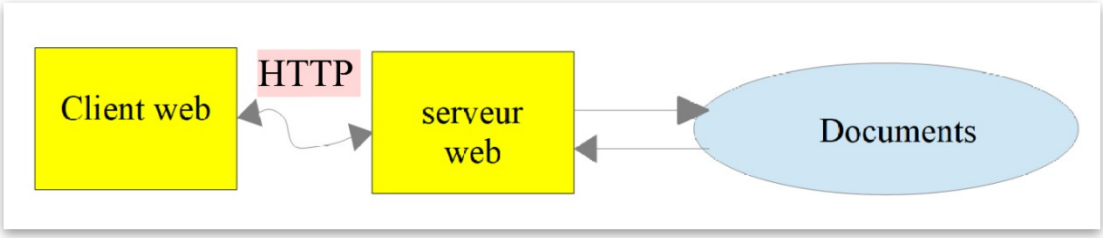
Les résultats obtenus sont identiques à ceux obtenus avec le script **[pop3/02/main]**. C'est normal puisque c'est le même parseur de mail **[mail_parser]** qui est utilisé.

22 Services web avec le framework Flask

Par *service web*, on entend ici tout application web délivrant des données brutes consommées par un client, souvent un script console dans les exemples qui vont suivre. On ne s'intéresse pas à une technologie particulière, REST (**RE**presentational **S**tate **T**ransfer) ou SOAP (**S**imple **O**bject **A**ccess **P**rotocol) par exemple, qui délivrent des données plus ou moins brutes dans un format bien défini. REST délivre du jSON alors que pour SOAP c'est du XML. Chacune de ces technologies décrit précisément la façon dont le client doit interroger le serveur et la forme que doit prendre la réponse de celui-ci. Dans ce cours, on sera beaucoup plus souple quant à la nature de la requête du client et celle de la réponse du serveur. Cependant, les scripts écrits et les outils utilisés sont proches de ceux de la technologie REST.

22.1 Introduction

Les scripts Python peuvent être exécutés par un serveur Web. Un tel script devient un programme serveur pouvant servir plusieurs clients. Du point de vue du client, appeler un service web revient à demander l'URL de ce service. Le client peut être écrit avec n'importe quel langage, notamment en Python. Dans ce dernier cas, on utilise alors les fonctions internet que nous venons de voir. Il nous faut par ailleurs savoir "converser" avec un service web, c'est à dire comprendre le protocole HTTP de communication entre un serveur Web et ses clients. C'était le but du paragraphe |le protocole HTTP|. Les clients web décrits dans cette partie du cours nous ont permis de découvrir une partie du protocole HTTP.

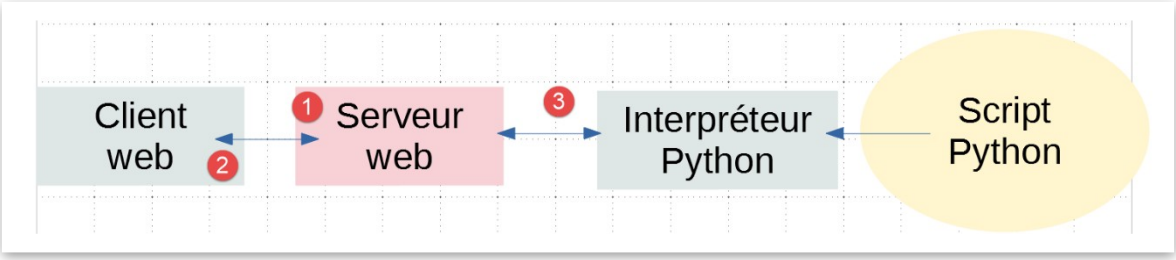


Dans leur version la plus simple, les échanges client / serveur sont les suivants :

- le client ouvre une connexion avec le port 80 du serveur web ;
- il fait une requête concernant un document ;
- le serveur web envoie le document demandé et ferme la connexion ;
- le client ferme à son tour la connexion ;

Le document peut être de nature diverse : un texte au format HTML, une image, une vidéo, ... Ce peut être un document existant (document statique) ou bien un document généré à la volée par un script (document dynamique). Dans ce dernier cas, on parle de programmation web. Le script de génération dynamique de documents peut être écrit dans divers langages : PHP, Python, Perl, Java, Ruby, C#, VB.net, ...

Dans la suite, nous allons utiliser des scripts Python pour générer dynamiquement des documents texte.



- en `[1]`, le client ouvre une connexion avec le serveur, demande un script Python, envoie ou non des paramètres à destination de ce script ;
- en `[3]`, le serveur web fait exécuter le script Python par l'interpréteur Python. Le script génère un document qui est envoyé au client `[2]` ;
- le serveur clôt la connexion. Le client en fait autant ;

Le serveur web peut traiter plusieurs clients à la fois.

Dans la suite, nous utiliserons deux serveurs web :

- le serveur léger **Werkzeug** [https://werkzeug.palletsprojects.com/en/1.0.x/]. Ce serveur est utilisé par le framework web Flask [https://flask.palletsprojects.com/en/1.1.x/]. Nous l'appellerons plus fréquemment serveur **Flask** ;

- le serveur Apache 2 [https://httpd.apache.org/] ;

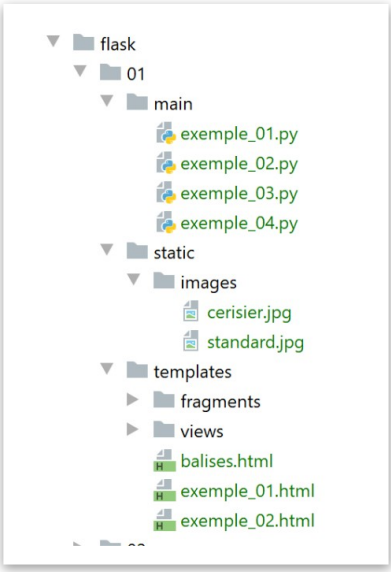
Le serveur Flask sera utilisé dans la totalité des exemples. Le serveur Apache sera utilisé pour héberger l'application web que nous allons développer.

Le framework Flask est développé en Python. C'est un module qu'on installe dans un terminal PyCharm :

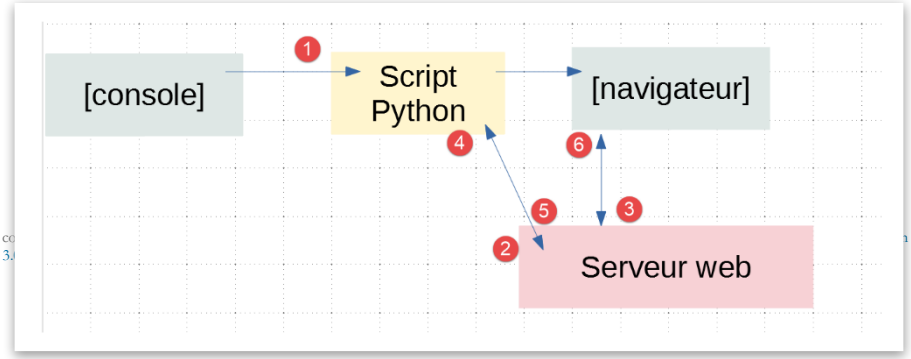
```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>pip install flask
2.  Collecting flask
3.    Downloading Flask-1.1.2-py2.py3-none-any.whl (94 kB)
4.       || 94 kB 1.1 MB/s
5.  Collecting click>=5.1
6.    Downloading click-7.1.2-py2.py3-none-any.whl (82 kB)
7.       || 82 kB 5.8 MB/s
8.  Collecting itsdangerous>=0.24
9.    Downloading itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)
10. Collecting Jinja2>=2.10.1
11.   Downloading Jinja2-2.11.2-py2.py3-none-any.whl (125 kB)
12.      || 125 kB 6.4 MB/s
13. Collecting Werkzeug>=0.15
14.   Downloading Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
15.      || 298 kB 6.4 MB/s
16. Collecting MarkupSafe>=0.23
17.   Downloading MarkupSafe-1.1.1-cp38-cp38-win_amd64.whl (16 kB)
18. Installing collected packages: click, itsdangerous, MarkupSafe, Jinja2, Werkzeug, flask
19. Successfully installed Jinja2-2.11.2 MarkupSafe-1.1.1 Werkzeug-1.0.1 click-7.1.2 flask-1.1.2
    itsdangerous-1.1.0
```

- ligne 1 : la commande exécutée ;
- ligne 19 : les éléments qui ont été installés :
  - **[flask-1.1.2]** : est un framework de développement web en Python ;
  - **[Werkzeug-1.0.1]** : est le serveur web qui va répondre aux demandes des clients ;
  - **[Jinja2-2.11.2]** : est un outil permetant d'insérer des éléments dynamiques dans des pages qui seraient autrement des pages statiques ;

22.2 scripts [flask/01] : premiers éléments de programmation web



Nos exemples seront exécutés dans l'architecture suivante :



- en **[1]**, un script Python sera exécuté comme l'est un script console classique ;
- en **[2]**, de façon transparente, un serveur web est instancié et attend des requêtes. En fait il n'acceptera qu'une unique URL ;
- en **[3]**, le navigateur demandera au serveur son unique URL ;
- en **[4]**, le serveur fera exécuter le script Python désigné par la console **[1]** ;
- en **[5]**, le script rendra ses résultats au serveur web, un document texte ;
- en **[6]**, le serveur web enverra au navigateur ce document texte ;

22.2.1 script `[exemple_01]` : rudiments du langage HTML

Un navigateur web peut afficher divers documents, le plus courant étant le document HTML (HyperText Markup Language). Celui-ci est un texte formaté avec des balises de la forme *<balise>texte</balise>*. Ainsi le texte *important* affichera le texte **important** en gras. Il existe des balises seules, telles que la balise *<hr/>* qui affiche une ligne horizontale. Nous ne passerons pas en revue les balises que l'on peut trouver dans un texte HTML. Il existe de nombreux logiciels WYSIWYG permettant de construire une page WEB sans écrire une ligne de code HTML. Ces outils génèrent automatiquement le code HTML d'une mise en page faite à l'aide de la souris et de contrôles prédéfinis. On peut ainsi insérer (avec la souris) dans la page un tableau puis consulter le code HTML généré par le logiciel pour découvrir les balises à utiliser pour définir un tableau dans une page WEB. Ce n'est pas plus compliqué que cela. Par ailleurs, la connaissance du langage HTML est indispensable puisque les applications web dynamiques doivent générer elles-mêmes le code HTML à envoyer aux clients web. Ce code est généré par programme et il faut bien sûr savoir ce qu'il faut générer pour que le client ait la page web qu'il désire.

Pour résumer, il n'est nul besoin de connaître la totalité du langage HTML pour démarrer la programmation web. Cependant cette connaissance est nécessaire et peut être acquise au travers de l'utilisation de logiciels WYSIWYG de construction de pages WEB tels que DreamWeaver et des dizaines d'autres. Une autre façon de découvrir les subtilités du langage HTML est de parcourir le web et d'afficher le code source des pages qui présentent des caractéristiques intéressantes et encore inconnues pour vous.

Considérons l'exemple suivant qui présente quelques éléments qu'on peut trouver dans un document web tels que :

- un tableau ;
- une image ;
- un lien ;



Un document HTML est encadré par les balises **<html>…</html>**. Il est formé de deux parties :

- **<head>…</head>** : c'est la partie non affichable du document. Elle donne des renseignements au navigateur qui va afficher le document. On y trouve souvent la balise **<title>…</title>** qui fixe le texte qui sera affiché dans la barre de titre du navigateur. On peut y trouver d'autres balises notamment des balises définissant les mots clés du document, mot clés utilisés

ensuite par les moteurs de recherche. On peut trouver également dans cette partie des scripts, écrits le plus souvent en javascript ou vbscript et qui seront exécutés par le navigateur ;
- **<body attributs>…</body>** : c'est la partie qui sera affichée par le navigateur. Les balises HTML contenues dans cette partie indiquent au navigateur la forme visuelle "souhaitée" pour le document. Chaque navigateur va interpréter ces balises à sa façon. Deux navigateurs peuvent alors visualiser différemment un même document web. C'est généralement l'un des casse-têtes des concepteurs web ;

Le code HTML de notre document exemple est le suivant :

```
1.  <!DOCTYPE html>
2.  <html xmlns="http://www.w3.org/1999/xhtml">
3.  <head>
4.    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5.    <title>Quelques balises HTML</title>
6.  </head>
7.
8.  <body style="background-image: url(/static/images/standard.jpg)">
9.    <h1 style="text-align: left">Quelques balises HTML</h1>
10.   <hr />
11.
12.   <table border="1">
13.     <thead>
14.       <tr>
15.         <th>Colonne 1</th>
16.         <th>Colonne 2</th>
17.         <th>Colonne 3</th>
18.       </tr>
19.     </thead>
20.     <tbody>
21.       <tr>
22.         <td>cellule(1,1)</td>
23.         <td style="text-align: center;">cellule(1,2)</td>
24.         <td>cellule(1,3)</td>
25.       </tr>
26.       <tr>
27.         <td>cellule(2,1)</td>
28.         <td>cellule(2,2)</td>
29.         <td>cellule(2,3</td>
30.       </tr>
31.     </tbody>
32.   </table>
33.   <br /><br />
34.   <table border="0">
35.     <tr>
36.       <td>Une image</td>
37.       <td>
38.         <img border="0" src="/static/images/cerisier.jpg" />
39.       </td>
40.     </tr>
41.     <tr>
42.       <td>Le site de Polytech'Angers</td>
43.       <td><a href="http://www.polytech-angers.fr/fr/index.html">ici</a></td>
44.     </tr>
45.   </table>
46. </body>
47. </html>
```

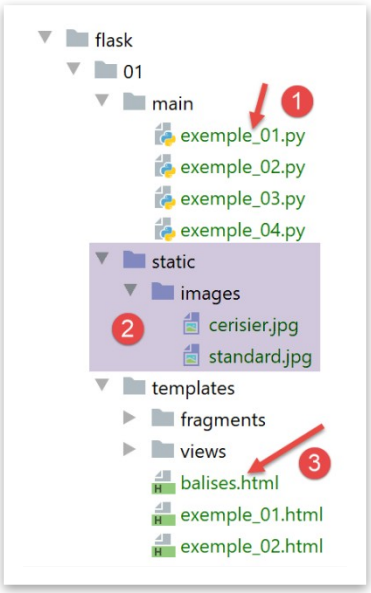
Elément	balises et exemples HTML
titre du document	`<title>Quelques balises HTML</title>` (ligne 5) le texte **[Quelques balises HTML]** apparaîtra dans la barre de titre du navigateur qui affichera le document
barre horizontale	*<hr />* : affiche un trait horizontal (ligne 10)
tableau	*<table attributs>….</table>* : pour définir le tableau (lignes 12, 32) *<thead>…</thead>* : pour définir les entêtes des colonnes (lignes 13, 19)

	<tbody>…</tbody> : pour définir le contenu du tableau (ligne 20, 31) *<tr attributs>…</tr>* : pour définir une ligne (lignes 21, 25) *<td attributs>…</td>* : pour définir une cellule (ligne 22) **exemples** : *<table border="1">…</table>* : l'attribut border définit l'épaisseur de la bordure du tableau `<td style="text-align: center;">cellule(1,2)</td>` (ligne 23) : définit une cellule dont le contenu sera **cellule(1,2)**. Ce contenu sera centré horizontalement (text-align: center).
`image`	`<img border="0" src="/static/images/cerisier.jpg"/>` (ligne 38) : définit une image sans bordure (border=0") dont le fichier source est **`[/static/images/cerisier.jpg]`** sur le serveur web (src="/static/images/cerisier.jpg"). Si ce lien se trouve sur un document web obtenu avec l'URL **`[http://server/chemin/balises.html]`**, alors le navigateur demandera l'URL **`[http://server/static/images/cerisier.jpg]`** pour avoir l'image référencée ici.
`lien`	*ici* (ligne 43) : fait que le texte *ici* sert de lien vers l'URL `http://www.polytech-angers.fr/fr/index.html`.
`fond de page`	`<body style="background-image: url(/static/images/standard.jpg)">` *(ligne 8)* : indique que l'image qui doit servir de fond de page se trouve à l'URL **`[/static/images/standard.jpg]`** du serveur web. Dans le contexte de notre exemple, le navigateur demandera l'URL **`[http://server/static/images/standard.jpg]`** pour obtenir cette image de fond.

On voit dans ce simple exemple que pour construire l'intégralité du document, le navigateur doit faire trois requêtes au serveur :

- **`[http://server/chemin/balises.html]`** pour avoir le source HTML du document ;
- **`[http://server/static/images/cerisier.jpg]`** pour avoir l'image *cerisier.jpg ;*
- **`[http://server/static/images/standard.jpg]`** pour obtenir l'image de fond *standard.jpg ;*

Le script **`[exemple_01]`** va nous permettre d'afficher la page statique **`[balises.html]`** précédente :



- en **`[1]`**, le script **`[exemple_01]`** qui va être exécuté ;
- en **`[3]`**, le document HTML qui va être affiché par le script ;
- en **`[2]`**, les images du document HTML ;

Le script **`[exemple_01]`** est le suivant :

```
1.  import os
2.
3.  from flask import Flask, make_response, render_template
```

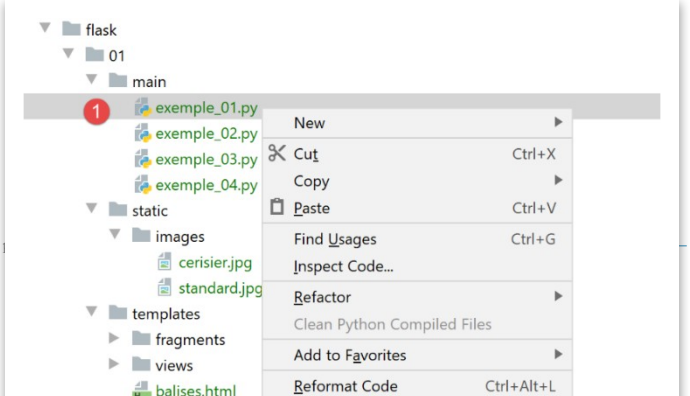
```
4.
5.  # application Flask
6.  script_dir = os.path.dirname(os.path.abspath(__file__))
7.  app = Flask(__name__, template_folder=f"{script_dir}/../templates", static_folder=f"{script_dir}/../static")
8.
9.
10. # Home URL
11. @app.route('/')
12. def index():
13.     # affichage de la page
14.     return make_response(render_template("balises.html"))
15.
16.
17. # main
18. if __name__ == '__main__':
19.     app.config.update(ENV="development", DEBUG=True)
20.     app.run()
```

- ligne 7 : on instancie une application Flask. Une application Flask est une application web ;
  - le 1er paramètre est le nom donné à l'application. On peut donner le nom que l'on veut. Ici on a utilisé l'attribut prédéfini **[__name__]** qui vaut **[__main__]** (ligne 18) ;
  - le second paramètre est un paramètre nommé, ç-à-d que sa position dans l'ordre des paramètres n'a pas d'importance. Le paramètre nommé **[template_folder]** désigne le dossier où trouver les pages statiques de l'application web. Les pages statiques sont délivrées telles quelles au navigateur. Ici, les pages statiques seront trouvées dans le dossier **[templates]** de l'arborescence du projet. Ligne 7, nous avons mis un chemin relatif au dossier **[script_dir]** contenant le script **[exemple_01]** exécuté ;
  - le troisième paramètre est également un paramètre nommé. **[static_folder]** désigne le dossier où on va trouver les ressources du document HTML (images, vidéos, …). La également, nous avons mis un chemin relatif au dossier **[script_dir]** contenant le script **[exemple_01]** exécuté ;
- lignes 10-14 : on définit les URL acceptées par l'application web. Chaque URL est associée à une fonction qui s'exécute lorsque l'URL est demandée par un navigateur web ;
- ligne 11 : l'unique URL de l'application est l'URL **[/]**. Notez que dans **[@app.route('/')]**, **[app]** est la variable initialisée ligne 7. La définition des routes (les différentes URL gérées par l'application) vient donc forcément après la définition de l'application **[app]**. Ce dernier nom est libre ;
- lignes 12-14 : la fonction qui s'exécute lorsqu'on demande l'URL **[/]** à l'application web **[exemple_01]** ;
- ligne 12 : la fonction associée à une URL peut porter un nom quelconque. Elle peut parfois avoir des paramètres pour récupérer des éléments de l'URL qui lui est associée. Ici elle n'en a pas ;
- ligne 14 :
  - la fonction **[render_template]** rend une chaîne de caractères qui est le document texte produit par son paramètre. Celui-ci est ici **[balises.html]**. A cause du **[template_folder]** de la ligne 7, ce document sera cherché dans le dossier **[f"{script_dir}/../templates"]**. C'est effectivement là qu'il se trouve ;
  - la fonction **[make_response]** génère une réponse HTTP pour le navigateur qui lui a demandé l'URL **[/]**. On a vu dans le paragraphe |le protocole HTTP| qu'une réponse HTTP a deux éléments :

    - des entêtes HTTP ;
    - le document demandé par le navigateur, ici un document HTML ;

  Ligne 14, on a donné aucun paramètre à la fonction **[make_response]** pour générer des entêtes HTTP. Elle va alors en générer par défaut. On verra ultérieurement comment fixer ces entêtes HTTP.
- finalement, lorsque le navigateur demande l'URL / à l'application Flask, il obtient la page **[balises.html]** ;
- lignes 17-20 : ces lignes servent à lancer le serveur web qui va exécuter l'application web **[exemple_01]** ;
  - ligne 18 : cette condition n'est vraie que lorsque le script **[exemple_01]** est lancé au sein d'une console ;
  - ligne 19 : l'application **[app]** de la ligne 7 est configurée :
    - le paramètre nommé **[ENV="development"]** met le serveur web en mode développement : dès que le développeur modifie un élément de l'application celle-ci est régénérée et délivrée au serveur web. Le développeur n'a pas besoin de demander une nouvelle exécution ;
    - le paramètre nommé **[DEBUG=True]** va permettre au développeur de mettre des points d'arrêt dans le code de l'application ;
  - ligne 20 : l'application web est lancée : un serveur web est instancié et l'application web est déployée dessus afin de répondre aux requêtes de clients web ;

Voici un exemple d'exécution :



Les logs suivants apparaissent alors dans la console d'exécution :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/flask/01/main/exemple_01.py
2.   * Serving Flask app "exemple_01" (lazy loading)
3.   * Environment: development
4.   * Debug mode: on
5.   * Restarting with stat
6.   * Debugger is active!
7.   * Debugger PIN: 334-263-283
8.   * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

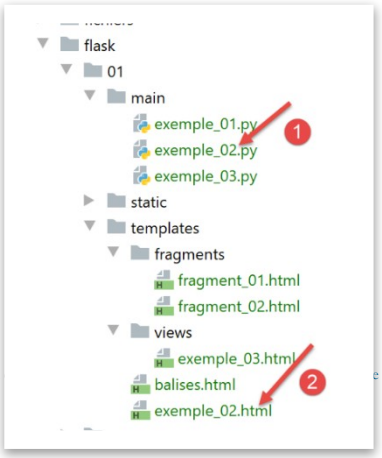
- ligne 2 : le serveur affiche le script exécuté ;
- ligne 3 : on est en mode développement ;
- lignes 4-5 : le serveur voit qu'on l'a lancé en mode **[debug]**. Il redémarre alors (ligne 5). Le mode **[debug]** ralentit donc un peu le démarrage ;
- ligne 8 : l'URL où l'application web déployée **[exemple_01]** est disponible ;

Avec un navigateur web demandons l'URL **[http://127.0.0.1:5000/]** :



On obtient bien le document **[balises.html]** attendu.

22.2.2 script [exemple_02] : générer un document HTML dynamiquement



Le script **[exemple_02] [1]** va générer le document **[exemple_02.html] [2]** suivant :

```
1.  <!DOCTYPE html>
2.  <html lang="fr">
3.  <head>
4.      <meta charset="UTF-8">
5.      <title>{{page.title}}</title>
6.  </head>
7.  <body>
8.      <b>{{page.contents}}</b>
9.  </body>
10. </html>
```

Ce document est dynamique parce que son contenu n'est totalement connu qu'au moment où le serveur web le sert. On y trouve en effet aux lignes 5 et 8 deux éléments non connus au moment de l'écriture de la page. Ils ne sont connus qu'au moment où la page est envoyée à un client. Ils sont alors remplacés par leurs valeurs, celles-ci étant des chaînes de caractères.

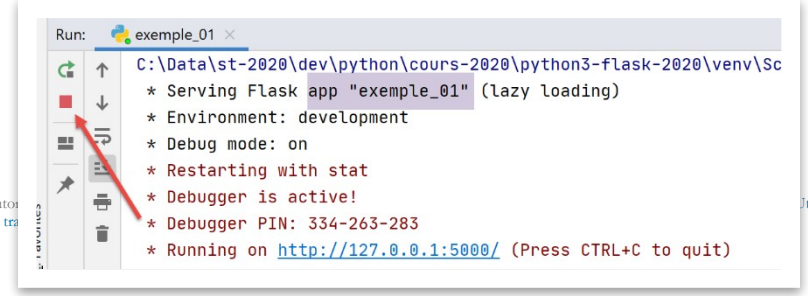
- lignes 5, 8 : la syntaxe **{{expression}}** est une syntaxe du langage de templates Jinja2 [https://jinja.palletsprojects.com/en/2.11.x/]. Avant que la page ne soit envoyée à un client, les éléments dynamiques de la page (lignes 5 et 8) sont évalués et remplacés par leurs valeurs ;
- ligne 5 : on a utilisé la syntaxe **[page.title]**. On a donc supposé qu'à la génération de la page avant son envoi, une variable **[page]** est connue, on verra comment. Dans la syntaxe **{{expression}}** on peut utiliser les noms de variables que l'on veut. Aux lignes 5 et 8, on pourrait ainsi avoir **{{title}}** et **{{contents}}**. On pourrait dire alors que **[title]** et **[contents]** sont des paramètres de la page. Dans la suite, nous utiliserons toujours la même technique :
  - l'unique paramètre de la page sera un dictionnaire **[page]** ;
  - les attributs de ce dictionnaire seront utilisés dans la page. Ici **[page.title]** ligne 5 et **[page.contents]** ligne 8 ;

L'application web **[exemple_02.py]** est la suivante :

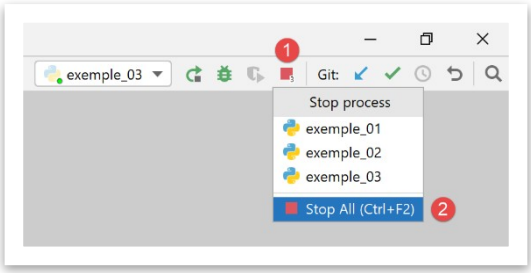
```
1.  from flask import Flask, make_response, render_template
2.
3.  # application Flask
4.  script_dir = os.path.dirname(os.path.abspath(__file__))
5.  app = Flask(__name__, template_folder=f"{script_dir}/../templates", static_folder=f"{script_dir}/../static")
6.
7.
8.  # Home URL
9.  @app.route('/')
10. def index():
11.     # contenu de la page sous la forme d'un dictionnaire
12.     page = {"title": "un titre", "contents": "un contenu"}
13.     # affichage de la page
14.     return make_response(render_template("exemple_02.html", page=page))
15.
16.
17. # main
18. if __name__ == '__main__':
19.     app.config.update(ENV="development", DEBUG=True)
20.     app.run()
```

- nous avons déjà expliqué dans l'exemple précédent, les lignes 4-5 et 18-20. Nous utiliserons toujours ce schéma dans nos exemples ;
- ligne 9 : la seule URL servie par l'application web est l'URL / ;
- ligne 14 : le document servi à l'URL / est le document **[exemple_02.html]** que nous venons de commenter. Nous savons qu'il a un paramètre, un dictionnaire appelé **[page]** ;
- ligne 12 : nous définissons le dictionnaire qui va être passé en paramètre à la page **[exemple_02.html]**. Il peut porter n'importe quel nom. Il doit cependant avoir les attributs **[title, contents]** utilisés dans le document HTML ;
- ligne 14 : la fonction **[render_template]** a pour rôle de rendre la chaîne de caractères du document **[exemple_02.html]**. Comme celui-ci est un document paramétré, on transmet à la fonction **[render_template]** le ou les paramètres attendus. Nous le faisons ici en donnant une valeur au paramètre nommé **[page]**. Dans l'opération **[page=page]** :
  - à gauche du signe =, on a le paramètre **[page]** utilisé dans le document **[exemple_02.html]** ;
  - à droite du signe =, on a la valeur **[page]** définie ligne 12 ;
  - de façon générale, si un document HTML a les paramètres **[param1, param2, …, paramn]**, on passera leurs valeurs à la fonction **[render_template]** sous la forme **[render_template(document, param1=valeur1, param2=valeur2, …]** ;

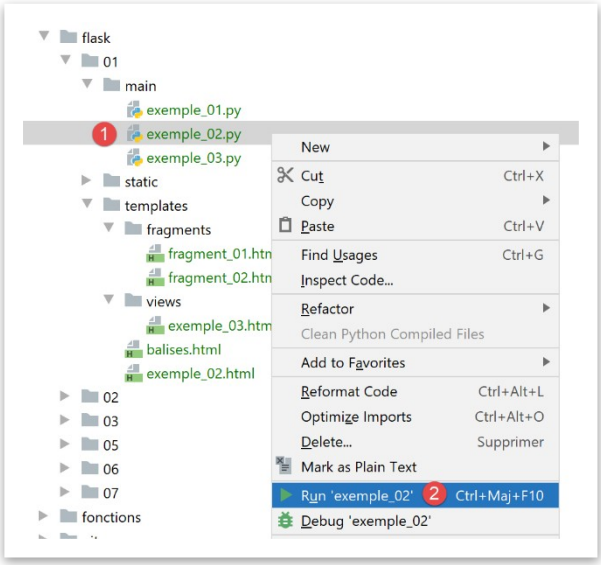
Avant d'exécuter **[exemple_02]**, nous devons arrêter l'exécution de **[exemple_01]** :



Si lors de l'exécution d'un script 1, vous avez l'impression que c'est un script 2 qui s'exécute c'est probablement parce que celui-ci est toujours en exécution. Pour revenir à un état connu, vous pouvez arrêter tous les processus en cours d'exécution dans PyCharm (en haut et à droite dans fenêtre PyCharm) :



Exécutons le script **[exemple_02]** :

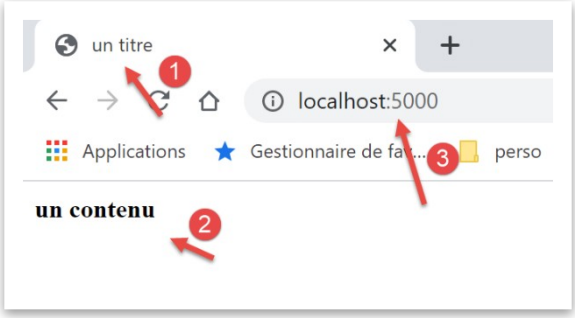


Les logs console sont alors les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/flask/01/main/exemple_02.py
2.   * Serving Flask app "exemple_02" (lazy loading)
3.   * Environment: development
4.   * Debug mode: on
5.   * Restarting with stat
6.   * Debugger is active!
7.   * Debugger PIN: 334-263-283
8.   * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

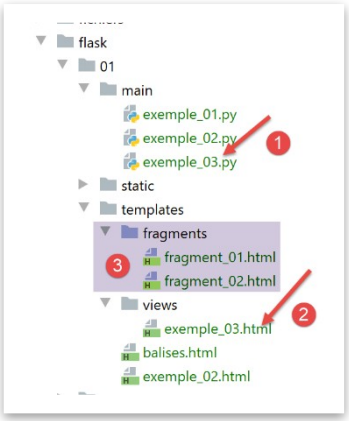
La ligne 8 indique le port de déploiement (5000) de l'application **[exemple_02]** (ligne 1) sur la machine **[localhost]**. Les lignes précédentes étant toujours les mêmes, nous ne les remontrerons plus.

Avec un navigateur, nous demandons l'URL **[http://localhost:5000/]** :



- l'expression {{page.title}} a produit **[1]** ;

- l'expression {{page.contents}} a produit **[2]** ;



22.2.3 script [exemple_03] : utiliser des fragments de page

- en **[1]**, le script **[exemple_03.py]** va générer le document dynamique **[exemple_03.html] [2]**. Celui-ci sera construit à partir des fragments de page **[fragment_01.html, fragment_02.html] [3]** ;

Le document **[exemple_03.html]** sera le suivant :

```
1.  <!DOCTYPE html>
2.  <html lang="fr">
3.  {% include "fragments/fragment_01.html" %}
4.  <body>
5.  {% include "fragments/fragment_02.html" %}
6.  </body>
7.  </html>
```

- lignes 3 et 5, on utilise la directive **[include]** de Jinja2 pour inclure dans le document des éléments externes à celui-ci ;
- la syntaxe est {% include … %}. Le paramètre de la directive **[include]** est le chemin du document à incorporer. Ce chemin est relatif au paramètre **[template_folder]** de l'application Flask :

```
app = Flask(__name__, template_folder="../templates", static_folder="../static")
```

  Donc ici, les chemins des documents sont mesurés par rapport au dossier **[templates]**.

Le fragment **[fragment_01.html]** (les noms sont bien sûr libres) est le suivant :

```
1.  <meta charset="UTF-8">
2.  <title>{{page.title}}</title>
```

Le fragment **[fragment_02.html]** est le suivant :

```
1.  <b>{{page.contents}}</b>
```

Si on reconstitue le document **[exemple_03.html]** avec ces fragments, on obtient le code suivant :

```
1.  <!DOCTYPE html>
2.  <html lang="fr">
3.  <meta charset="UTF-8">
4.  <title>{{page.title}}</title>
5.  <body>
6.  <b>{{page.contents}}</b>
7.  </body>
8.  </html>
```

On a donc un document identique à **[exemple_02.html]** mais construit à partir de fragments.

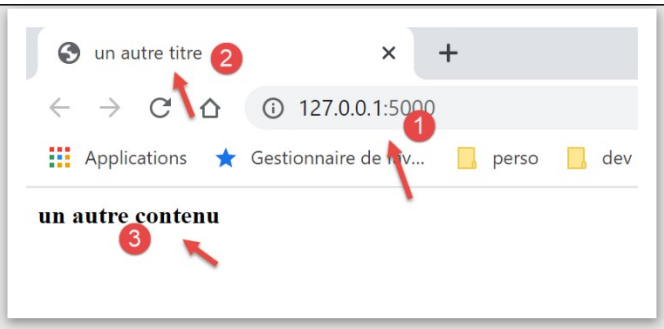
Le script web **[exemple_03.py]** est le suivant :

```
1.  import os
2.
```

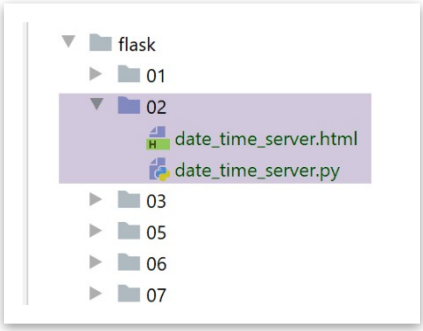
```
3.  from flask import Flask, make_response, render_template
4.
5.  # application Flask
6.  script_dir = os.path.dirname(os.path.abspath(__file__))
7.  app = Flask(__name__, template_folder=f"{script_dir}/../templates", static_folder=f"{script_dir}/../static")
8.
9.
10. # Home URL
11. @app.route('/')
12. def index():
13.     # contenu de la page
14.     page = {"title": "un autre titre", "contents": "un autre contenu"}
15.     # affichage de la page
16.     return make_response(render_template("views/exemple_03.html", page=page))
17.
18.
19. # main
20. if __name__ == '__main__':
21.     app.config.update(ENV="development", DEBUG=True)
22.     app.run()
```

Le code est analogue à celui de **[exemple_02.py]**. Ligne 16, on montre comment on peut référencer des documents présents dans des sous-dossiers de **[template_folder]** de la ligne 7.

L'exécution du script **[exemple_03.py]** donne les résultats suivants dans le navigateur :



22.3 scripts [flask/02] : service web de date et heure



Le document **[date_time_server.html]** est le suivant :

```
1.  <!DOCTYPE html>
2.  <html lang="fr">
3.  <head>
4.      <meta charset="UTF-8">
5.      <title>Date et heure du moment</title>
6.  </head>
7.  <body>
8.      <b>Date et heure du moment : {{page.date_heure}}</b>
9.  </body>
10. </html>
```

- ligne 8 : la page admet le paramètre **[page.date_heure]** ;

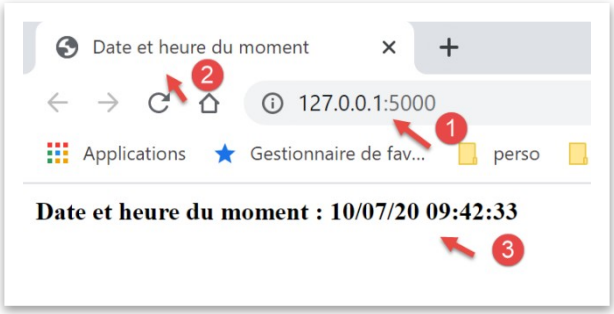
Le service web **[date_time_server.py]** est le suivant :

```
1.  # imports
2.  import os
3.  import time
```

```
4.
5.  from flask import Flask, make_response, render_template
6.
7.  # application Flask
8.  script_dir = os.path.dirname(os.path.abspath(__file__))
9.  app = Flask(__name__, template_folder=f"{script_dir}")
10.
11.
12. # Home URL
13. @app.route('/')
14. def index():
15.     # envoi heure au client
16.     # time.localtime : nb de millisecondes depuis 01/01/1970
17.     # time.strftime permet de formater l'heure et la date
18.     # format affichage date-heure
19.     # d: jour sur 2 chiffres
20.     # m: mois sur 2 chiffres
21.     # y : année sur 2 chiffres
22.     # H : heure 0,23
23.     # M : minutes
24.     # S: secondes
25.
26.     # date / heure du moment
27.     time_of_day = time.strftime('%d/%m/%y %H:%M:%S', time.localtime())
28.     # on génère le document à envoyer au client
29.     page = {"date_heure": time_of_day}
30.     document = render_template("date_time_server.html", page=page)
31.     print("document", type(document), document)
32.     # réponse HTTP au client
33.     response = make_response(document)
34.     print("response", type(response), response)
35.     return response
36.
37.
38. # main seulement
39. if __name__ == '__main__':
40.     app.config.update(ENV="development", DEBUG=True)
41.     app.run()
```

- ligne 13 : l'application web ne sert que l'URL / ;
- lignes 15-24 : expliquent comment obtenir date et heure et comment les afficher ;
- ligne 27 : chaîne de caractères représentant la date et l'heure du moment ;
- lignes 28-30 : on génère le document dynamique **[date_time_server.html]** en lui passant le dictionnaire **[page]** de la ligne 29 ;
- ligne 31 : on affiche le type de **[document]** et le document lui-même. On veut montrer que c'est une chaîne de caractères ;
- ligne 33 : on génère la réponse HTTP qui va être envoyée au client (elle n'est pas encore envoyée) ;
- ligne 34 : on affiche son type et sa valeur ;
- ligne 35 : la réponse HTTP est envoyée au client ;

L'exécution du script donne le résultat suivant dans un navigateur :



Les logs dans la console sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe C:
    \Data\st-2020\dev\python\cours-2020\python3-flask-2020\flask\02\date_time_server.py
2.   * Serving Flask app "date_time_server" (lazy loading)
3.   * Environment: development
4.   * Debug mode: on
5.   * Restarting with stat
6.   * Debugger is active!
7.   * Debugger PIN: 334-263-283
8.   * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

```
9.  127.0.0.1 - - [10/Jul/2020 09:32:09] "GET / HTTP/1.1" 200 -
10. document <class 'str'> <!DOCTYPE html>
11. <html lang="fr">
12. <head>
13.     <meta charset="UTF-8">
14.     <title>Date et heure du moment</title>
15. </head>
16. <body>
17.     <b>Date et heure du moment : 10/07/20 09:42:33</b>
18. </body>
19. </html>
20. response <class 'flask.wrappers.Response'> <Response 195 bytes [200 OK]>
```

- ligne 10 : on voit que le type de la valeur rendue par **[render_template]** est de type **[str]**. Cette chaîne de caractères n'est autre que le document **[date_time_server.html]** une fois interprété (lignes 10-19) ;
- ligne 20 : on voit que le type de la valeur rendue par **[make_response]** est de type **[flask.wrappers.Response]**. La fonction **[Response.__str__]** a été implicitement appelée pour afficher l'objet **[Response]**. La chaîne rendue par cette fonction donne deux informations sur la réponse HTTP qui va être faite :
  - le document envoyé fait 195 octets ;
  - le statut de la réponse HTTP est **[200 OK]**. On verra ultérieurement qu'on a accès à ce code de statut ;

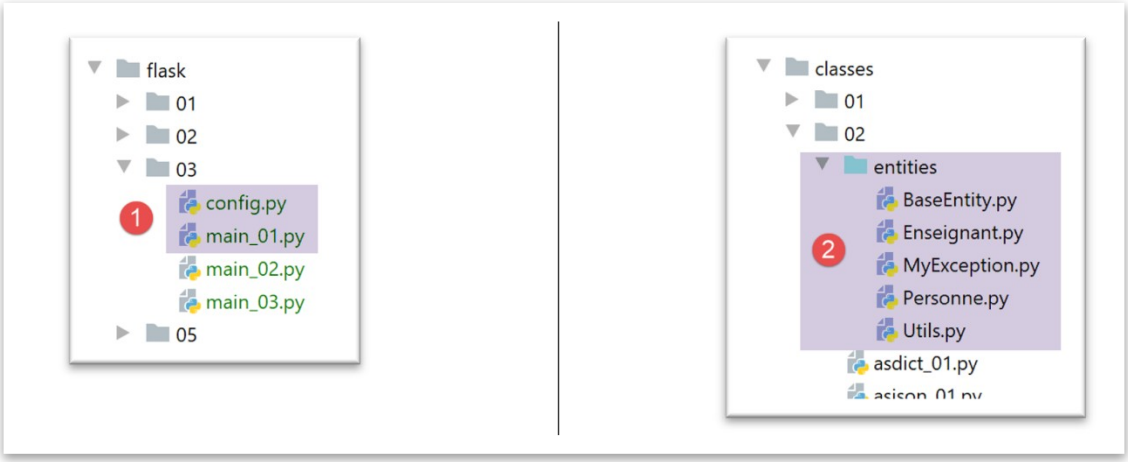
22.4 scripts [flask/03] : services web générant du texte brut

Nous avons vu dans un précédent exemple que le service web délivrait le document suivant :

```
1.  <!DOCTYPE html>
2.  <html lang="fr">
3.  <head>
4.      <meta charset="UTF-8">
5.      <title>Date et heure du moment</title>
6.  </head>
7.  <body>
8.      <b>Date et heure du moment : {{page.date_heure}}</b>
9.  </body>
10. </html>
```

Un client web pourrait n'être intéressé que par l'information **[page.date_heure]** de la ligne 8 et pas par l'habillage HTML qu'il y a autour. Le service web pourrait délivrer cette information comme une simple chaîne de caractères. Nous allons présenter ici des exemples de ce type de service web.

22.4.1 script [main_01]



- **[main_01]** est le service web ;
- **[config]** est le script de configuration de l'application web ;
- le service web utilise certaines des entités définies en **[2]** ;

Le script **[config]** est le suivant :

```
1.  def configure():
2.      # chemin absolu référence des chemins relatifs de la configuration
3.      rootDir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
4.
```

```
5.      # dépendances de l'application
6.      absolute_dependencies = [
7.          # Personne, Utils, MyException
8.          f"{rootDir}/classes/02/entities",
9.
10.     ]
11.     # on fixe le syspath
12.     from myutils import set_syspath
13.     set_syspath(absolute_dependencies)
14.
15.     # on rend la config
16.     return {}
```

Le rôle premier de cette configuration est de définir le Python Path du service web. Il faut qu'on puisse trouver les entités **[2]** (ligne 8).

Le script web **[main_01]** est le suivant :

```
1.  # on configure l'application
2.  import config
3.  config=config.configure()
4.
5.  # imports
6.  from flask import Flask, make_response
7.  from flask_api import status
8.
9.  # dépendances
10. from Personne import Personne
11.
12. # application Flask (pas de documents statiques ici)
13. app = Flask(__name__)
14.
15.
16. # Home URL
17. @app.route('/')
18. def index():
19.     # une personne
20.     personne = Personne().fromdict({"prénom": "Aglaë", "nom": "de la Hûche", "âge": 87})
21.     # réponse HTTP
22.     response = make_response(str(personne))
23.     # headers HTTP
24.     response.headers.set("Content-type", "application/json; charser=utf8")
25.     # on rend la réponse HTTP
26.     return response, status.HTTP_200_OK
27.
28.
29. # main uniquement
30. if __name__ == '__main__':
31.     # on lance le serveur
32.     app.config.update(ENV="development", DEBUG=True)
33.     app.run()
```

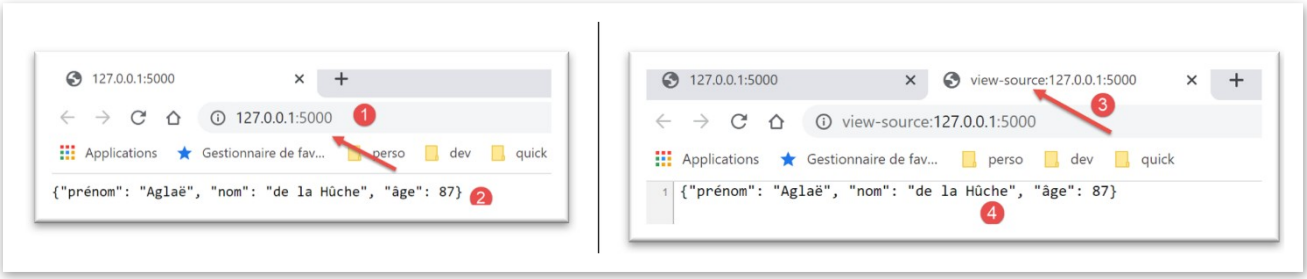
- lignes 1-3 : le Python Path de l'application est fixé ;
- lignes 5-10 : on importe les éléments dont a besoin le script ;
- ligne 17 : le service web ne sert que l'URL / ;
- ligne 20 : on crée un objet **[Personne]** ;
- ligne 22 : on crée une réponse HTTP avec la chaîne de caractères représentant la personne. La fonction **[Personne.__str__]** va être appelée. Celle-ci rend la chaîne jSON du dictionnaire **[asdict]** de la personne (cf. | classe BaseEntity |). Le paramètre de la fonction **[make_response]** est le document texte envoyé au client, donc ici la chaîne jSON d'une personne ;
- ligne 24 : on met dans les entêtes HTTP de la réponse, un entête **[Content-type]** qui indique au client quel type de document il va recevoir, ici un document jSON codé en UTF-8 ;
- ligne 26 : on rend un tuple de deux éléments :
  - la réponse au client, entêtes HTTP et document ;
  - le code de statut de la réponse. Ici on veut rendre le code de statut **[200 OK]**. Les différents codes de statut sont définis par des constantes dans le module **[flask_api]** importé ligne 7 ;

Le module **[flask_api]** n'est pas disponible nativement. Il faut l'installer. On fait cela dans un terminal PyCharm :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\inet\utilitaires>pip install flask_api
2.  Collecting flask_api
3.    Downloading Flask_API-2.0-py3-none-any.whl (119 kB)
4.       || 119 kB 544 kB/s
```

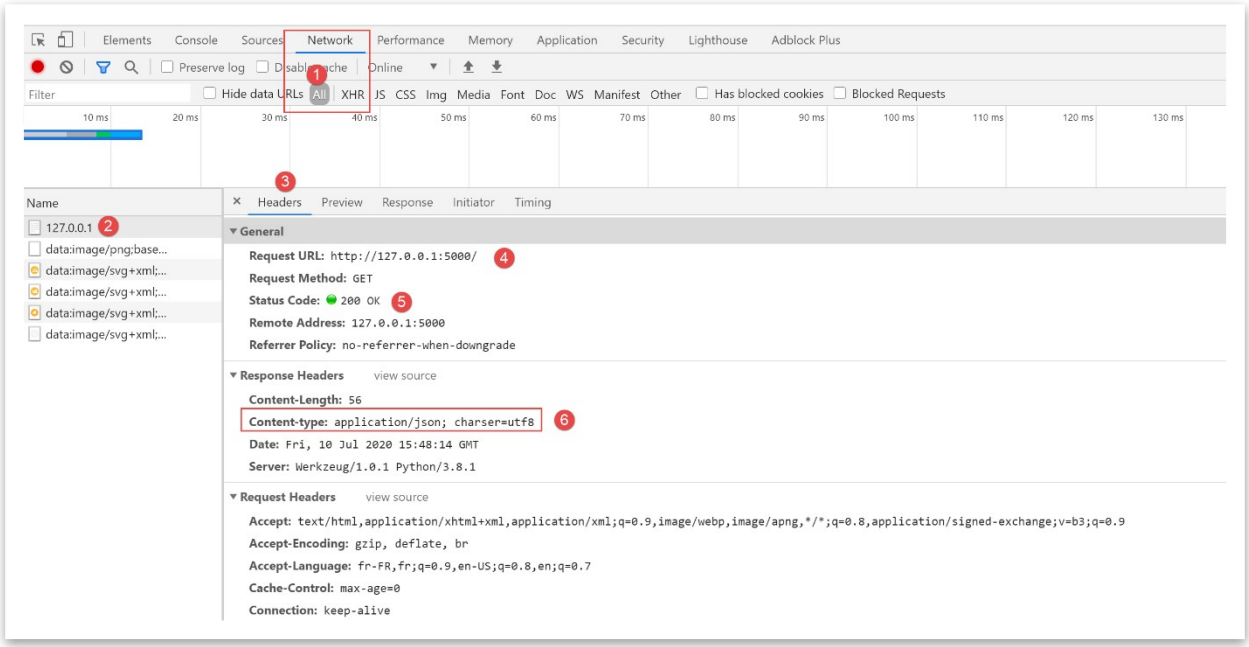
```
5.  Requirement already satisfied: Flask>=1.1 in c:\data\st-2020\dev\python\cours-2020\python3-
    flask-2020\venv\lib\site-packages (from flask_api) (1.1.2)
6.  Requirement already satisfied: Jinja2>=2.10.1 in c:\data\st-2020\dev\python\cours-2020\python3-
    flask-2020\venv\lib\site-packages (from Flask>=1.1->flask_api) (2.11.2)
7.  Requirement already satisfied: Werkzeug>=0.15 in c:\data\st-2020\dev\python\cours-2020\python3-
    flask-2020\venv\lib\site-packages (from Flask>=1.1->flask_api) (1.0.1)
8.  Requirement already satisfied: click>=5.1 in c:\data\st-2020\dev\python\cours-2020\python3-
    flask-2020\venv\lib\site-packages (from Flask>=1.1->flask_api) (7.1.2)
9.  Requirement already satisfied: itsdangerous>=0.24 in c:\data\st-2020\dev\python\cours-2020\python3-
    flask-2020\venv\lib\site-packages (from Flask>=1.1->flask_api) (1.1.0)
10. Requirement already satisfied: MarkupSafe>=0.23 in c:\data\st-2020\dev\python\cours-2020\python3-
    flask-2020\venv\lib\site-packages (from Jinja2>=2.10.1->Flask>=1.1->flask_api) (1.1.1
11. )
12. Installing collected packages: flask-api
13. Successfully installed flask-api-2.0
```

Lorsqu'on exécute le script web **[main_01]**, on obtient les résultats suivants dans un navigateur :

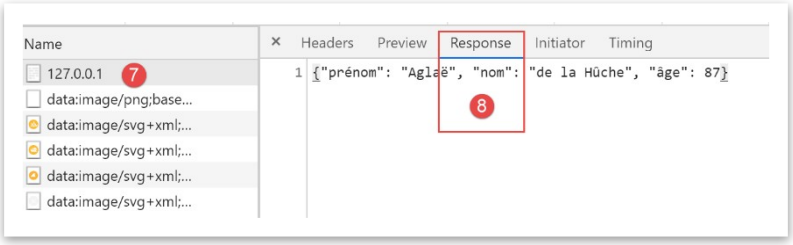


- en **[2]**, la chaîne jSON reçue ;
- en **[3-4]**, on fait afficher le contenu du document reçu. On voit qu'il n'y a aucun habillage HTML, seulement la chaîne jSON ;

Voyons maintenant le rôle de l'entête **[Content-Type]** envoyé au client par le service web. On met le navigateur en mode développeur (F12 en général) et on redemande la même URL. Ci-dessous une copie d'écran d'un navigateur Chrome :



- en **[1]**, sélectionner l'onglet **[Network]** ;
- en **[2, 4]** : l'URL demandée par le navigateur ;
- en **[3]**, sélectionner l'onglet **[Headers]** (entêtes HTTP) ;
- en **[5]**, le code de statut de la réponse HTTP reçue ;
- en **[6]**, l'entête indiquant au client qu'il va recevoir un texte jSON. Cela permet au client de s'adapter à la réponse. Ainsi la police de caractères utilisée par Chrome pour afficher une réponse jSON ou une réponse texte basique n'est pas la même ;



- en **[8]**, on sélectionne l'onglet **[Response]** pour avoir accès au document envoyé par le serveice web, ici une simple chaîne jSON ;

22.4.2 Postman

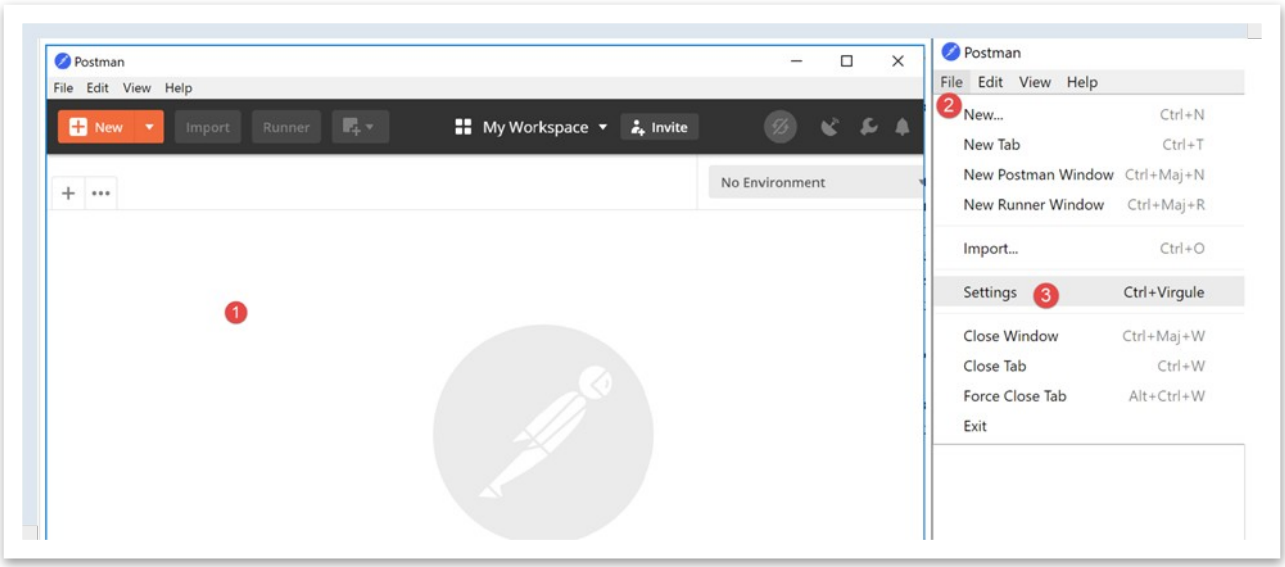
[Postman] est l'outil qui va nous permettre d'interroger les différentes URL d'une application web. Il nous permet :

- d'utiliser n'importe quelle URL : celles-ci sont fabriquées à la main ;
- de requêter le serveur web par un GET, POST, PUT, OPTIONS… ;
- de préciser les paramètres du GET ou du POST ;
- de fixer les entêtes HTTP de la requête ;
- de recevoir une réponse au format jSON, XML, HTML,
- d'avoir accès aux entêtes HTTP de la réponse. On a donc ainsi accès à la réponse HTTP complète du serveur ;

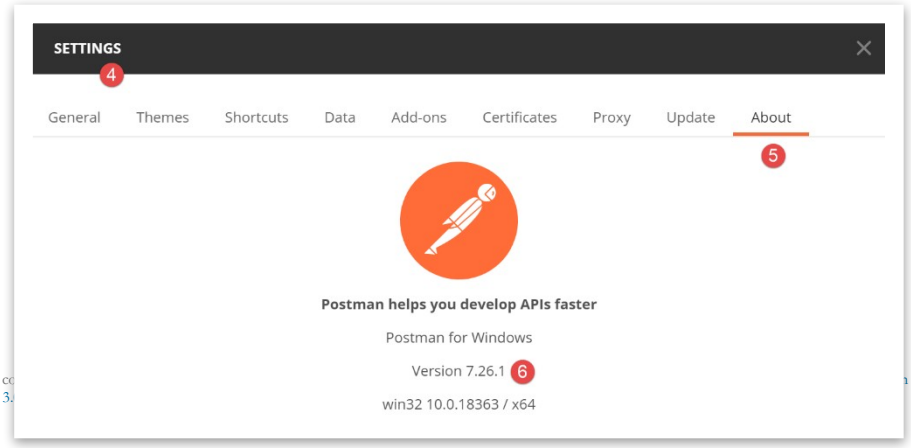
[Postman] est un excellent outil pédagogique pour comprendre la communication client / serveur du protocole HTTP.

[Postman] est disponible à l'URL **[https://www.getpostman.com/downloads/]**. Procédez à l'installation de votre version de **[Postman]**. Au cours de l'installation, on vous demandera de créer un compte : celui-ci sera inutile ici. Le compte **[Postman]** sert à synchroniser différents appareils afin que la configuration de l'un soit répliqué sur un autre. Rien de tout ceci n'est utile ici.

Une fois installé, **[Postman]** présente l'interface suivante :



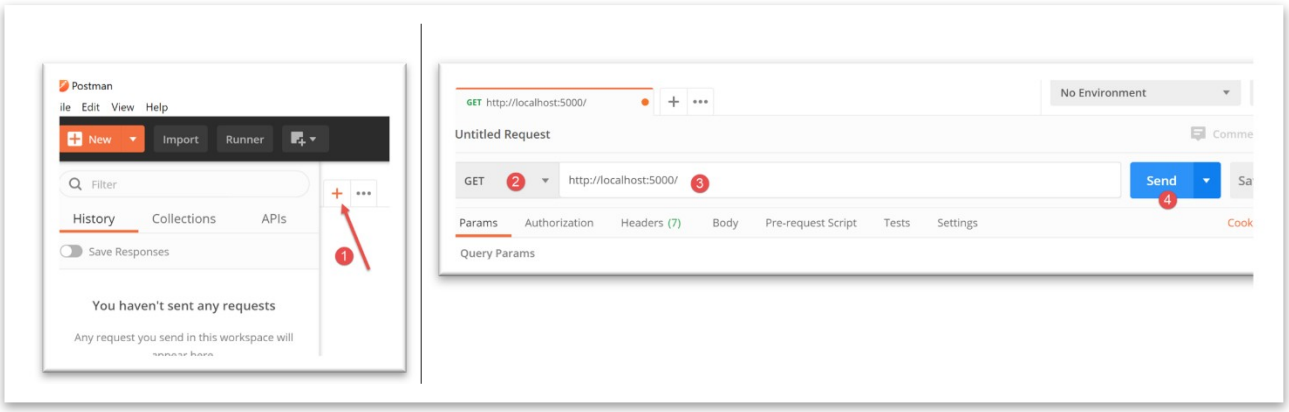
- en **[2-3]**, on a accès au paramétrage du produit ;



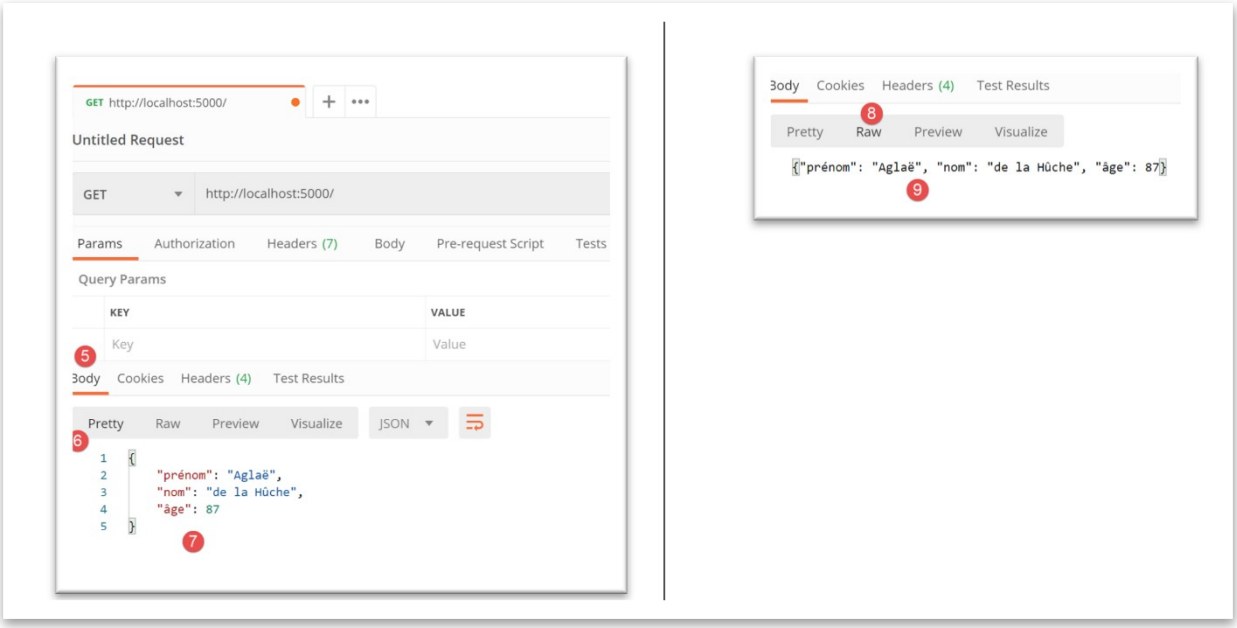
- en **[6]**, la version utilisée dans ce document ;

Nous allons ici utiliser **[Postman]** pour tester le service web jSON précédent :

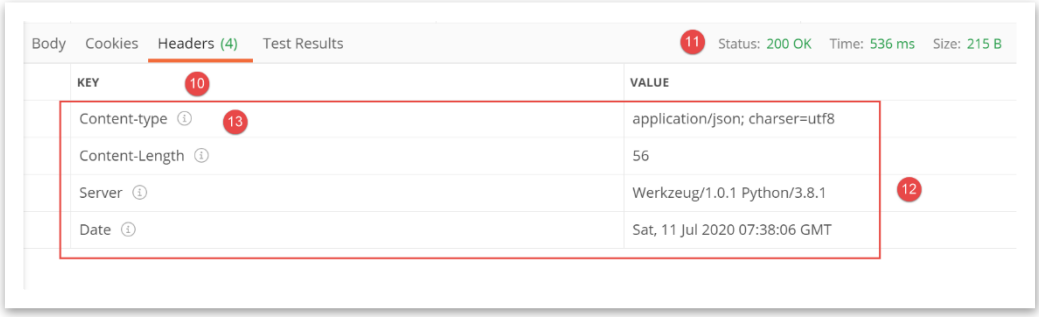
- noux exécutons le script **[flask/03/main_01]** ;
- puis nous demandons l'URL **[http://localhost:5000/]** avec Postman ;



- en **[1]**, on crée une requête ;
- en **[2]**, ce sera une requête HTTP GET ;
- en **[3]**, l'URL du service web interrogé ;
- en **[4]**, on envoie la requête au service web ;

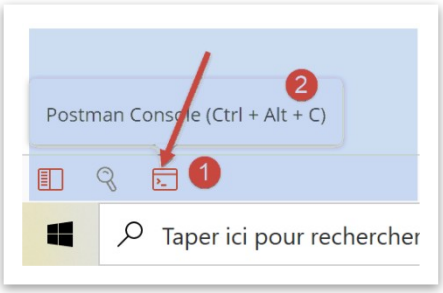


- en **[5]**, on sélectionne l'onglet **[Body]** qui affiche le document reçu ;
- en **[6]**, on sélectionne l'onglet **[Pretty]** qui affiche le document reçu avec une mise en forme appropriée, ici une forme appropriée à une chaîne jSON ;
- en **[7]**, le document jSON reçu ;
- en **[8-9]**, le document reçu sans mise en forme ;

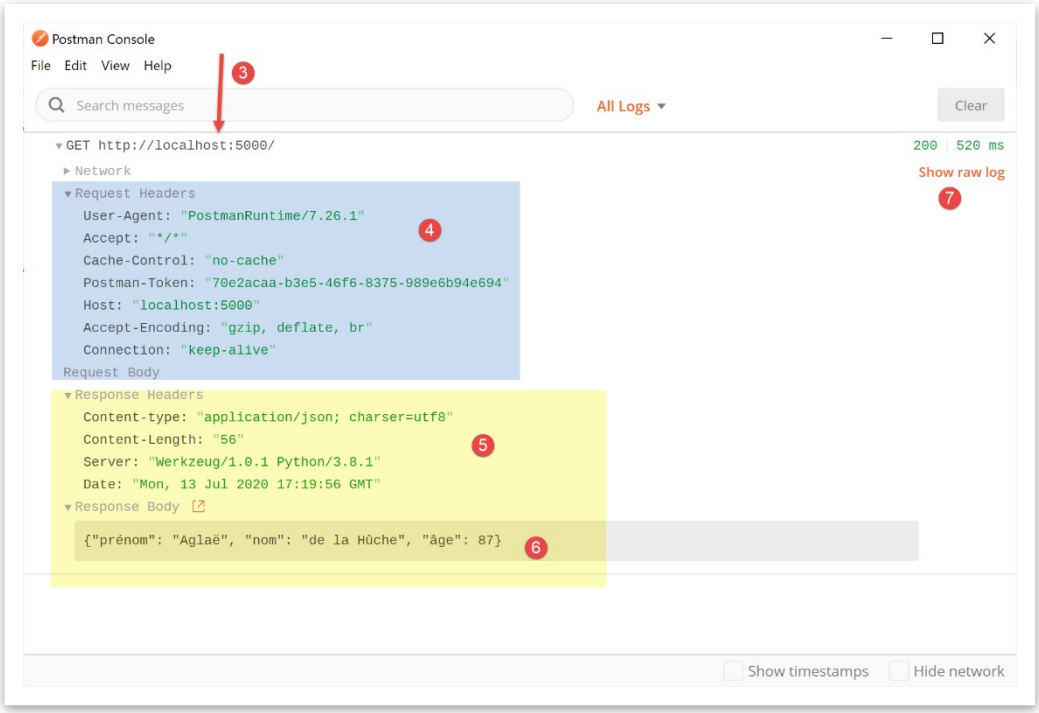


- en **[10]**, on affiche les entêtes HTTP reçus par Postman ;
- en **[11]**, le statut HTTP de la réponse reçue ;
- en **[12]**, les entêtes HTTP reçus ;
- en **[13]**, l'entête **[Content-type]** qui a permis à Postman de savoir qu'il allait recevoir une chaîne jSON. Postman a utilisé cette information pour mettre en forme, d'une certaine façon, le document reçu ;

Il y a une autre façon d'utiliser Postman. Elle consiste à utiliser la console Postman (Ctrl-Alt-C). Celle-ci permet de voir le dialogue client / serveur. Outre la séquence Ctrl-Alt-C, la console Postman est disponible via une icône en bas à gauche de la fenêtre principale de Postman :

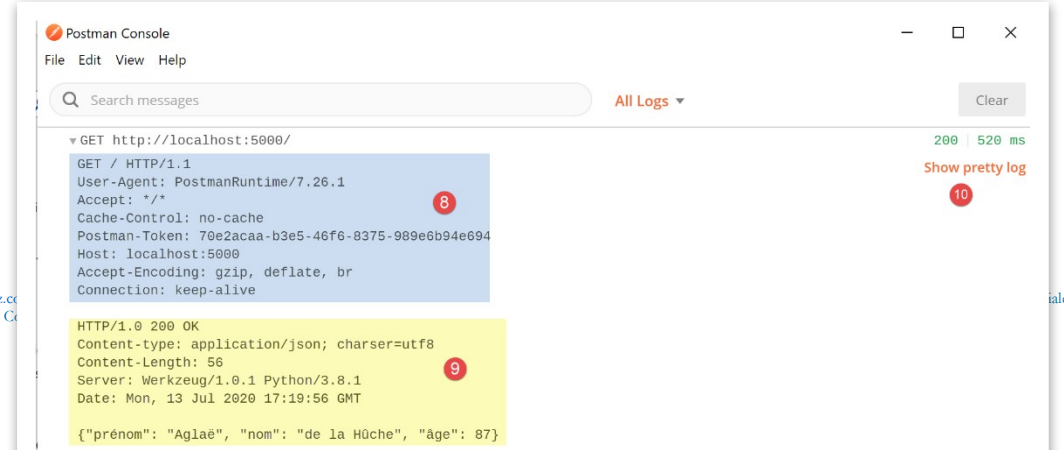


La console Postman mémorise les dialogues client / serveur qui ont lieu lorsqu'une requête Postman est exécutée :



- en **[3]**, la liste des requêtes faites par Postman depuis qu'il a été lancé. Les plus récentes sont en bas de la liste ;
- en **[4]**, la requête HTTP faite par Postman ;
- en **[5-6]**, la réponse HTTP faite par le serveur web ;
- en **[7]**, on peut voir les logs en mode **[raw]**, ç-à-d sans artifice de présentation ;

En mode **[raw]** la fenêtre de la console devient la suivante :



- en **[8]**, la requête HTTP faite par Postman au serveur web ;
- en **[9]**, la réponse HTTP faite par le serveur web ;
- en **[10]**, on peut revenir au mode **[pretty logs]** ;

Afin de faciliter les explications, nous numéroterons les lignes obtenues à partir de la console Postman.

Pour le client :

```
1.  GET / HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: 70e2acaa-b3e5-46f6-8375-989e6b94e694
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
```

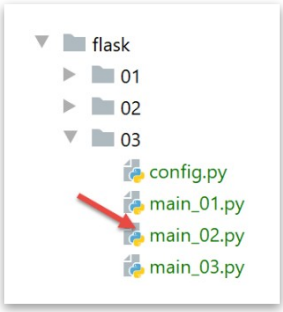
Pour le serveur :

```
1.  HTTP/1.0 200 OK
2.  Content-type: application/json; charser=utf8
3.  Content-Length: 56
4.  Server: Werkzeug/1.0.1 Python/3.8.1
5.  Date: Mon, 13 Jul 2020 17:19:56 GMT
6.  {"prénom": "Aglaë", "nom": "de la Hûche", "âge": 87}
```

A partir de maintenant, nous utiliserons principalement :

- **[Postman]** comme client web ;
- la console **[Postman]** en **[raw mode]** pour expliquer le dialogue client / serveur ;

22.4.3 script [main_02]



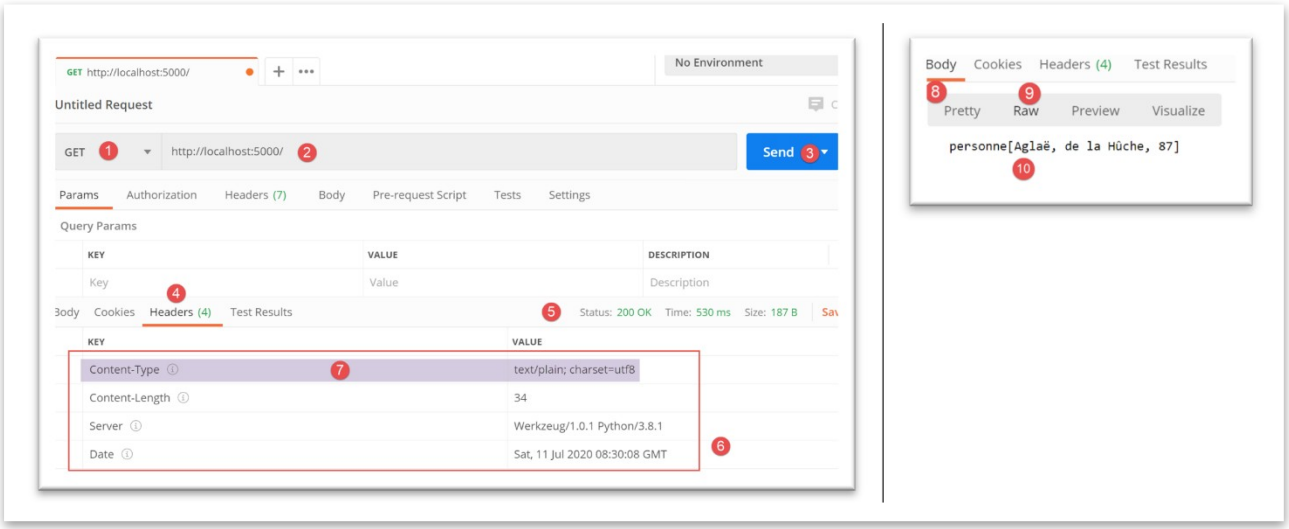
Le script web **[main_02]** est le suivant :

```
1.  # on configure l'application
2.  import config
3.  config=config.configure()
4.
5.  # imports
6.  from flask import Flask, make_response
7.  from flask_api import status
8.
9.  # dépendances
10. from Personne import Personne
11.
12. # application Flask
13. app = Flask(__name__)
14.
15.
16. # Home URL
17. @app.route('/')
18. def index():
19.     # une personne
20.     personne = Personne().fromdict({"prénom": "Aglaë", "nom": "de la Hûche", "âge": 87})
21.     # contenu
22.     response = make_response(f"personne[{personne.prénom}, {personne.nom}, {personne.âge}]")
```

```
23.     # headers HTTP
24.     response.headers.set("Content-Type", "text/plain; charset=utf8")
25.     # réponse HTTP
26.     return response, status.HTTP_200_OK
27.
28.
29. # main uniquement
30. if __name__ == '__main__':
31.     # on lance le serveur
32.     app.config.update(ENV="development", DEBUG=True)
33.     app.run()
```

- le script **[main_02]**est analogue au script **[main_01]**. Il en diffère sur deux points :
  - ligne 22 : le document envoyé au client est une chaîne de caractères brute, pas une chaîne jSON ;
  - ligne 24 : ceci est reflété dans l'entête HTTP **[Content-Type]** qui indique le type **[text/plain]** pour le document ;

Nous exécutons le script web **[main_02]** puis utilisons **[Postman]** pour l'interroger :



- en **[1-3]**, on fait la requête au service web ;
- en **[5]**, le statut OK de la réponse ;
- en **[4, 6]**, les entêtes HTTP de la réponse ;
- en **[7]**, l'entête **[Content-Type]** ;
- en **[8-10]**, le document envoyé par le service web, une chaîne de caractères ;

La console Postman donne les logs suivants :

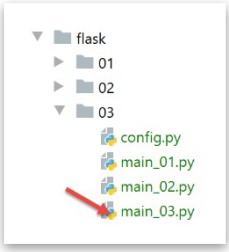
Requête du client :

```
1.  GET / HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: 7c7fc9f3-8df8-49ae-9dc8-53c2d87d111a
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
```

Réponse du serveur :

```
1.  HTTP/1.0 200 OK
2.  Content-Type: text/plain; charset=utf8
3.  Content-Length: 34
4.  Server: Werkzeug/1.0.1 Python/3.8.1
5.  Date: Mon, 13 Jul 2020 17:34:22 GMT
6.
7.  personne[Aglaë, de la Hûche, 87]
```

22.4.4 script [main_03]

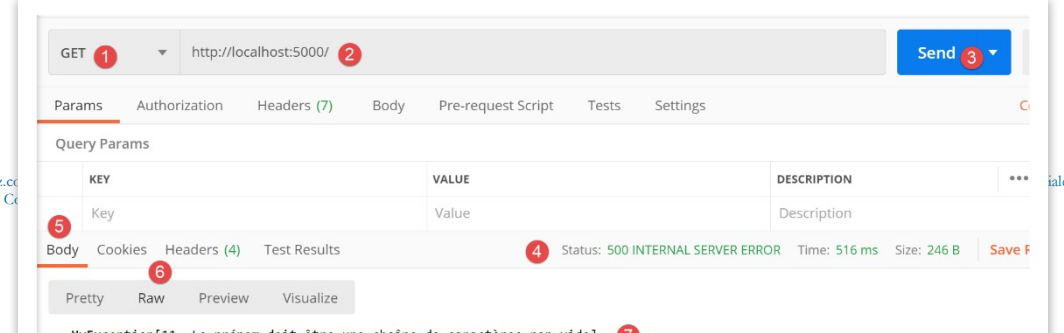


Le script web **[main_03]** est le suivant :

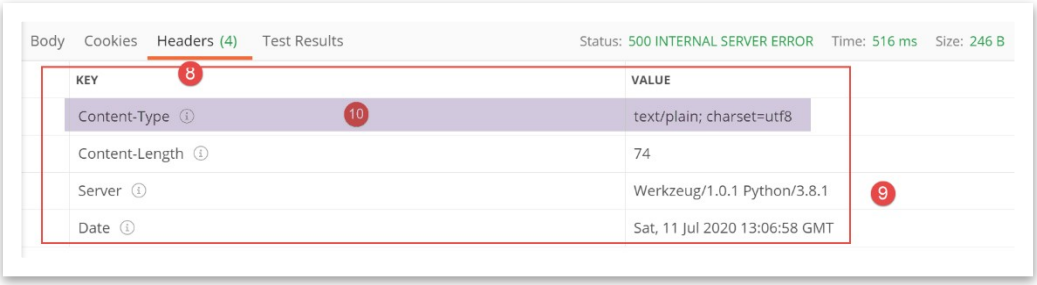
```
1.  # on configure l'application
2.  import config
3.  config = config.configure()
4.
5.  # imports
6.  from flask import Flask, make_response
7.  from flask_api import status
8.
9.  # dépendances
10. from MyException import MyException
11. from Personne import Personne
12.
13. # application Flask
14. app = Flask(__name__)
15.
16.
17. # Home URL
18. @app.route('/')
19. def index():
20.     # une personne incorrecte
21.     msg_erreur = None
22.     try:
23.         personne = Personne().fromdict({"prénom": "", "nom": "", "âge": 87})
24.     except MyException as erreur:
25.         msg_erreur = f"{erreur}"
26.     # erreur ?
27.     if msg_erreur:
28.         response = make_response(msg_erreur)
29.         status_code = status.HTTP_500_INTERNAL_SERVER_ERROR
30.     else:
31.         response = make_response(f"personne[{personne.prénom}, {personne.nom}, {personne.âge}]")
32.         status_code = status.HTTP_200_OK
33.     # headers HTTP
34.     response.headers.set("Content-Type", "text/plain; charset=utf8")
35.     # réponse HTTP
36.     return response, status_code
37.
38.
39. # main uniquement
40. if __name__ == '__main__':
41.     # on lance le serveur
42.     app.config.update(ENV="development", DEBUG=True)
43.     app.run()
```

- ligne 23 : on provoque une erreur en instanciant une personne incorrecte ;
- lignes 27-29 : à cause de l'erreur :
  - ligne 28 : on prépare une réponse HTTP ayant comme contenu le message d'erreur ;
  - ligne 29 : on donne au code de statut HTTP une valeur d'erreur **[500 Internal Server Error]** ;
- ligne 34 : on indique au client qu'on lui envoie un texte brut ;
- ligne 36 : on envoie la réponse HTTP au client ;

Nous lançons le service web **[main_03]** et nous utilisons Postman pour l'interroger :



- en **[1-3]**, nous envoyons la requête ;
- en **[4]**, on obtient une réponse avec un code de statut **[500 INTERNAL SERVER ERROR]** ;
- en **[5-7]** : la réponse est un texte décrivant l'erreur qui s'est produite ;



- en **[8-10]**, les entêtes HTTP de la réponse du service web ;

Dans la console Postman, les résultats en mode **[raw]** sont les suivants :

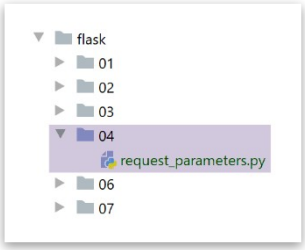
Requête du client :

```
1.  GET / HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: 925ff036-a360-47af-adf6-78173c01a247
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
```

Réponse du serveur :

```
1.  HTTP/1.0 500 INTERNAL SERVER ERROR
2.  Content-Type: text/plain; charset=utf8
3.  Content-Length: 74
4.  Server: Werkzeug/1.0.1 Python/3.8.1
5.  Date: Mon, 13 Jul 2020 17:39:24 GMT
6.
7.  MyException[11, Le prénom doit être une chaîne de caractères non vide]
```

22.5 scripts [flask/04] : informations encapsulées dans la requête

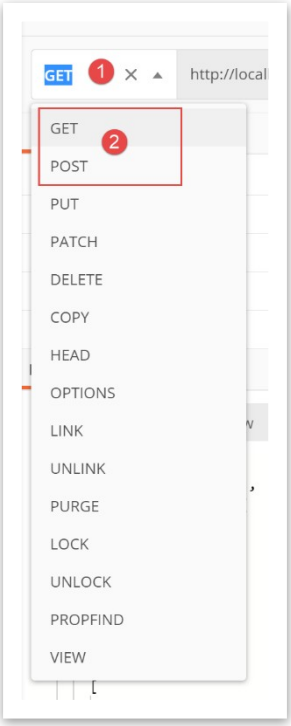


Le script **[request_parameters.py]** se propose de montrer que le service web a accès à diverses informations encapsulées dans la requête d'un client web. Le code est le suivant :

```
1.  # import
2.  from flask import Flask, make_response, request
3.  from flask_api import status
4.  # application Flask
5.  app = Flask(__name__)
6.
7.
8.  # Home URL
9.  @app.route('/', methods=['GET', 'POST'])
10. def index():
11.     # paramètres de la requête
12.     request_data = {}
13.     request_data["environ"] = f"{request.environ}"
14.     request_data["path"] = request.path
```

```
15.     request_data["full_path"] = request.full_path
16.     request_data["script_root"] = request.script_root
17.     request_data["url"] = request.url
18.     request_data["base_url"] = request.base_url
19.     request_data["url_root"] = request.url_root
20.     request_data["accept_charsets"] = request.accept_charsets
21.     request_data["accept_encodings"] = request.accept_encodings
22.     request_data["accept_languages"] = request.accept_languages
23.     request_data["accept_mimetypes"] = request.accept_mimetypes
24.     request_data["args"] = request.args
25.     request_data["content_encoding"] = request.content_encoding
26.     request_data["content_length"] = request.content_length
27.     request_data["content_type"] = request.content_type
28.     request_data["endpoint"] = request.endpoint
29.     request_data["files"] = request.files
30.     request_data["form"] = request.form
31.     request_data["host"] = request.host
32.     request_data["method"] = request.method
33.     request_data["query_string"] = request.query_string.decode()
34.     request_data["referrer"] = request.referrer
35.     request_data["remote_addr"] = request.remote_addr
36.     request_data["remote_user"] = request.remote_user
37.     request_data["scheme"] = request.scheme
38.     request_data["script_root"] = request.script_root
39.     request_data["user_agent"] = f"{request.user_agent}"
40.     request_data["values"] = request.values
41.     # réponse HTTP
42.     response = make_response(request_data)
43.     # headers HTTP
44.     response.headers["Content-Type"] = "application/json; charset=utf-8"
45.     # envoi réponse HTTP
46.     return response, status.HTTP_200_OK
47.
48.
49. # main
50. if __name__ == '__main__':
51.     app.config.update(ENV="development", DEBUG=True)
52.     app.run()
```

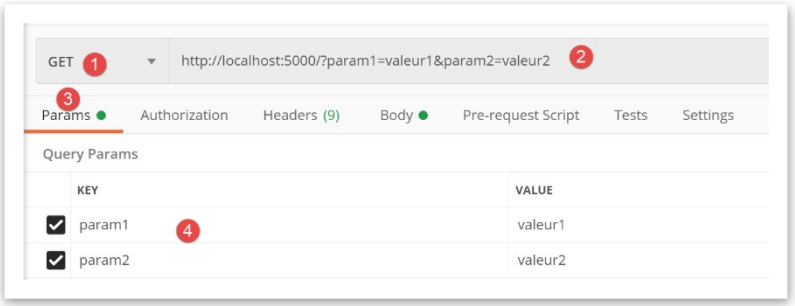
- ligne 9 : nous introduisons un changement. Nous précisons quelles sont les verbes autorisés dans la requête du client. Postman en donne la liste :



Les deux premiers **[GET, POST]** sont les plus utilisés et seront également les seuls à être utilisés dans ce document. Pour en revenir à la ligne 9 du code, le paramètre **[methods]** contient la liste des méthodes de la liste ci-dessus autorisées par l'URL. En l'absence de ce paramètre, seule la méthode **[GET]** est autorisée. C'est ce qui s'est passé jusqu'à maintenant ;

- ligne 12 : nous allons construire le dictionnaire **[request_data]** ;
- ligne 13 : la requête du client est disponible dans un objet prédéfini **[request]**, importé ligne 2, de type **[werkzeug.local.LocalProxy]**. Les lignes qui suivent récupèrent divers attributs de cet objet ;
- plutôt que de détailler chaque attribut de l'objet **[request]**, nous allons exécuter ce code et regarder les résultats. On comprendra alors mieux la signification des différents attributs affichés ;
- ligne 42 : le dictionnaire **[request_data]** sera le contenu de la réponse HTTP. On se rappelle que celui-ci doit être du texte. Flask transforme automatiquement les dictionnaires en chaînes jSON ;
- ligne 44 : on dit au client qu'il va recevoir du jSON ;
- ligne 46 : on envoie la réponse au client ;

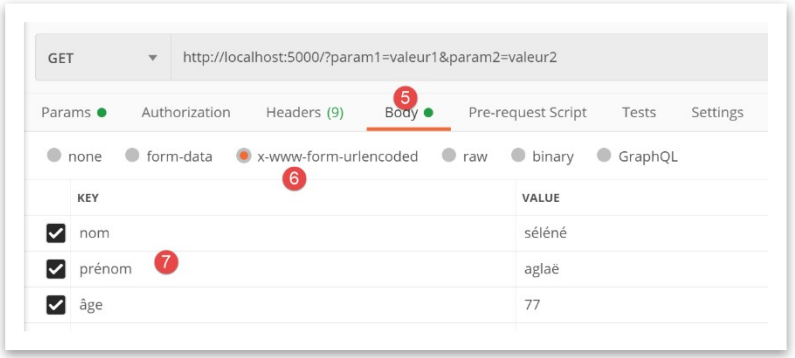
Avec le client Postman, nous envoyons la requête suivante au service web précédent :



- en **[1-2]**, la requête envoyée ;
- en **[2]**, la requête est paramétrée. Les paramètres sont accolés à l'URL sous la forme **[?param1=valeur1&param2=valeur2]**. Il y a deux façons de saisir ces paramètres dans Postman :
  - les écrire directement dans l'URL ;
  - les écrire en **[3-4]** ;

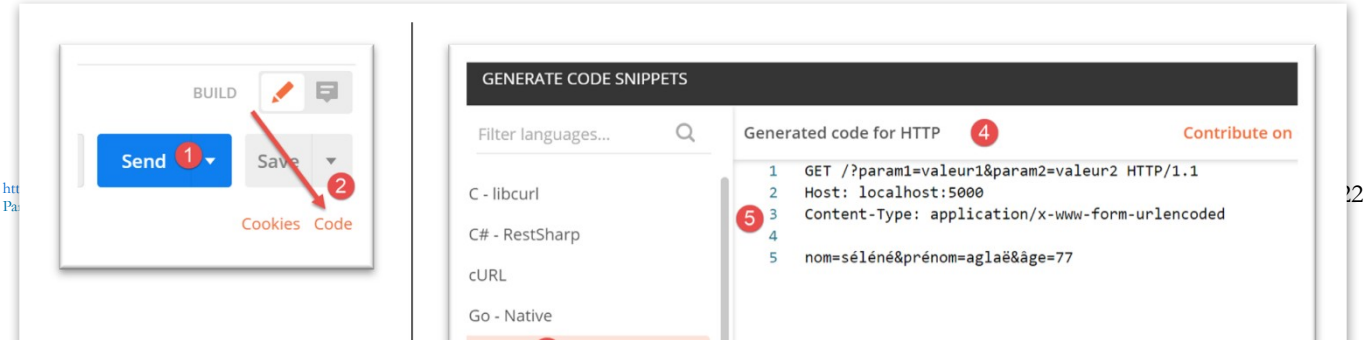
Les deux méthodes sont équivalentes ;

Nous ajoutons d'autres paramètres à la requête :

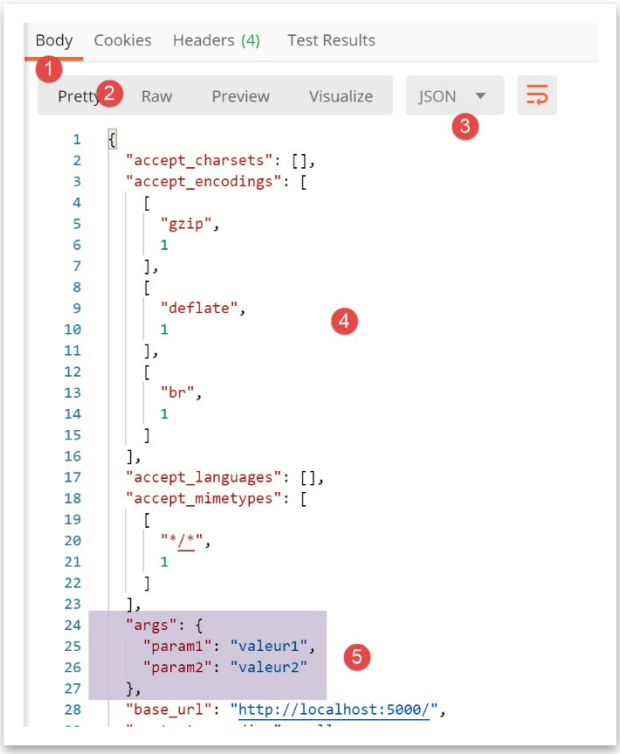


- en **[5-7]**, nous ajoutons des paramètres dans le corps (=body) de le requête. Alors que les paramètres de l'URL sont visibles par l'utilisateur d'un navigateur web, ceux qui font partie du corps de la requête ne sont pas visibles. Le navigateur (ou Postman ici) les envoie au serveur après les entêtes HTTP. La requête du client web a alors la même structure que la réponse du serveur web : des entêtes HTTP suivis par un document. Cela va faire apparaître deux nouveaux entêtes HTTP dans la requête du client :
  - **[Content-Type]** : le client dit au serveur quel type de document il envoie ;
  - **[Content-Length]** : la taille du document en octets ;
- en **[6]**, le codage à employer pour les paramètres déclarés en **[7]**. Ceux-ci peuvent être codés de diverses façons. **[x-www-form-urlencoded]** est une méthode utilisée fréquemment par les navigateurs ;

On peut voir la requête qui va être générée :

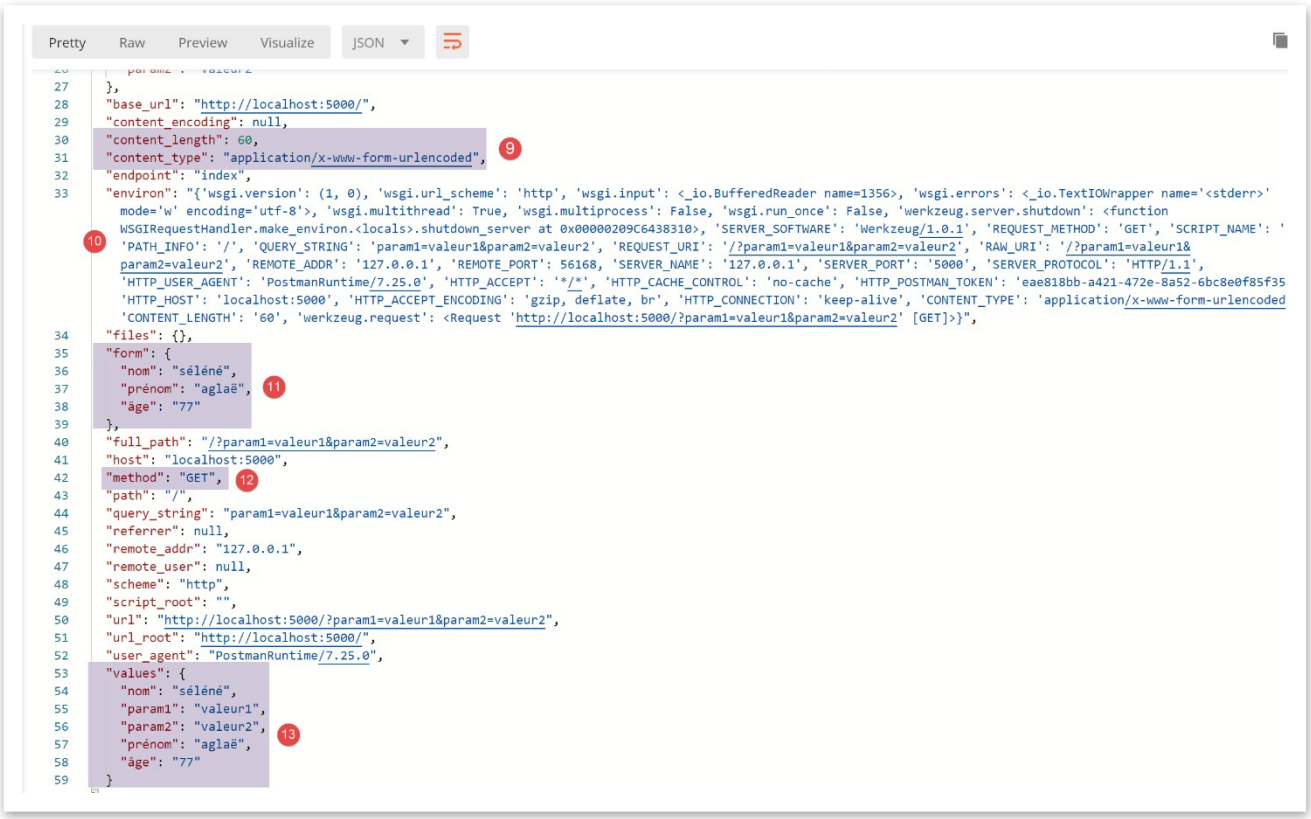


La réponse à cette requête est la suivante :



- en **[1-5]**, on a reçu une chaîne jSON **[3]** ;
- ce qui généralement intéresse le service web, ce sont les paramètres de l'URL **[?param1=valeur1&param2=valeur2]** et ceux qui ont été transmis dans le corps de la requête (document). C'est comme ça, en général, que le client lui transmet des informations. On voit en **[5]** que les paramètres de l'URL sont disponibles dans **[request.args]** ;

Le reste de la réponse est le suivant :



- en **[9]**, les attributs des paramètres mis dans le corps de la requête :

    - **[content_type]** est le type du document accompagnant la requête. On a vu que ce document contenait des informations de type **[param=valeur]** encodées sous la forme **[x-www-form-urlencoded]**. Postman a donc généré un entête HTTP **[Content-Type]** indiquant la nature du document ;
    - **[content_length]** est la taille en octets de ce document ;
- en **[10]**, l'attribut **[request.environ]** contient de nombreuses informations sur l'environnement dans lequel la requête du client est traitée. La plupart de ces informations se retrouvent dans les autres attributs de l'objet **[request]** ;
- en **[11]**, les paramètres présents dans le corps de la requête sont disponibles dans l'attribut **[request.form]** ;
- en **[12]**, la méthode utilisée pour envoyer la requête, ici la méthode **[GET]** ;
- en **[13]**, l'attribut **[request.values]** est le dictionnaire de tous les paramètres, ceux de l'URL et ceux du corps du document. Pour obtenir les paramètres de la requête, on utilisera l'attribut :
    - **[request.args]** pour avoir ceux présents dans l'URL ;
    - **[request.form]** pour avoir ceux présents dans le corps du document ;

Dans la console Postman, les logs sont les suivants :

Requête du client :

```
1.  GET /?param1=valeur1&param2=valeur2 HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: cbfac6aa-71a0-4076-a0c3-91d36d74a4c0
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
9.  Content-Type: application/x-www-form-urlencoded
10. Content-Length: 60
11.
12. nom=s%C3%A9l%C3%A9n%C3%A9&pr%C3%A9nom=agla%C3%AB&%C3%A2ge=77
```

- ligne 9 : le type du document envoyé ligne 12 au serveur ;
- ligne 11 : les entêtes HTTP de la requête sont séparés du document envoyé par une ligne vide. C'est comme cela que le serveur repère la fin des entêtes HTTP du client ;
- ligne 12 : le document 'url-encodé'. Tous les caractères accentués ont subi un encodage ;

La réponse du client est la suivante :

```
1.  HTTP/1.0 200 OK
2.  Content-Type: application/json; charset=utf-8
3.  Content-Length: 2433
4.  Server: Werkzeug/1.0.1 Python/3.8.1
5.  Date: Wed, 15 Jul 2020 06:09:09 GMT
6.
7.  {
8.    "accept_charsets": [],
9.    "accept_encodings": [
10.     [
11.       "gzip",
12.       1
13.     ],
14.     [
15.       "deflate",
16.       1
17.     ],
18.     [
19.       "br",
20.       1
21.     ]
22.   ],
23.   "accept_languages": [],
24.   "accept_mimetypes": [
25.     [
26.       "*/*",
27.       1
28.     ]
29.   ],
30.   "args": {
31.     "param1": "valeur1",
32.     "param2": "valeur2"
33.   },
34.   "base_url": "http://localhost:5000/",
35.   "content_encoding": null,
36.   "content_length": 60,
```

```
37.   "content_type": "application/x-www-form-urlencoded",
38.   "endpoint": "index",
39.   "environ": "{'wsgi.version': (1, 0), 'wsgi.url_scheme': 'http', 'wsgi.input': <_io.BufferedReader
    name=908>, 'wsgi.errors': <_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>,
    'wsgi.multithread': True, 'wsgi.multiprocess': False, 'wsgi.run_once': False, 'werkzeug.server.shutdown':
    <function WSGIRequestHandler.make_environ.<locals>.shutdown_server at 0x00000173CA6E5160>,
    'SERVER_SOFTWARE': 'Werkzeug/1.0.1', 'REQUEST_METHOD': 'GET', 'SCRIPT_NAME': '', 'PATH_INFO': '/',
    'QUERY_STRING': 'param1=valeur1&param2=valeur2', 'REQUEST_URI': '/?param1=valeur1&param2=valeur2',
    'RAW_URI': '/?param1=valeur1&param2=valeur2', 'REMOTE_ADDR': '127.0.0.1', 'REMOTE_PORT': 50592,
    'SERVER_NAME': '127.0.0.1', 'SERVER_PORT': '5000', 'SERVER_PROTOCOL': 'HTTP/1.1', 'HTTP_USER_AGENT':
    'PostmanRuntime/7.26.1', 'HTTP_ACCEPT': '*/*', 'HTTP_CACHE_CONTROL': 'no-cache', 'HTTP_POSTMAN_TOKEN':
    'cbfac6aa-71a0-4076-a0c3-91d36d74a4c0', 'HTTP_HOST': 'localhost:5000', 'HTTP_ACCEPT_ENCODING': 'gzip,
    deflate, br', 'HTTP_CONNECTION': 'keep-alive', 'CONTENT_TYPE': 'application/x-www-form-urlencoded',
    'CONTENT_LENGTH': '60', 'werkzeug.request': <Request 'http://localhost:5000/?param1=valeur1&param2=valeur2'
    [GET]>}",
40.   "files": {},
41.   "form": {
42.     "nom": "s\u00e9l\u00e9n\u00e9",
43.     "pr\u00e9nom": "agla\u00eb",
44.     "\u00e2ge": "77"
45.   },
46.   "full_path": "/?param1=valeur1&param2=valeur2",
47.   "host": "localhost:5000",
48.   "method": "GET",
49.   "path": "/",
50.   "query_string": "param1=valeur1&param2=valeur2",
51.   "referrer": null,
52.   "remote_addr": "127.0.0.1",
53.   "remote_user": null,
54.   "scheme": "http",
55.   "script_root": "",
56.   "url": "http://localhost:5000/?param1=valeur1&param2=valeur2",
57.   "url_root": "http://localhost:5000/",
58.   "user_agent": "PostmanRuntime/7.26.1",
59.   "values": {
60.     "nom": "s\u00e9l\u00e9n\u00e9",
61.     "param1": "valeur1",
62.     "param2": "valeur2",
63.     "pr\u00e9nom": "agla\u00eb",
64.     "\u00e2ge": "77"
65.   }
66. }
```

- lignes 1-5 : les entêtes HTTP de la réponse terminés par une ligne vide ;
- lignes 41-45 : les éléments accentués ont subi un encodage UTF-8 ;

Si maintenant on utilise la méthode **[POST]** pour envoyer la même requête avec les mêmes paramètres, on obtiendra la même réponse si ce n'est qu'en **[12]**, on aura **['method' : 'POST']**.

Aussi quelle est la différence entre les méthodes GET et POST ? La différence est mince et a été instituée par l'usage qu'en ont fait historiquement les navigateurs :

- les paramètres dans l'URL sont pratiques parce qu'une URL ainsi paramétrée peut servir de lien dans un document HTML. L'utilisateur peut également changer les paramètres lui-même pour obtenir des réponses différentes du serveur. Dans ce cas, les navigateurs utilisent couramment la méthode **[GET]** et il n'y a pas de corps (content_length=0) dans la requête envoyée au serveur web (pas de paramètres cachés) ;
- parfois on ne veut pas que les paramètres soient affichés dans l'URL. C'est le cas des mots de passe envoyés au serveur. Par ailleurs, la taille occupée par les paramètres de l'URL est limitée (une URL ne peut dépasser une certaine taille). Les paramètres du corps de la requête n'ont pas cette limitation. Egalement, beaucoup de paramètres dans l'URL la rendent illisible. Prenons le cas courant d'un formulaire d'inscription à un site web. Historiquement, lorsque les pages HTML n'embarquaient pas encore du Javascript, les navigateurs envoyaient les renseignements saisis par un POST. On parlait alors de valeurs postées ;

Donc au début de la programmation web :

- les méthodes GET étaient plutôt associées à la demande d'informations délivrées par un serveur web ;
- les méthodes POST étaient plutôt associées à l'envoi d'informations du navigateur vers le serveur. Le serveur était alors 'enrichi' par celles-ci ;

Depuis Javascript est passé par là. Alors que dans les exemples précédents, le développeur n'avait pas la main (cliquer sur un lien déclenchait forcément un GET, valider un formulaire passait forcément par un POST), le Javascript leur a redonné la main. Dans ce modèle, la page HTML est associée à du code Javascript qui peut court-circuiter le navigateur. Ainsi le clic sur un lien peut-il être

intercepté par le code Javascript qui peut ensuite exécuter un code faisant une requête au serveur. Cette requête sera transparente pour l'utilisateur. Il ne la verra pas. Ce code est un client web et comme nous l'avons fait avec Postman le développeur peut créer la requête qu'il veut. Pour revenir sur le clic sur un lien, il peut réaliser un POST alors que par défaut le navigateur aurait réalisé un GET. Ces évolutions ont rendu les différences entre GET et POST moins pertinentes.

Cependant, les développeurs adoptent souvent les règles suivantes :

- un GET ne doit pas modifier l'état du serveur. Des GET successifs réalisés avec les mêmes paramètres dans l'URL divent ramener le même document. De plus le GET n'a le plus souvent pas de corps (pas de document associé), seulement des paramètres dans l'URL ;
- le POST peut modifier l'état du serveur. Les paramètres sont le plus souvent envoyés dans le corps de la requête. On parle alors de valeurs postées. L'exemple du formulaire est le plus éloquent : les valeurs saisies par l'utilisateur vont être mises dans le corps du POST et le serveur va les enregistrer quelque part, souvent une base de données ;

Dans la suite du document, nous ne nous astreignons à respecter aucune règle particulière.

22.6 scripts `[flask-05]` : gestion de la mémoire de l'utilisateur

22.6.1 Introduction

Dans les exemples client / serveur précédents on avait le fonctionnement suivant :

- le client ouvre une connexion vers le port 80 de la machine du service web ;
- il envoie la séquence de texte : en-têtes HTTP, ligne vide, **`[document] ;`**
- en réponse, le serveur envoie une séquence du même type ;
- le serveur clôt la connexion vers le client ;
- le client clôt la connexion vers le serveur ;

Si le même client fait peu après une nouvelle demande au serveur web, une nouvelle connexion est créée entre le client et le serveur. Celui-ci ne peut pas savoir si le client qui se connecte est déjà venu ou si c'est une première demande. Entre deux connexions, le serveur "oublie" son client. Pour cette raison, on dit que le protocole HTTP est un **protocole sans état**. Il est pourtant utile que le serveur se souvienne de ses clients. Ainsi si une application est sécurisée, le client va envoyer au serveur un login et un mot de passe pour s'identifier. Si le serveur "oublie" son client entre deux connexions, celui-ci devra s'identifier à chaque nouvelle connexion, ce qui n'est pas envisageable.

Pour faire le suivi d'un client, le serveur peut procéder de diverses façons :

1. lors d'une première demande d'un client, il inclut dans sa réponse un identifiant que le client doit ensuite lui renvoyer à chaque nouvelle demande. Grâce à cet identifiant, différent pour chaque client, le serveur peut reconnaître un client. Il peut alors gérer une mémoire pour ce client sous la forme d'une mémoire associée de façon unique à l'identifiant du client. C'est ainsi que fonctionnent, par exemple, les services PHP ;
2. lors d'une première demande d'un client, il inclut dans sa réponse non pas un identifiant mais la mémoire de l'utilisateur elle-même. Il ne garde rien côté serveur. Pour maintenir sa mémoire, le client web doit renvoyer cette mémoire à chaque nouvelle requête. Celle-ci est modifiée (ou non) à chaque nouvelle requête et renvoyée (ou non) au client. C'est la méthode utilisée par le framework Flask ;

Les différences entre les deux méthodes sont les suivantes :

- la méthode 1 est moins gourmande en bande passante. Seul est échangé entre le client et le serveur un identifiant. Lorsque la mémoire de l'utilisateur grandit, cela n'a aucune conséquence sur l'identifiant qui reste le même. Ce n'est pas le cas de la méthode 2 où la mémoire de l'utilisateur est échangée à chaque requête et peut grossir au fil des requêtes ;
- la méthode 1 est plus gourmande en espace mémoire. En effet, le serveur stocke la mémoire de l'utilisateur sur ses systèmes de fichiers. S'il y a un million d'utilisateurs, cela peut peut-être poser un problème. La méthode 2 ne stocke rien sur le serveur ;

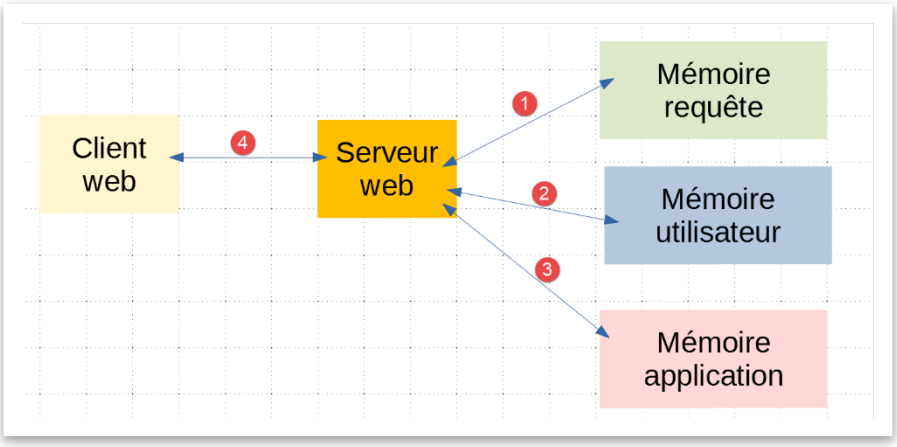
Techniquement cela se passe ainsi dans les deux méthodes :

- dans la réponse à un nouveau client, le serveur inclut l'en-tête HTTP **`[Set-Cookie : MotClé=Identifiant]`** ou **`[Set-Cookie : mémoire]`**. Avec la méthode 1, il ne fait cela qu'à la première demande. Avec la méthode 2, il le fait à chaque fois que la mémoire de l'utilisateur change ;
- dans ses demandes, le client renvoie systématiquement ce qu'il a reçu, un identifiant ou une mémoire. Il le fait via l'en-tête HTTP **`[Cookie : MotClé=Valeur]`** ;

On peut se demander comment le serveur fait pour savoir qu'il a affaire à un nouveau client plutôt qu'à un client déjà venu. C'est la présence de l'en-tête HTTP *Cookie* dans les en-têtes HTTP du client qui le lui indique. Pour un nouveau client, cet en-tête est absent.

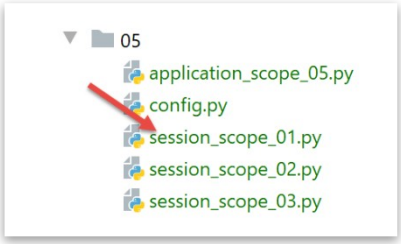
L'ensemble des connexions d'un client donné est appelé une **session**.

Le serveur peut maintenir d'autres types de mémoire :



- en **[1]**, la mémoire de la requête est particulière. Elle est utilisée lorsque la demande du client web est traitée non pas par un service (ou application) mais par plusieurs. Pour passer des informations au service i+1, le service i peut enrichir la requête traitée (request) avec ces informations. C'est ce qu'on appelle la mémoire de niveau requête. Nous n'utiliserons pas ce type de mémoire dans ce document ;
- en **[2, 4]**, la mémoire de l'utilisateur que nous venons de décrire. Elle peut être implémentée localement **[2]** ou maintenue à l'aide du client **[4]** ;
- en **[3]**, la mémoire de niveau 'application' est très généralement une mémoire en lecture seule. Elle est partagée par tous les utilisateurs. On y retrouve souvent des éléments de la configuration de l'application web, configuration partagée par tous les utilisateurs de l'application. On doit être prudents avec ce type de mémoire : l'écriture dedans doit se faire à un moment où les utilisateurs n'ont pas encore envoyé de requêtes, au démarrage de l'application le plus souvent. Ensuite, lorsque les requêtes arrivent, il est difficile d'écrire dans cette mémoire. Lorsque le serveur web sert simultanément plusieurs utilisateurs et que deux d'entre-eux veulent écrire dans la mémoire de niveau 'application', il y a un risque que cette mémoire soit corrompue. En effet, alors que l'utilisateur 1 a commencé à écrire dans la mémoire de niveau 'application', il peut être interrompu avant même d'avoir fini. On a alors une mémoire d'application incomplète. Comme elle est partagée, un utilisateur 2 peut la lire et obtenir un état incorrect ;

22.6.2 script `[session_scope_01]`



Les scripts **[session_scope_xx]** illustrent la gestion des mémoires utilisateurs.

Le script **[session_scope_01]** est le suivant :

```
1.  # on configure l'application
2.  import config
3.  config = config.configure()
4.
5.  # dépendances
6.  import json
7.  from flask import Flask, make_response, session
8.  from flask_api import status
9.
10. # application Flask
```

```
11. app = Flask(__name__)
12.
13. # clé secrète de la session
14. app.secret_key = config["SECRET_KEY"]
15.
16.
17. @app.route('/set-session', methods=['GET'])
18. def set_session():
19.     # on met qq chose dans la session
20.     session['nom'] = 'séléné'
21.     # on envoie une réponse vide
22.     response = make_response()
23.     response.headers['Content-Length'] = 0
24.     return response, status.HTTP_200_OK
25.
26.
27. @app.route('/get-session', methods=['GET'])
28. def get_session():
29.     # on récupère la session et on envoie la réponse
30.     response = make_response(json.dumps({"nom": session['nom']}, ensure_ascii=False))
31.     response.headers['Content-Type'] = 'application/json; charset=utf-8'
32.     return response, status.HTTP_200_OK
33.
34.
35. # main seulement
36. if __name__ == '__main__':
37.     app.config.update(ENV="development", DEBUG=True)
38.     app.run()
```

- ligne 11 : une application Flask est instanciée ;
- ligne 14 : l'attribut **[secret_key]** de cette application reçoit une valeur prise dans le fichier de configuration exploité aux lignes 1-3. Une session Flask n'est possible que si cet attribut est initialisé. On peut mettre n'importe quoi dedans. Il sert à crypter une partie de la 'mémoire utilisateur' qui sera envoyée au client. On met en général quelque chose difficile à deviner. Dans le fichier **[config]**, la clé secrète est définie de la façon suivante :

```
1.      # on rend la config
2.      config = {
3.          # configuration Flask
4.          "SECRET_KEY": "vibnFfrdWYUp?*LQ"
5.      }
```

- pour la première fois, nous définissons une application web qui sert autre chose que l'URL /
  - ligne 17 : l'URL **[/set-session]** sert à initialiser la session de l'utilisateur ;
  - ligne 27 : l'URL **[/get-session]** sert à retrouver la mémoire de l'utilisateur (ou session de l'utilisateur) ;
- ligne 20 : on met quelque chose dans la mémoire (= la session) de l'utilisateur, ici un nom. La session se gère un peu comme un dictionnaire. On ne peut pas mettre n'importe quoi dans la session. Il faut que les valeurs qu'on y met puissent être transformées en jSON. Pour les types prédéfinis de Python, cela se fait sans intervention du développeur. Pour des objets propriétaires que Python ne connaît pas, il faut faire soi-même la conversion jSON ;
- ligne 22 : on crée une réponse HTTP sans contenu (absence de paramètre à make_response) ;
- ligne 23 : on dit au client qu'il va recevoir un document vide (taille de 0 octet) ;
- ligne 24 : on envoie la réponse HTTP au client. L'URL **[/set-session]** ne fait donc rien d'autres que d'initialiser une session utilisateur ;
- ligne 27 : l'URL **[/get-session]** permet à l'utilisateur de savoir ce qu'il y a dans sa session ;
- ligne 30 : on crée une réponse HTTP contenant la chaîne jSON de la session de l'utilisateur. Ici on a créé la chaîne jSON nous-mêmes au lieu de laisser Flask la générer. En effet, on ne veut pas que les caractères accentués soient échappés (`ensure_ascii=False`) ;
- ligne 31 : on dit au client qu'on lui envoie du jSON ;
- ligne 32 : on envoie la réponse HTTP au client ;

Le but de ce script est de montrer que la session utilisateur permet de faire le lien entre les requêtes successives de celui-ci :

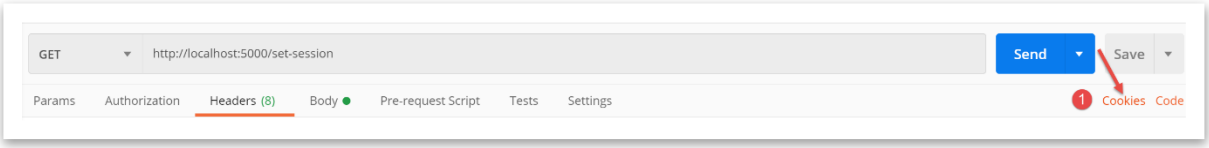
- la requête 1 demandera l'URL **[/set-session]** ;
- la requête 2 demandera l'URL **[/get-session]** et va récupérer le nom que la requête 1 aura initialisé ;

Le script **[config]** qui configure les scripts du dossier **[flask/05]** est le suivant :

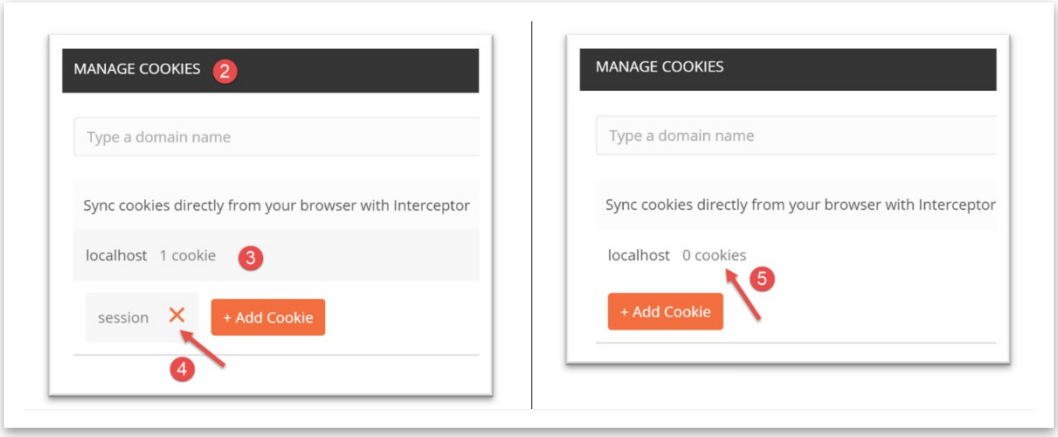
```
1.  def configure():
2.      # chemin absolu référence des chemins relatifs de la configuration
3.      root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
4.
5.      # dépendances de l'application
```

```
6.      absolute_dependencies = [
7.          # Personne, Utils, MyException
8.          f"{root_dir}/classes/02/entities",
9.      ]
10.     # on fixe le syspath
11.     from myutils import set_syspath
12.     set_syspath(absolute_dependencies)
13.
14.     # on rend la config
15.     config = {
16.         # configuration Flask
17.         "SECRET_KEY": "vibnFfrdWYUp?*LQ"
18.     }
19.
20.     return config
```

Nous lançons le script **[session_scope_01]** puis avec Postman nous allons demander l'URL **[/set-session]**. Auparavant nous allons vérifier quelques éléments de la requête qui va être faite :

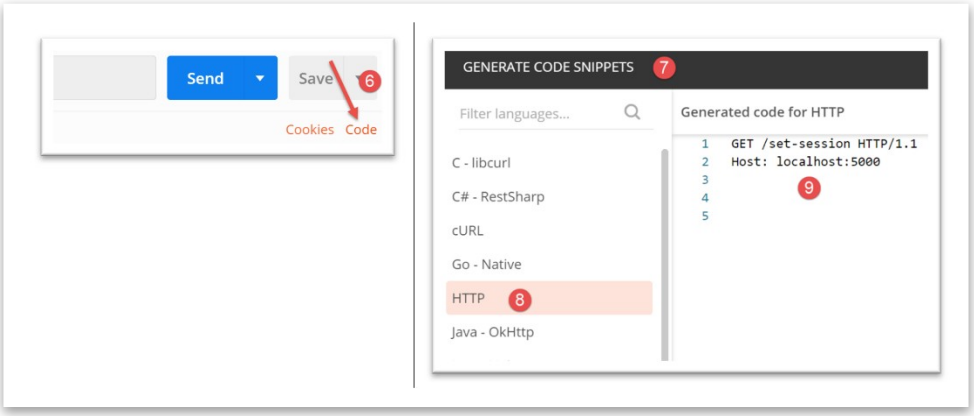


- en **[1]**, accéder aux cookies de Postman ;



- en **[2-4]**, on vérifie les cookies connus de Postman et on les supprime tous **[4-5]** ;

Maintenant vérifions la requête HTTP qui va être générée :



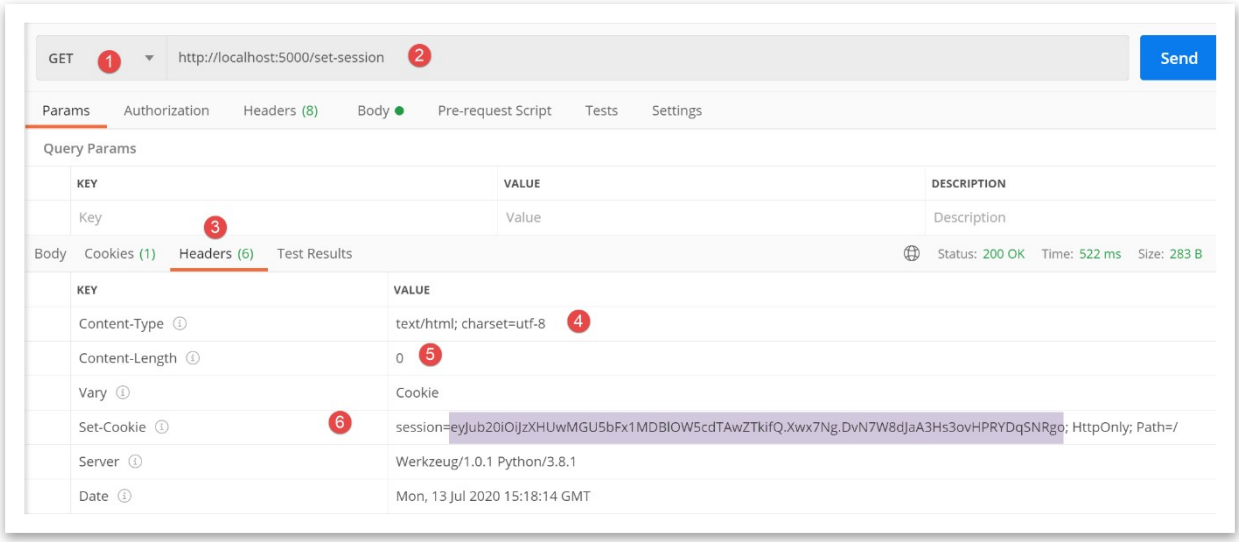
- en **[9]** : une partie des entêtes HTTP que Postman va mettre dans la requête à partir de la configuration que nous avons pu faire pour celle-ci. Cette vérification vous permet de vérifier que vous n'avez pas omis des paramètres ou au contraire laissé des paramètres inutiles ;

Ceci fait, on peut exécuter la requête :



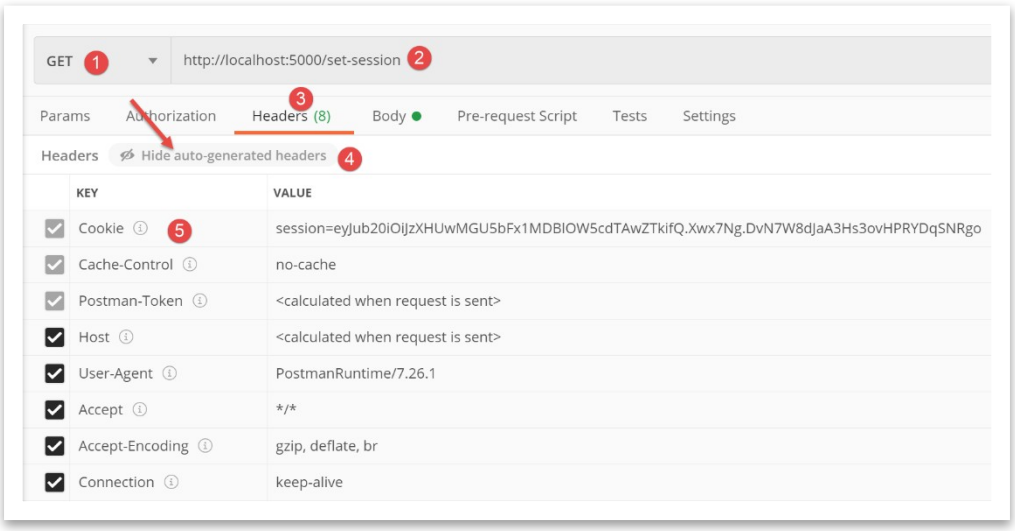
https://tahe.developpez.com/ • Ce cours tutoriel écrit par Serge Tahé est mis à disposition du public selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Partage dans les Mêmes Conditions 3.0 non transposé.

Il y a différentes façons de vérifier le résultat. On peut déjà regarder la fenêtre principale :



- en **[1-2]**, la requête faite au service web ;
- en **[3-6]**, les entêtes HTTP de la réponse ;
- en **[4]**, comme dans le code on n'a pas précisé le type de la réponse, Flask a utilisé par défaut le type **[text/html]** ;
- en **[5]**, le client sait qu'il n'y a pas de document dans la réponse ;
- ligne 6 : l'entête **[Set-Cookie]** a été envoyé par le serveur Flask. Sa valeur est appelée un **cookie de session**. Elle est constituée de trois éléments :
  - **[session=valeur]** : valeur représente la mémoire de l'utilisateur sous une forme codée. Cette mémoire est décodable (cf. |https://blog.miguelgrinberg.com/post/how-secure-is-the-flask-user-session|). Néanmoins à cause de la clé secrète utilisée par le serveur, l'utilisateur ne peut pas modifier la mémoire reçue pour la renvoyer ensuite au serveur. Lorsque le serveur reçoit une session, il est ainsi assuré de recevoir une session non corrompue ;
  - **[HttpOnly]** : la présence de cet élément indique au navigateur qui le reçoit que le cookie ne doit pas être accessible au Javascript que peut contenir la page qu'il affiche ;
  - **[Path=/]** est le chemin pour lequel il faut renvoyer le cookie de session donc ici tout chemin de l'application web. A chaque fois que l'utilisateur au clavier demandera explicitement (il tape une URL) ou implicitement (il clique sur un lien) une URL de ce domaine, le navigateur renverra **automatiquement** le cookie de session qu'il a reçu ;

L'inconvénient de la fenêtre principale est qu'on n'a pas accès à la requête complète qui a amené à cette réponse. Ce qui est présenté dans cette fenêtre prête à confusion :



- dans les entêtes HTTP **[3-4]** est présenté en **[5]** un cookie de session. On pourrait croire alors que Postman a mis dans la requête un cookie de session alors que ce n'est pas le cas. Les entêtes **[3]** représentent en fait les entêtes HTTP qui seront envoyés lors de la prochaine requête telle que celle-ci est actuellement configurée. Postman vient de recevoir un cookie de session qu'il renverra lors de la prochaine requête. C'est pourquoi on a **[5]** ;

On peut avoir accès au dialogue client / serveur dans la console Postman qu'on obtient avec Ctrl-Alt-C :

```
1.  GET /set-session HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: 3673b73f-7600-4df4-8c4b-c37973e50df8
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
9.
10. HTTP/1.0 200 OK
11. Content-Type: text/html; charset=utf-8
12. Content-Length: 0
13. Vary: Cookie
14. Set-Cookie: session=eyJub20iOiJzXHUwMGU5bFx1MDBlOW5cdTAwZTkifQ.Xw6jGQ.y5Icu70wTIN-B0o_hwx0xDH247I;
    HttpOnly; Path=/
15. Server: Werkzeug/1.0.1 Python/3.8.1
16. Date: Wed, 15 Jul 2020 06:32:57 GMT
```

- ligne 14 : le cookie de session envoyé par le serveur ;

Maintenant demandons l'URL **[/get-session]** :

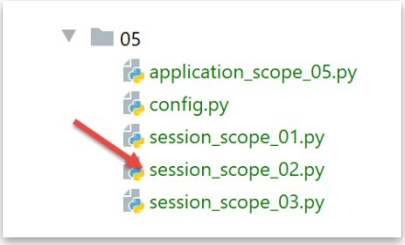
```
1.  GET /get-session HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: ce991398-2d9a-46d0-9ccd-c7ff3c7f4d6d
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
9.  Cookie: session=eyJub20iOiJzXHUwMGU5bFx1MDBlOW5cdTAwZTkifQ.Xw6jGQ.y5Icu70wTIN-B0o_hwx0xDH247I
10.
11. HTTP/1.0 200 OK
12. Content-Type: application/json; charset=utf-8
13. Content-Length: 20
14. Vary: Cookie
15. Server: Werkzeug/1.0.1 Python/3.8.1
16. Date: Wed, 15 Jul 2020 06:36:52 GMT
17.
18. {"nom": "séléné"}
```

- ligne 9 : le client Postman a renvoyé au serveur le cookie de session qu'il avait reçu ;
- ligne 18 : la chaîne jSON envoyée par le serveur ;

Cet exemple nous montre divers points :

- le client Postman renvoie le cookie de session qu'il reçoit du serveur Flask. Les navigateurs web procèdent toujours ainsi ;
- nous voyons que la requête 2 **[/get-session]** a permis de récupérer une information créée lors de la requête 1 **[/set-session]**. On a donc là une mémoire de l'utilisateur ;
- lignes 11-16 : le serveur Flask n'a pas renvoyé de cookie de session. Ce n'est pas systématique. Le serveur Flask ne renvoie le cookie de session que si la dernière requête a modifié la mémoire de l'utilisateur ;

22.6.3 script [session_scope_02]



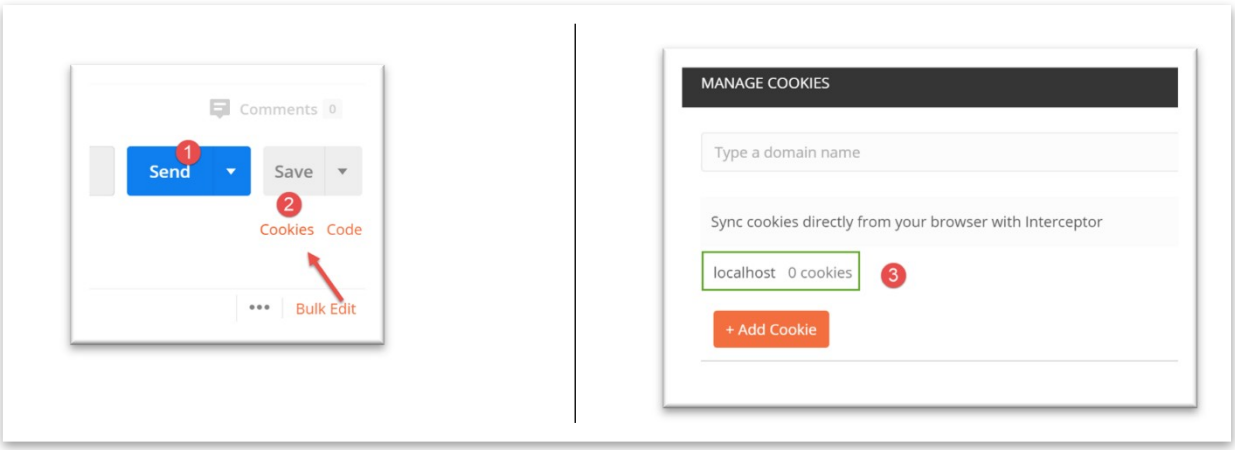
Le script **[session_02]** est le suivant :

```
1.  # dépendances
2.  import os
```

```
3.
4.  from flask import Flask, make_response, session
5.  from flask_api import status
6.
7.  # application Flask
8.  app = Flask(__name__)
9.
10. # clé secrète de la session
11. app.secret_key = os.urandom(12).hex()
12.
13.
14. # Home URL
15. @app.route('/', methods=['GET'])
16. def index():
17.     # on gère tois compteurs
18.     if session.get('n1') is None:
19.         session['n1'] = 0
20.     else:
21.         session['n1'] = session['n1'] + 1
22.     if session.get('n2') is None:
23.         session['n2'] = 10
24.     else:
25.         session['n2'] = session['n2'] + 1
26.     if session.get('n3') is None:
27.         session['n3'] = 100
28.     else:
29.         session['n3'] = session['n3'] + 1
30.     # dictionnaire des compteurs
31.     compteurs = {"n1": session['n1'], "n2": session['n2'], "n3": session['n3']}
32.     # on envoie la réponse
33.     response = make_response(compteurs)
34.     response.headers['Content-Type'] = 'application/json; charset=utf-8'
35.     return response, status.HTTP_200_OK
36.
37.
38. # main
39. if __name__ == '__main__':
40.     app.config.update(ENV="development", DEBUG=True)
41.     app.run()
```

- ligne 11 : ici la clé secrète est générée à l'aide d'une fonction. L'intérêt de celle-ci est qu'elle génère une chaîne de caractères complexe de façon aléatoire. On rappelle que la variable **[app]** est l'instance de classe Flask créée ligne 8 ;
- ligne 15 : cette fois-ci, il n'y aura qu'une route, la route / ;
- lignes 17-29 : on gère une session contenant trois compteurs **[n1, n2, n3]**. Lors du 1er appel de l'utilisateur **[n1, n2, n3]=[0, 10, 100]** puis à chaque appel ces compteurs sont incrémentés de 1 ;
- ligne 18 : lors de la 1ère requête, la session de l'application est vide. L'expression **[session.get('clé')]** rend la valeur **[None]**. Pour les requêtes suivantes, cette expression rendra la valeur associée à la clé ;
- ligne 31 : ces compteurs sont mis dans un dictionnaire ;
- ligne 33 : ce dictionnaire est le document de la réponse HTTP. On rappelle, que Flask transforme automatiquement les dictionnaires en chaîne jSON ;
- ligne 34 : on dit au client web qu'il va recevoir du jSON ;
- ligne 35 : on envoie la réponse HTTP au client ;

Exécutons ce script et interrogeons l'application web ainsi créée avec Postman après avoir supprimé tous les cookies du client Postman **[1-3]** :



Dans la console Postman, les échanges client / serveur sont les suivants :

```
1.  GET / HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: c7db536d-9352-4aa6-9877-04560e03d935
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
9.
10. HTTP/1.0 200 OK
11. Content-Type: application/json; charset=utf-8
12. Content-Length: 41
13. Vary: Cookie
14. Set-Cookie: session=eyJuMSI6MCwibjIiOjEwLCJuMyI6MTAwfQ.Xw6nLg.v49CeDWwqP-6Dp9Qt330GAe-dNA; HttpOnly; Path=/
15. Server: Werkzeug/1.0.1 Python/3.8.1
16. Date: Wed, 15 Jul 2020 06:50:22 GMT
17.
18. {
19. "n1": 0,
20. "n2": 10,
21. "n3": 100
22. }
```

- en **[14]**, le cookie de session envoyé par le serveur ;
- en **[18-22]**, la réponse du serveur sous la forme d'une chaîne jSON ;

Refaisons une deuxième fois la même requête. Les logs évoluent de la façon suivante :

```
1.  GET / HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: 8205ad85-37b3-41f2-a171-70dd3b3a1679
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
9.  Cookie: session=eyJuMSI6MCwibjIiOjEwLCJuMyI6MTAwfQ.Xw6nLg.v49CeDWwqP-6Dp9Qt330GAe-dNA
10.
11. HTTP/1.0 200 OK
12. Content-Type: application/json; charset=utf-8
13. Content-Length: 41
14. Vary: Cookie
15. Set-Cookie: session=eyJuMSI6MSwibjIiOjExLCJuMyI6MTAxfQ.Xw6nsw.OuxIQnGhmhSsan5Qu_FL3Iyu-9k; HttpOnly; Path=/
16. Server: Werkzeug/1.0.1 Python/3.8.1
17. Date: Wed, 15 Jul 2020 06:52:35 GMT
18.
19. {
20. "n1": 1,
21. "n2": 11,
22. "n3": 101
23. }
```

- ligne 9 : le client Postman renvoie le cookie de session qu'il a reçu ;
- ligne 15 : dans sa réponse, le serveur envoie un nouveau cookie de session, ceci parce que la requête du client a modifié la mémoire de l'utilisateur (= la session) ;
- lignes 19-23 : les nouvelles valeurs des compteurs ;

22.6.4 script `[session_scope_03]`

Ce nouveau script vise à montrer qu'on peut mettre différents types Python dans une session : liste, dictionnaire, objet. La seule contrainte est que les objets mis en session soient sérialisables en jSON. S'ils ne le sont pas par défaut (listes, dictionnaires), il faut alors faire soi-même la conversion en jSON.

```
1.  # on configure l'application
2.  import config
3.  config = config.configure()
4.
5.  # dépendances
6.  import json
7.  import os
8.
9.  from flask import Flask, make_response, session
```

```
10. from flask_api import status
11. from Personne import Personne
12.
13. # application Flask
14. app = Flask(__name__)
15.
16. # clé secrète de la session
17. app.secret_key = os.urandom(12).hex()
18.
19.
20. # Home URL
21. @app.route('/', methods=['GET'])
22. def index():
23.     # gestion d'une liste
24.     liste = session.get('liste')
25.     if liste is None:
26.         # 1ère requête
27.         liste = [0, 10, 100]
28.     else:
29.         # requêtes suivantes
30.         for i in range(len(liste)):
31.             liste[i] += 1
32.     # on remet la liste dans la session
33.     session['liste'] = liste
34.
35.     # gestion d'un dictionnaire
36.     dico = session.get('dico')
37.     if not dico:
38.         # 1ère requête
39.         dico = {"un": 0, "deux": 10, "trois": 100}
40.     else:
41.         # requêtes suivantes
42.         dico = session['dico']
43.         for key in dico.keys():
44.             dico[key] += 1
45.     # on remet le dictionnaire dans la session
46.     session['dico'] = dico
47.
48.     # gestion d'une personne
49.     personne_json = session.get('personne')
50.     if personne_json is None:
51.         # 1ère requête
52.         personne = Personne().fromdict({"prénom": "aglaë", "nom": "séléné", "âge": 70})
53.     else:
54.         # requêtes suivantes
55.         personne = Personne().fromjson(personne_json)
56.         personne.âge += 1
57.     # on remet la personne dans la session
58.     session['personne'] = personne.asjson()
59.
60.     # dictionnaire des résultats
61.     résultats = {"liste": liste, "dict": dico, "personne": personne.asdict()}
62.
63.     # on envoie une réponse jSON
64.     response = make_response(json.dumps(résultats, ensure_ascii=False))
65.     response.headers['Content-Type'] = 'application/json; charset=utf-8'
66.     return response, status.HTTP_200_OK
67.
68.
69. # main
70. if __name__ == '__main__':
71.     app.config.update(ENV="development", DEBUG=True)
72.     app.run()
```

- lignes 1-3 : l'application web est configurée ;
- lignes 5-11 : les dépendances sont importées ;
- ligne 14 : l'application Flask est instanciée ;
- ligne 17 : l'attribut **[secret_key]** est initialisé. C'est ce qui permet l'utilisation de sessions ;
- ligne 21 : l'unique route de l'application ;
- lignes 23-33 : gestion d'une liste dans la session. On a mis dans celle-ci des éléments sérialisables par défaut en jSON ;
- lignes 35-46 : gestion d'un dictionnaire dans la session. On a mis dans celui-ci des éléments sérialisables par défaut en jSON ;
- lignes 48-58 : gestion d'une personne. Un objet **[Personne]** n'est pas sérialisable par défaut en jSON. Il faut donc prendre des précautions ;

- ligne 58 : on utilise la méthode **[BaseEntity.asjson]** pour stocker dans la session la chaîne jSON de la personne. Noter qu'on aurait pu utiliser **[personne.asdict]** car **[personne.asdict]** est un dictionnaire contenant des valeurs sérialisables par défaut en jSON ;
- ligne 55 : parce qu'on a stocké une chaîne jSON dans la session, on récupère la personne dans celle-ci en utilisant la méthode **[BaseEntity.fromjson]** ;
- ligne 61 : on crée le dictionnaire **[résultats]** qui sera envoyé comme réponse au client. On sait que dans ce cas là, Flask envoie la chaîne jSON du dictionnaire. Il faut donc que celui-ci ne contienne que des valeurs sérialisables par défaut en jSON ;
- ligne 64 : on met explicitement la chaîne jSON du dictionnaire **[résultats]** dans a réponse HTTP. Flask l'aurait fait par défaut. Seulement, toujours par défaut, il utilise le paramètre **[ensure_ascii=True]**, ce qui ne nous convenait pas ;
- ligne 65 : on dit au client qu'il va recevoir du jSON ;
- ligne 66 : on lui envoie la réponse ;

On lance l'application web. On supprime tous les cookies du client Postman. Puis celui-ci demande l'URL **[http://localhost:5000]**. Le dialogue client / serveur dans la console Postman est le suivant :

```
1.  GET / HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: 5f8b7c63-aa8a-4429-a2fa-62141423d933
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
9.
10. HTTP/1.0 200 OK
11. Content-Type: application/json; charset=utf-8
12. Content-Length: 135
13. Vary: Cookie
14. Set-Cookie: session=.eJw9isEKwyAQRH-lzHkPm15K91dqD2mzBMFq0AgF8d-
    jsRQG9u3MK1jsO0AKFs1fyMSEPQabOjbOHsKV4GzaFfJgmnr4Sdg0puB9a1EMtmgys959-
    BjIxWBe3XxWLwNq_39IQ3Q_f5zhnHxdtYs3rqgH4gQvMg.Xw6yGw.Bwpt3q-sH03gFLmg2FIPXV_ZNt8; HttpOnly; Path=/
15. Server: Werkzeug/1.0.1 Python/3.8.1
16. Date: Wed, 15 Jul 2020 07:36:59 GMT
17.
18. {"liste": [0, 10, 100], "dict": {"un": 0, "deux": 10, "trois": 100}, "personne": {"prénom": "aglaë", "nom":
    "séléné", "âge": 70}}
```

Nous faisons la requête une seconde fois :

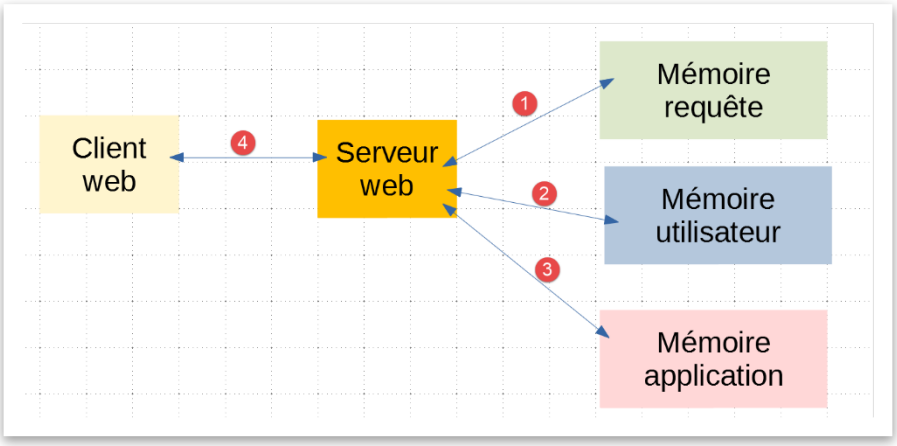
```
1.  GET / HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: 40fd00ea-d45c-46b7-a51e-d4d433a37b5c
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
9.  Cookie: session=.eJw9isEKwyAQRH-lzHkPm15K91dqD2mzBMFq0AgF8d-
    jsRQG9u3MK1jsO0AKFs1fyMSEPQabOjbOHsKV4GzaFfJgmnr4Sdg0puB9a1EMtmgys959-
    BjIxWBe3XxWLwNq_39IQ3Q_f5zhnHxdtYs3rqgH4gQvMg.Xw6yGw.Bwpt3q-sH03gFLmg2FIPXV_ZNt8
10.
11. HTTP/1.0 200 OK
12. Content-Type: application/json; charset=utf-8
13. Content-Length: 135
14. Vary: Cookie
15. Set-Cookie: session=.eJw9isEKwyAQRH-lzHkP2kupv9LtIW2WIBgNGqEg_nu3seQ0b2Zew-zfCa5hlvqBs5aw5-
    SLolGuUaETgi-7wD0sqaHPk7BJLilGXdEYW-ZqjNxjWhnuwpiWMB3Ti0Haz6MMMfz9EcM5-
    LrIT7zZjv4F5NYvOQ.Xw6ydQ.PMWRCqKx9HNnb_DyK-ha-9pCF7M; HttpOnly; Path=/
16. Server: Werkzeug/1.0.1 Python/3.8.1
17. Date: Wed, 15 Jul 2020 07:38:29 GMT
18.
19. {"liste": [1, 11, 101], "dict": {"deux": 11, "trois": 101, "un": 1}, "personne": {"prénom": "aglaë", "nom":
    "séléné", "âge": 71}}
```

- ligne 9 : le client renvoie le cookie de session qu'il a reçu ;
- ligne 15 : le serveur lui en renvoie un autre car le contenu de la session a changé (ligne 19). On rappelle que ce contenu est présent dans le cookie de session sous forme codée ;

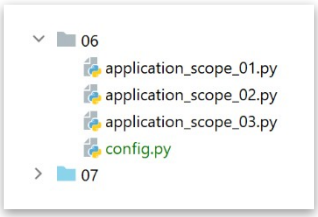
22.7 scripts [flask/06] : informations partagées par tous les utilisateurs

22.7.1 Introduction

Cette section vise à montrer comment gérer des informations de portée application, ç-à-d partagées par tous les utilisateurs. Ces informations sont typiquement des informations de configuration de l'application. Nous avons vu qu'une application web pouvait maintenir différents types de mémoire :



Nous nous intéressons ici à la mémoire de l'application **[3]**.



22.7.2 script [application_scope_01]

Le script **[application_scope_01]** montre une façon de gérer des données de portée 'application' :

```
1.  # on configure l'application
2.  import config
3.  config = config.configure()
4.
5.  # dépendances
6.  from flask import Flask, make_response
7.  from flask_api import status
8.
9.  # application Flask
10. app = Flask(__name__)
11.
12.
13. # Home URL
14. @app.route('/', methods=['GET'])
15. def index():
16.     # on vise à montrer que l'application reste en mémoire entre les requêtes des différents clients
17.     # chaque client a affaire à la même application
18.
19.     # app_infos représente des informations de niveau application et non de niveau session
20.     # c-à-d qu'elle concerne tous les utilisateurs et non un en particulier
21.     # cette information est ici stockée dans [config] (pas obligatoire)
22.
23.     # dictionnaire des résultats
24.     résultats = {"config": config}
25.
26.     # on envoie la réponse
```

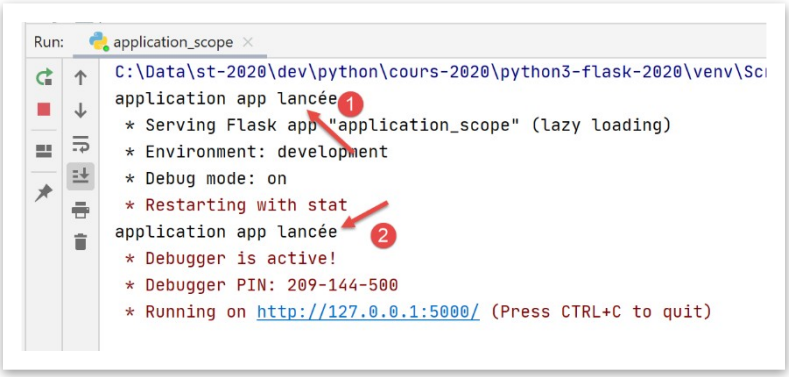
```
27.     response = make_response(résultats)
28.     response.headers['Content-Type'] = 'application/json; charset=utf-8'
29.     return response, status.HTTP_200_OK
30.
31.
32. # main
33. if __name__ == '__main__':
34.     # on vérifie si ce code est exécuté plusieurs fois
35.     print("application app lancée")
36.     # on lance l'application web
37.     app.config.update(ENV="development", DEBUG=True)
38.     app.run()
```

- lignes 1-3 : on récupère le dictionnaire de la configuration. On va montrer que le code situé en-dehors des fonctions de routage n'est exécuté qu'une fois. L'application Flask reste en mémoire. Toutes les informations initialisées en-dehors des routes sont globales à celles-ci et donc connues de celles-ci. Ainsi le dictionnaire **[config]** de la ligne 3 va-t-il être rendu par la route / (ligne 24). On va montrer que tous les clients web vont recevoir le même dictionnaire et que celui-ci est donc partagé par tous les clients. C'est donc une information de portée 'application' ;
- ligne 35 : on met un log pour voir si le code des lignes en-dehors de la fonction de routage (lignes 1-10, 32-38) est exécuté plusieurs fois ;

La configuration **[config]** est la suivante :

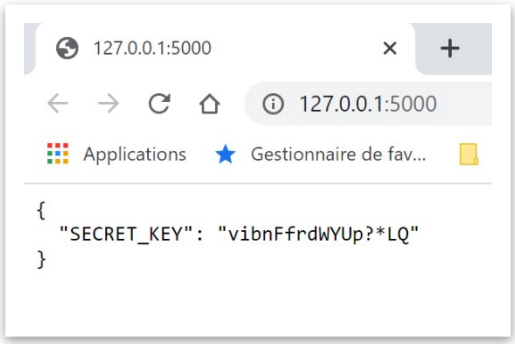
```
1.  def configure():
2.      # on rend la config
3.      config = {
4.          # configuration Flask
5.          "SECRET_KEY": "vibnFfrdWYUp?*LQ"
6.      }
7.
8.      return config
```

Nous lançons cette application. Les logs dans la console PyCharm sont les suivants :

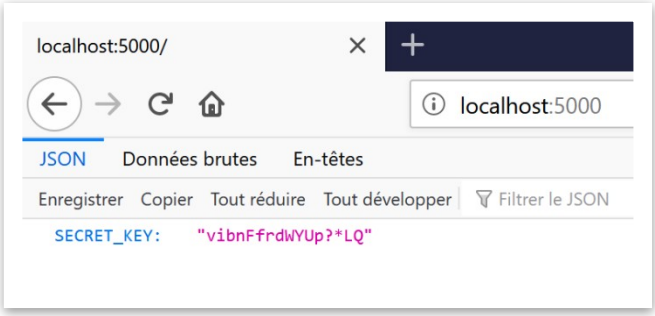


- en **[1]**, lancement initial de l'application ;
- en **[2]** parce qu'on a demandé le mode **[Debug]**, l'application est relancée en mode **[Debug]** ;

Maintenant avec un navigateur (Chrome ci-dessous), on demande l'URL [http://127.0.0.1:5000/] :



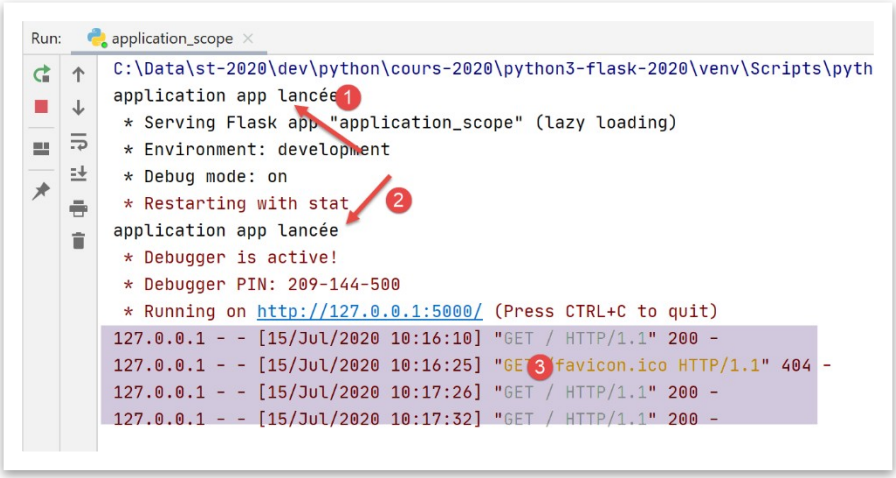
Maintenant avec un navigateur Firefox :



Maintenant avec le client Postman :

```
1.  GET / HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: 51e75099-8ecb-4f27-ae3b-9386e982ede4
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
9.
10. HTTP/1.0 200 OK
11. Content-Type: application/json; charset=utf-8
12. Content-Length: 39
13. Server: Werkzeug/1.0.1 Python/3.8.1
14. Date: Wed, 15 Jul 2020 10:34:26 GMT
15.
16. {
17. "SECRET_KEY": "vibnFfrdWYUp?*LQ"
18. }
```

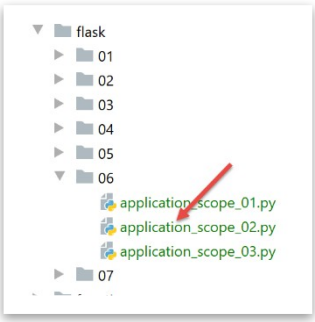
Maintenant, on revient dans la console **[Run]** de Pycharm :



- les deux logs **[1, 2]** sont toujours là mais il n'y en a pas d'autres alors que l'on voit les trois requêtes reçues par le serveur web ;

Pour être totalement sûrs que l'application n'est pas rechargée à chaque nouvelle requête, on peut mettre un compteur dans la configuration et l'incrémenter à chaque nouvelle requête. On verra alors que chaque client voit le compteur dans l'état où l'a laissé le précédent client. On rappelle cependant que les clients ne devraient pas modifier des données de portée application parce qu'elles sont partagées entre tous les clients et que dans un contexte où le serveur sert simultanément plusieurs clients sans garantie que la requête d'un client soit exécutée entièrement sans être interrompue, un client 1 qui a émis une requête 1 interrompue avant sa fin peut laisser les données partagées dans un état corrompu pour les clients suivants.

22.7.3 script [application_scope_02]



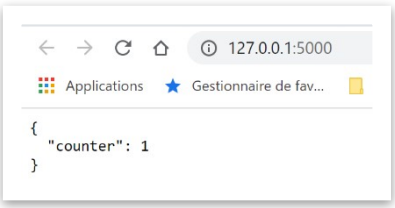
Le script **[application_scope_02]** va faire ce qu'il ne faut pas faire : permettre aux clients de modifier des informations partagées avec les autres utilisateurs. On va partager un compteur entre les utilisateurs qui vont l'incrémenter. On va voir que chaque utilisateur voit les modifications apportées par les autres utilisateurs au compteur.

Le script est le suivant :

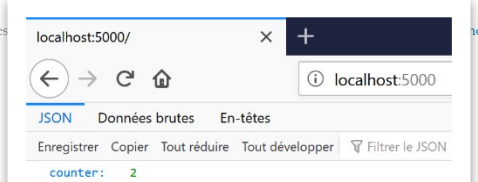
```
1.  # dépendances
2.
3.  from flask import Flask, make_response
4.  from flask_api import status
5.
6.  # application Flask
7.  app = Flask(__name__)
8.
9.  # données de portée application
10. config = {
11.     "counter": 0
12. }
13.
14.
15. # Home URL
16. @app.route('/', methods=['GET'])
17. def index():
18.     # on vise à montrer que le dictionnaire [config] est partagé entre tous les clients
19.     # de l'application web
20.
21.     # on incrémente le compteur
22.     config["counter"] += 1
23.     # on envoie la réponse
24.     response = make_response(config)
25.     response.headers['Content-Type'] = 'application/json; charset=utf-8'
26.     return response, status.HTTP_200_OK
27.
28.
29. # main
30. if __name__ == '__main__':
31.     app.config.update(ENV="development", DEBUG=True)
32.     app.run()
```

- lignes 10-12 : le dictionnaire **[config]** partagé par les utilisateurs. Il contient un compteur ;
- ligne 22 : à chaque fois qu'un utilisateur demandera l'URL /, le compteur de la configuration sera incrémenté ;
- lignes 23-26 : la chaîne jSON du dictionnaire est envoyée à chaque client ;

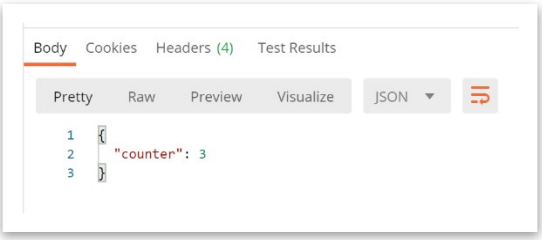
On lance ce script. Puis on demande l'URL [http://127.0.0.1:5000/] avec un premier navigateur :



On fait ensuite la même chose avec un second navigateur :



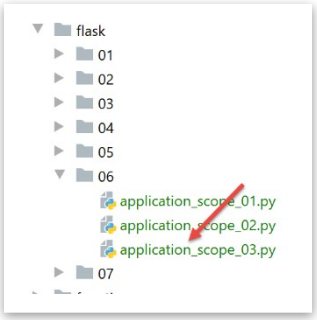
Puis une troisième fois avec Postman :



On voit que chaque client récupère le compteur dans l'état où le client précédent l'a laissé. Ils ont donc bien accès à la même information.

22.7.4 script `[application_scope_03]`

Le script **`[application_scope_03]`** montre pourquoi l'information partagée entre utilisateurs doit être en lecture seule.



Le script est le suivant :

```
1.  # dépendances
2.  import threading
3.  from time import sleep
4.
5.  from flask import Flask, make_response
6.  from flask_api import status
7.
8.  # application Flask
9.  app = Flask(__name__)
10.
11. # données de portée application
12. config = {
13.     "counter": 0
14. }
15.
16.
17. # Home URL
18. @app.route('/', methods=['GET'])
19. def index():
20.     # on vise à montrer que le dictionnaire [config] est partagé entre tous les clients
21.     # de l'application web et qu'il doit être en lecture seule
22.
23.     # nom du thread
24.     thread_name = threading.current_thread().name
25.     # on lit le compteur
26.     counter = config["counter"]
27.     print(f"compteur lu : {counter}, par le thread {thread_name}")
28.     # on s'arrête 5 secondes - du coup d'autres clients vont être servis
29.     sleep(5)
30.     # on incrémente le compteur de la configuration
31.     config["counter"] = counter + 1
32.     # log
33.     print(f"compteur écrit : {config['counter']}, par le thread {thread_name}")
34.     # on envoie la réponse
35.     response = make_response(config)
36.     response.headers['Content-Type'] = 'application/json; charset=utf-8'
37.     return response, status.HTTP_200_OK
38.
39.
40. # main
41. if __name__ == '__main__':
```

```
42.     app.config.update(ENV="development", DEBUG=True)
43.     app.run(threaded=True)
```

- ligne 43 : on a changé le mode d'exécution de l'application web. On a écrit **[threaded=True]** pour indiquer que l'application devait servir les utilisateurs simultanément. Cela se fait au moyen de threads d'exécution :
  - il peut y avoir plusieurs threads d'exécution simultanés, chacun servant un utilisateur ;
  - le processeur de la machine est partagé par ces threads ;
  - un thread peut être interrompu avant qu'il n'ait terminé son travail. Il sera repris ultérieurement ;
- ligne 19 : la fonction **[index]** peut être exécutée simultanément par plusieurs threads ;
- ligne 24 : on récupère le nom du thread qui exécute la fonction **[index]** ;
- ligne 26 : on lit la valeur du compteur. Pour les besoins de notre démonstration on décompose l'incrémentation du compteur de la façon suivante :
  - étape 1 : lecture du compteur (1 par exemple) par le thread 1 ;
  - étape 2 : pause du thread 1 pendant 5 secondes (ligne 29). Parce que le thread 1 a demandé une pause, le processeur est donné à un autre thread, le thread 2. Le but est que ce nouveau thread lise la même valeur du compteur (=1). Puis lui aussi fait une pause de 5 secondes et perd le processeur ;
  - étape 3 : incrémentation du compteur, ligne 31 à partir de la valeur lue à l'étape 1 (=1). Le thread 1 est le 1er à le faire : il passe le compteur à 2 puis termine l'exécution de la fonction **[index]**. Puis c'est au tour du thread 2 de se réveiller et de passer lui aussi le compteur à 2 à partir de la valeur lue à l'étape 1 (=1). Au final, après le passage des deux threads, le compteur est à 2 alors qu'il devrait être à 3 ;
- ligne 33 : on affiche la valeur du compteur pour vérification ;

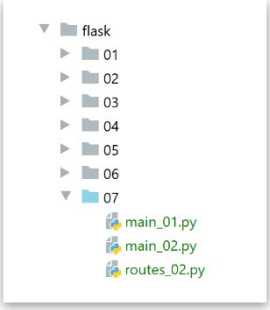
Nous lançons le script puis demandons l'url **[http://loaclhost :5000/]** avec deux navigateurs puis avec Postman. Les logs dans la console PyCharm sont alors les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/flask/06/application_scope_03.py
2.   * Serving Flask app "application_scope_03" (lazy loading)
3.   * Environment: development
4.   * Debug mode: on
5.   * Restarting with stat
6.   * Debugger is active!
7.   * Debugger PIN: 334-263-283
8.   * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
9.  compteur lu : 0, par le thread Thread-2
10. compteur lu : 0, par le thread Thread-4
11. compteur écrit : 1, par le thread Thread-2
12. 127.0.0.1 - - [16/Jul/2020 08:55:37] "GET / HTTP/1.1" 200 -
13. compteur écrit : 1, par le thread Thread-4
14. 127.0.0.1 - - [16/Jul/2020 08:55:40] "GET / HTTP/1.1" 200 -
15. compteur lu : 1, par le thread Thread-5
16. compteur écrit : 2, par le thread Thread-5
17. 127.0.0.1 - - [16/Jul/2020 08:55:46] "GET / HTTP/1.1" 200 -
```

- lignes 9-10 : les deux premiers threads 2 et 4 lisent la même valeur 0 du compteur ;
- ligne 11 : le thread 2 passe le compteur à 1 ;
- ligne 13 : le thread 4 passe le compteur à 1. A partir de maintenant la valeur du compteur est incorrecte ;
- lignes 15-16 : le thread 5 n'est pas interrompu et gère correctement la valeur du compteur ;

On retiendra de cet exemple que le code d'une application web ne doit pas modifier la valeur d'informations partagées par les utilisateurs.

22.8 scripts [flask/07] : gestion des routes



Nous nous intéressons ici à la gestion des routes d'une application, ç-à-d les URL servies par l'application web.

22.8.1 script [main_01] : routes paramétrées

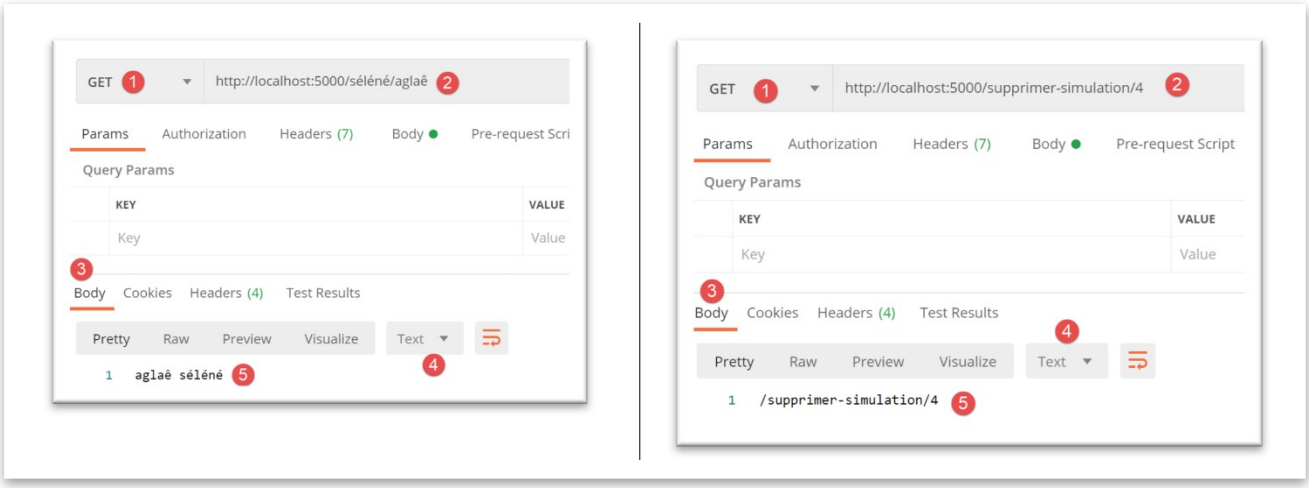
Le script **[main_01]** introduit la possibilité de paramétrer les routes :

```
1.  from flask import Flask, make_response
2.  from flask_api import status
3.
4.  # application Flask
5.  app = Flask(__name__)
6.
7.
8.  # envoi de la réponse
9.  def send_plain_response(réponse: str):
10.     # on envoie la réponse
11.     response = make_response(réponse)
12.     response.headers['Content-Type'] = 'text/plain; charset=utf-8'
13.     return response, status.HTTP_200_OK
14.
15.
16. # /nom/prenom
17. @app.route('/<string:nom>/<string:prenom>', methods=['GET'])
18. def index(nom, prenom):
19.     # réponse
20.     return send_plain_response(f"{prenom} {nom}")
21.
22.
23. # init-session
24. @app.route('/init-session/<string:type>', methods=['GET'])
25. def init_session(type: str):
26.     # réponse
27.     return send_plain_response(f"/init-session/{type}")
28.
29.
30. # authentifier-utilisateur
31. @app.route('/authentifier-utilisateur', methods=['POST'])
32. def authentifier_utilisateur():
33.     # réponse
34.     return send_plain_response("/authentifier-utilisateur")
35.
36.
37. # calculer-impot
38. @app.route('/calculer-impot', methods=['POST'])
39. def calculer_impot():
40.     # réponse
41.     return send_plain_response("/calculer-impot")
42.
43.
44. # lister-simulations
45. @app.route('/lister-simulations', methods=['GET'])
46. def lister_simulations():
47.     # réponse
48.     return send_plain_response("/lister-simulations")
49.
50.
51. # supprimer-simulation
52. @app.route('/supprimer-simulation/<int:numero>', methods=['GET'])
53. def supprimer_simulation(numero: int):
54.     # réponse
55.     return send_plain_response(f"/supprimer-simulation/{numero}")
56.
57.
58. # fin-session
59. @app.route('/fin-session', methods=['GET'])
60. def fin_session():
61.     # réponse
62.     return send_plain_response(f"/fin-session")
63.
64.
65. # main
66. if __name__ == '__main__':
67.     app.config.update(ENV="development", DEBUG=True)
68.     app.run()
```

- ligne 17 : on précise le type des paramètres de l'URL. Cela permet à Flask de faire des vérifications. Si le paramètre n'est pas du type attendu, la requête du client sera refusée (erreur 400 Bad Request). Donc Flask fait une partie du travail que nous aurions du faire ;

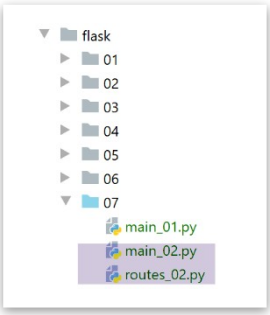
- ligne 18 : pour les paramètres, on doit reprendre les noms exacts des paramètres de la ligne 17 mais pas forcément leur ordre ;
- ligne 20 : on utilise la fonction **[send_plain_response]** pour envoyer la réponse au client web ;
- ligne 9 : la fonction **[send_plain_response]** reçoit la chaîne de caractères à envoyer au client ;
- ligne 11 : le corps de la réponse HTTP est construit ;
- ligne 12 : on dit au client qu'on lui envoie du texte brut ;
- ligne 13 : on envoie la réponse HTTP ;
- lignes 23-62 : d'autres routes paramétrées qui seront utilisées ultérieurement dans un exercice d'application ;

On lance le script et on l'interroge avec le client Postman :



22.8.2 script [main_02] : externalisation des routes

Dans le script **[main_01]** précédent, le code peut devenir important s'il y a de nombreuses routes. Le script **[main_02]** montre comment externaliser les routes.



Le script **[routes_02]** rassemble les fonctions associées aux routes du script précédent :

```
1.  from flask import make_response
2.  from flask_api import status
3.
4.
5.  def send_response(réponse: str):
6.      # on envoie la réponse
7.      response = make_response(réponse)
8.      response.headers['Content-Type'] = 'text/plain; charset=utf-8'
9.      return response, status.HTTP_200_OK
10.
11.
12. # Home URL
13. def index(nom, prenom):
14.     # réponse
15.     return send_response(f"{prenom} {nom}")
16.
17.
18. # init-session
19. def init_session(type: str):
20.     # réponse
```

```
    return send_response(f"/init-session/{type}")


# authentifier-utilisateur
def authentifier_utilisateur():
    # réponse
    return send_response("/authentifier-utilisateur")


# calculer-impot
def calculer_impot():
    # réponse
    return send_response("/calculer-impot")


# lister-simulations
def lister_simulations():
    # réponse
    return send_response("/lister-simulations")


# supprimer-simulation
def supprimer_simulation(numero: int):
    # réponse
    return send_response(f"/supprimer-simulation/{numero}")


# fin-session
def fin_session():
    # réponse
    return send_response(f"/fin-session")
```

On notera que le script **[routes_02]** n'est pas un script de routes. C'est une liste de fonctions. C'est le script principal **[main_02]** qui fait le lien entre routes et fonctions :

```
1.  from flask import Flask
2.
3.  # on déporte les fonctions des routes dans leur propre script
4.  import routes_02
5.
6.  # application Flask
7.  app = Flask(__name__)
8.
9.  # associations routes / fonctions
10. app.add_url_rule('/<string:nom>/<string:prenom>', methods=['GET'], view_func=routes_02.index)
11. app.add_url_rule('/init-session/<string:type>', methods=['GET'], view_func=routes_02.init_session)
12. app.add_url_rule('/authentifier-utilisateur', methods=['POST'], view_func=routes_02.authentifier_utilisateur)
13. app.add_url_rule('/calculer-impot', methods=['POST'], view_func=routes_02.calculer_impot)
14. app.add_url_rule('/lister-simulations', methods=['GET'], view_func=routes_02.lister_simulations)
15. app.add_url_rule('/supprimer-simulation/<int:numero>', methods=['GET'], view_func=routes_02.supprimer_simulation)
16. app.add_url_rule('/fin-session', methods=['GET'], view_func=routes_02.fin_session)
17.
18. # main
19. if __name__ == '__main__':
20.     app.config.update(ENV="development", DEBUG=True)
21.     app.run()
```

- ligne 4 : on importe le script des fonctions associées aux routes ;
- lignes 9-16 : association routes / fonctions ;

Avec cette méthode, chaque fonction associée à une route peut faire l'objet d'un script séparé si nécessaire.

Les résultats sont les mêmes que ceux obtenus avec le script **[main_01]** précédent.

23 Exercice d'application : version 6

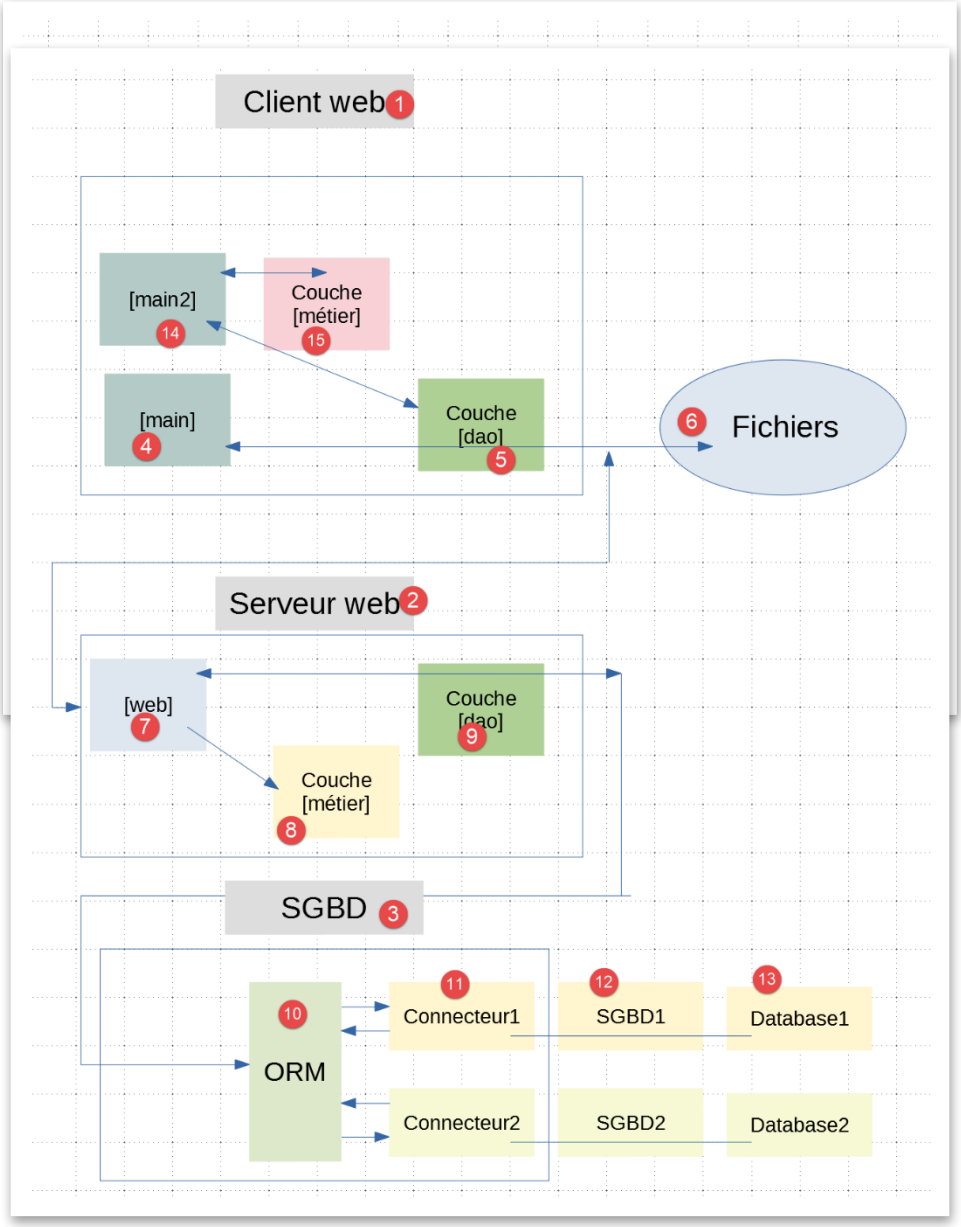
23.1 Introduction

Nous revenons maintenant à notre application de calcul de l'impôt. Nous allons construire autour d'elle différentes applications web.

Dans la version 5 de notre exercice d'application, les données de l'administration fiscale étaient rangées dans une base de données. Cette version 5 comportait deux applications distinctes mais ayant des couches en commun :

- une application qui calculait l'impôt en mode |batch| pour des contribuables enregistrés dans un fichier texte ;
- une application qui calculait l'impôt en mode |interactif| pour des contribuables dont on saisissait les informations au clavier ;

La version 5 de l'application de calcul de l'impôt par lots (mode batch) avait l'architecture suivante :



Ultimement, la version web de cette application aura l'architecture suivante :

- le client web **[1]** s'adresse au serveur web **[2]** qui lui communique avec le SGBD **[3]** ;
- le serveur web **[2]** conserve les couches **[métier] [8]** et **[dao] [9]** de l'application initiale ;
- l'application initiale conserve son script principal **[4]** et sa couche **[métier] [15]**. Les couches **[métier] [8]** et **[15]** sont identiques;
- la communication client / serveur nécessite deux couches supplémentaires :
  - la couche **[web] [7]** qui implémente l'application web ;

- la couche **[dao]** **[5]** cliente de l'application web **[7]** ;

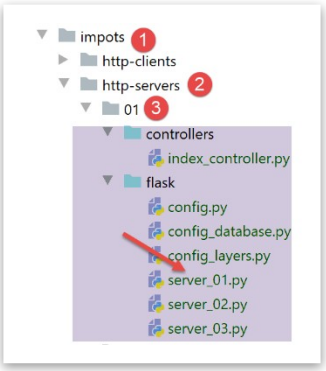
Dans la version finale, le calcul de l'impôt par lots pourra se dérouler de deux façons :

- le calcul métier de l'impôt se fait par la couche **[métier]** du serveur. Le script **[main]** utilisera cette méthode ;
- le calcul métier de l'impôt se fait par la couche **[métier]** du client. Le script **[main2]** utilisera cette méthode ;

A partir de maintenant, nous allons développer plusieurs applications client / serveur du type ci-dessus, chacune illustrant une ou de nouvelles technologies de développement web.

23.2 Le serveur web de calcul de l'impôt

23.2.1 Version 1

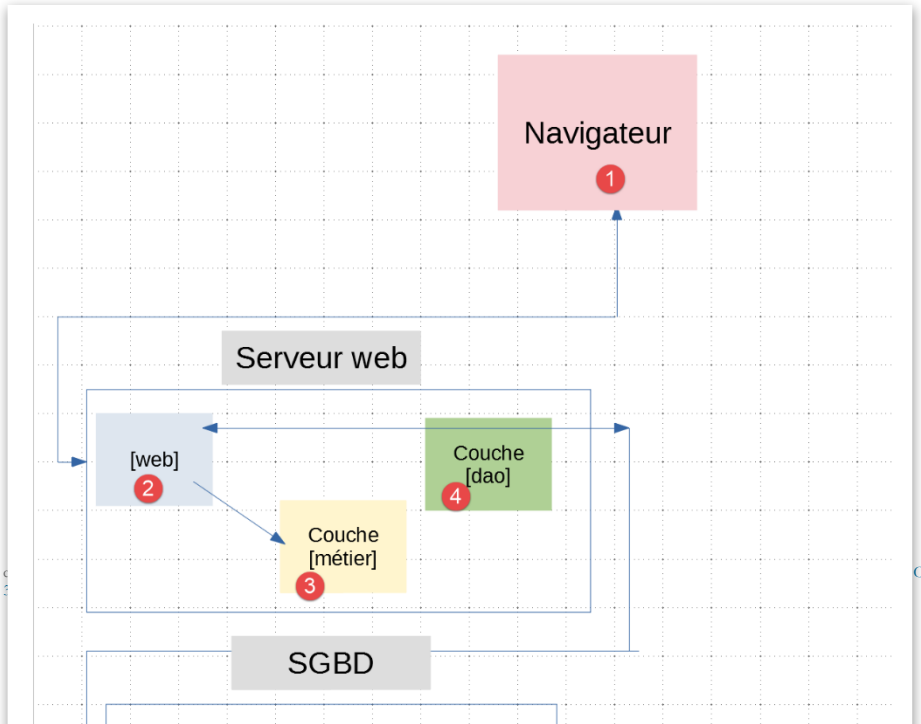


Le script **[server_01]** est l'application web suivante :



- en **[1]**, on utilise une URL paramétrée dans laquelle on passe trois valeurs :
  - **[marié]** (oui / non) pour indiquer si le contribuable est marié ;
  - **[enfants]** : le nombre d'enfants du contribuable ;
  - **[salaire]** : salaire annuel du contribuable ;
- en **[2]**, le serveur web renvoie une chaîne jSON qui donne le montant de l'impôt à payer avec avec ses différentes composantes ;

L'architecture de l'application est la suivante :



- le navigateur **[1]** interroge le serveur **[2]**. Le script **[server_01]** implémente la couche **[web] [2]** du serveur ;
- les couches **[3-8]** sont celles déjà utilisées dans la |version 5| de l'application du calcul de l'impôt. Nous les reprenons telles-quelles ;
  - la couche **[métier] [3]** est définie |ici| ;
  - la couche **[dao] [4]** est définie |ici| ;

L'application web **[server_01]** est configurée à l'aide de trois scripts :

- **[config]** qui configure la totalité de l'application ;
- **[config_database]** qui configure l'accès à la base de données. On travaillera avec les SGBD MySQL et PostgreSQL ;
- **[config_layers]** qui configure les couches de l'application ;

Le script **[config]** est le suivant :

```
1.  def configure(config: dict) -> dict:
2.      import os
3.
4.      # étape 1 ------
5.      # dossier de ce fichier
6.      script_dir = os.path.dirname(os.path.abspath(__file__))
7.      # chemin racine
8.      root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
9.      # dépendances absolues
10.     absolute_dependencies = [
11.         # dossiers du projet
12.         # BaseEntity, MyException
13.         f"{root_dir}/classes/02/entities",
14.         # InterfaceImpôtsDao, InterfaceImpôtsMétier, InterfaceImpôtsUi
15.         f"{root_dir}/impots/v04/interfaces",
16.         # AbstractImpôtsdao, ImpôtsConsole, ImpôtsMétier
17.         f"{root_dir}/impots/v04/services",
18.         # ImpotsDaoWithAdminDataInDatabase
19.         f"{root_dir}/impots/v05/services",
20.         # AdminData, ImpôtsError, TaxPayer
21.         f"{root_dir}/impots/v04/entities",
22.         # Constantes, tranches
23.         f"{root_dir}/impots/v05/entities",
24.         # IndexController
25.         f"{script_dir}/../controllers",
26.         # scripts [config_database, config_layers]
27.         script_dir,
28.     ]
29.     # on fixe le syspath
30.     from myutils import set_syspath
31.     set_syspath(absolute_dependencies)
32.
33.     # étape 2 ------
34.     # configuration de l'application
35.     # liste des utilisateurs autorisés à utiliser l'application
36.     config['users'] = [
37.         {
38.             "login": "admin",
39.             "password": "admin"
40.         }
41.     ]
42.
43.     # étape 3 ------
44.     # configuration base de données
45.     import config_database
46.     config["database"] = config_database.configure(config)
47.
48.     # étape 4 ------
49.     # instanciation des couches de l'application
50.     import config_layers
51.     config['layers'] = config_layers.configure(config)
52.
53.     # on rend la configuration
54.     return config
```

- la fonction **[configure]** reçoit un dictionnaire **[config]** en paramètre (ligne 1) et le rend comme résultat (ligne 54) après avoir enrichi son contenu. On aurait pu dire depuis longtemps qu'il n'était pas nécessaire de rendre le résultat **[config]**. En effet **[config]** est une référence de dictionnaire que le code appelant partage avec le code appelé. Le code appelant possède donc déjà cette référence (ligne 1) et il est inutile de la lui redonner (ligne 54). Ainsi écrire :

```
config=[module].configure(config) (1)
```

est redondant. Il suffit d'écrire :

```
[module].configure(config) (2)
```

Néanmoins, j'ai gardé le type (1) d'écriture parce que j'ai pensé qu'il montrait peut-être mieux, que le code appelé modifiait le dictionnaire **[config]**.

- ligne 1 : le dictionnaire **[config]** reçu par la fonction **[configure]** a une clé 'sgbd' qui prend sa valeur dans la liste **['mysql', 'pgres']**. **[mysql]** signifie que la base de données utilisée est gérée par MySQL alors que 'pgres' signifie que la base de données utilisée est gérée par PostgreSQL ;
- lignes 4-27 : on liste tous les dossiers contenant des éléments nécessaires à l'application web. Ils feront partie du Python Path de l'application (lignes 30-31) ;
- lignes 33-40 : on ne va autoriser que certains utilisateurs à accéder à l'application. Ici on a une liste avec un unique utilisateur ;
- lignes 43-46 : c'est le script **[config_database]** qui construit la configuration de la base de données utilisée ;
- ligne 46 : la configuration construite par le script **[config_database]** est un dictionnaire qu'on range dans la configuration générale associé à la clé 'database' ;
- lignes 48-51 : le script **[config_layers]** instancie les couches de l'application web. Il rend un dictionnaire qu'on range dans la configuration générale associé à la clé 'layers' ;

Le script **[config_database]** est celui déjà utilisé dans la |version 5|. On le redonne pour mémoire :

```
1.  def configure(config: dict) -> dict:
2.      # configuration sqlalchemy
3.      from sqlalchemy import create_engine, Table, Column, Integer, MetaData, Float
4.      from sqlalchemy.orm import mapper, sessionmaker
5.
6.      # chaînes de connexion aux bases de données exploitées
7.      connection_strings = {
8.          'mysql': "mysql+mysqlconnector://admimpots:mdpimpots@localhost/dbimpots-2019",
9.          'pgres': "postgresql+psycopg2://admimpots:mdpimpots@localhost/dbimpots-2019"
10.     }
11.     # chaîne de connexion à la base de données exploitée
12.     engine = create_engine(connection_strings[config['sgbd']])
13.
14.     # metadata
15.     metadata = MetaData()
16.
17.     # la table des constantes
18.     constantes_table = Table("tbconstantes", metadata,
19.                              Column('id', Integer, primary_key=True),
20.                              Column('plafond_qf_demi_part', Float, nullable=False),
21.                              Column('plafond_revenus_celibataire_pour_reduction', Float, nullable=False),
22.                              Column('plafond_revenus_couple_pour_reduction', Float, nullable=False),
23.                              Column('valeur_reduc_demi_part', Float, nullable=False),
24.                              Column('plafond_decote_celibataire', Float, nullable=False),
25.                              Column('plafond_decote_couple', Float, nullable=False),
26.                              Column('plafond_impot_celibataire_pour_decote', Float, nullable=False),
27.                              Column('plafond_impot_couple_pour_decote', Float, nullable=False),
28.                              Column('abattement_dixpourcent_max', Float, nullable=False),
29.                              Column('abattement_dixpourcent_min', Float, nullable=False)
30.                              )
31.
32.     # la table des tranches de l'impôt
33.     tranches_table = Table("tbtranches", metadata,
34.                            Column('id', Integer, primary_key=True),
35.                            Column('limite', Float, nullable=False),
36.                            Column('coeffr', Float, nullable=False),
37.                            Column('coeffn', Float, nullable=False)
38.                            )
39.     # les mappings
40.     from Tranche import Tranche
41.     mapper(Tranche, tranches_table)
42.
43.     from Constantes import Constantes
44.     mapper(Constantes, constantes_table)
45.
46.     # la session factory
47.     session_factory = sessionmaker()
48.     session_factory.configure(bind=engine)
49.
```

```
50.     # une session
51.     session = session_factory()
52.
53.     # on enregistre certaines informations et on les rend dans un dictionnaire
54.     return {"engine": engine, "metadata": metadata, "tranches_table": tranches_table,
55.             "constantes_table": constantes_table, "session": session}
```

Le script **[config_layers]** configure les couches du serveur web. On reprend un |script| déjà rencontré :

```
1.  def configure(config: dict) -> dict:
2.      # instanciation des couches de l'application
3.
4.      # dao
5.      from ImpotsDaoWithAdminDataInDatabase import ImpotsDaoWithAdminDataInDatabase
6.      dao = ImpotsDaoWithAdminDataInDatabase(config)
7.
8.      # métier
9.      from ImpôtsMétier import ImpôtsMétier
10.     métier = ImpôtsMétier()
11.
12.     # on met les instances de couches dans un dictionnaire qu'on rend au code appelant
13.     return {
14.         "dao": dao,
15.         "métier": métier
16.     }
```

- ligne 6 : la couche **[dao]** est implémentée avec une base de données ;
- **[ImpotsDaoWithAdminDataInDatabase]** a été définie |ici| ;
- **[ImpôtsMétier]** a été définie |ici| ;

Le script principal **[server_01]** est le suivant :

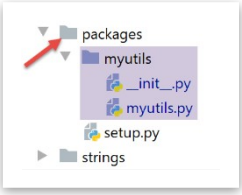
```
1.  # on attend un paramètre mysql ou pgres
2.  import sys
3.  syntaxe = f"{sys.argv[0]} mysql / pgres"
4.  erreur = len(sys.argv) != 2
5.  if not erreur:
6.      sgbd = sys.argv[1].lower()
7.      erreur = sgbd != "mysql" and sgbd != "pgres"
8.  if erreur:
9.      print(f"syntaxe : {syntaxe}")
10.     sys.exit()
11.
12. # on configure l'application
13. import config
14. config = config.configure({'sgbd': sgbd})
15.
16. # dépendances
17. from ImpôtsError import ImpôtsError
18. from TaxPayer import TaxPayer
19. import re
20. from flask import request
21. from myutils import json_response
22. from flask import Flask
23. from flask_api import status
24.
25. # récupération des données de l'administration fiscale
26. try:
27.     # admindata sera une donnée de portée application en lecture seule
28.     admindata = config["layers"]["dao"].get_admindata()
29. except ImpôtsError as erreur:
30.     print(f"L'erreur suivante s'est produite : {erreur}")
31.     sys.exit(1)
32.
33. # application Flask
34. app = Flask(__name__)
35.
36.
37. # Home URL : /?marié=xx&enfants=yy&salaire=zz
38. @app.route('/', methods=['GET'])
39. def index():
40.     # au départ pas d'erreurs
41.     erreurs = []
42.     # la requête doit avoir trois paramètres dans l'URL
43.     if len(request.args) != 3:
```

```
44.         erreurs.append("Méthode GET requise avec les seuls paramètres [marié, enfants, salaire]")
45.
46.     # on récupère le statut marital dans l'URL
47.     marié = request.args.get('marié')
48.     if marié is None:
49.         erreurs.append("paramètre [marié] manquant")
50.     else:
51.         marié = marié.strip().lower()
52.         erreur = marié != "oui" and marié != "non"
53.         if erreur:
54.             erreurs.append(f"paramétre marié [{marié}] invalide")
55.
56.     # on récupère le nombre d'enfants dans l'URL
57.     enfants = request.args.get('enfants')
58.     if enfants is None:
59.         erreurs.append("paramètre [enfants] manquant")
60.     else:
61.         enfants = enfants.strip()
62.         match = re.match(r"^\d+", enfants)
63.         if not match:
64.             erreurs.append(f"paramétre enfants [{enfants}] invalide")
65.         else:
66.             enfants = int(enfants)
67.
68.     # on récupère le salaire dans l'URL
69.     salaire = request.args.get('salaire')
70.     if salaire is None:
71.         erreurs.append("paramètre [salaire] manquant")
72.     else:
73.         salaire = salaire.strip()
74.         match = re.match(r"^\d+", salaire)
75.         if not match:
76.             erreurs.append(f"paramétre salaire [{salaire}] invalide")
77.         else:
78.             salaire = int(salaire)
79.
80.     # des paramètres invalides dans l'URL ?
81.     for key in request.args.keys():
82.         if key not in ['marié', 'enfants', 'salaire']:
83.             erreurs.append(f"paramètre [{key}] invalide")
84.
85.     # des erreurs ?
86.     if erreurs:
87.         # on envoie une réponse d'erreur au client
88.         résultats = {"réponse": {"erreurs": erreurs}}
89.         return json_response(résultats, status.HTTP_400_BAD_REQUEST)
90.
91.     # pas d'erreurs, on peut travailler
92.     # calcul de l'impôt
93.     taxpayer = TaxPayer().fromdict({'marié': marié, 'enfants': enfants, 'salaire': salaire})
94.     config["layers"]["métier"].calculate_tax(taxpayer, admindata)
95.     # on envoie la réponse au client
96.     return json_response({"réponse": {"result": taxpayer.asdict()}}, status.HTTP_200_OK)
97.
98.
99. # main uniquement
100.if __name__ == '__main__':
101.    # on lance le serveur Flask
102.    app.config.update(ENV="development", DEBUG=True)
103.    app.run()
```

- lignes 1-10 : on récupère le paramètre qui indique quel SGBD utiliser ;
- lignes 12-14 : munie de cette information on peut configurer l'application. Le Python Path est notamment construit ;
- lignes 16-23 : muni du nouveau Python Path, on importe les éléments dont on a besoin ;
- lignes 25-31 : on récupère les données de l'administration fiscale qui permettent de faire le calcul de l'impôt ;
- lignes 33-34 : instanciation de l'application Flask ;
- ligne 38 : l'application Flask ne sert que l'URL **[/]**. Elle attend une URL paramétrée de la façon suivante **[/ ? marié=xx&enfants=yy&salaire=zz]** avec :
  - xx : oui / non ;
  - yy : nombre d'enfants ;
  - zz : salaire annuel ;
- lignes 40-89 : on vérifie la validité des paramètres de l'URL ;
- ligne 41 : on va cumuler les messages d'erreur dans la liste **[erreurs]** ;
- ligne 43 : on se rappelle peut-être que les paramètres de l'URL paramétrée sont retrouvés dans **[request.args]** (voir |ici|) :

    - l'objet **[request]** est l'objet Flask importé ligne 20 ;
    - l'objet **[request.args]** se comporte comme un dictionnaire ;
- lignes 43-44 : on vérifie qu'on a trois paramètres exactement (pas moins, pas plus) ;
- lignes 46-49 : on vérifie que le paramètre **[marié]** est présent dans l'URL ;
- lignes 50-54 : s'il est présent on vérifie que sa valeur minuscule débarrassée des 'blancs' de début / fin est oui ou non ;
- lignes 56-59 : on vérifie que le paramètre **[enfants]** est dans l'URL ;
- lignes 60-66 : s'il est présent on vérifie que sa valeur est un entier positif ;
- ligne 66 : il ne faut pas oublier que les paramètres de l'URL et leurs valeurs sont des chaînes de caractères. La valeur du paramètre **[enfants]** est transformée en 'int' ;
- lignes 68-78 : pour le paramètre **[salaire]**, on fait les mêmes tests que pour le paramètre **[enfants]** ;
- lignes 81-83 : on vérifie qu'il n'y a pas de paramètres autres que **['marié, 'enfants', 'salaire']** dans l'URL ;
- lignes 85-89 : si après toutes ces vérifications, la liste **[erreurs]** n'est pas vide alors on envoie cette liste d'erreurs au client sous la forme d'une chaîne jSON et le code de statut **[400 Bad Request]** ;

Comme par la suite, nous aurons souvent l'occasion d'envoyer une chaîne jSON en réponse au client, les quelques lignes nécessaires à cet envoi ont été factorisées dans le module **[myutils.py]** que nous avons déjà utilisé :



Le script **[myutils.py]** devient le suivant :

```
1.  # imports
2.  import json
3.  import os
4.  import sys
5.
6.  from flask import make_response
7.
8.
9.  def set_syspath(absolute_dependencies: list):
10.     # absolute_dependencies : une liste de noms absolus de dossiers
11.
12.     ….
13.
14.
15. # génération d'une réponse HTTP jSON
16. def json_response(réponse: dict, status_code: int) -> tuple:
17.     # corps de la réponse HTTP
18.     response = make_response(json.dumps(réponse, ensure_ascii=False))
19.     # corps de la réponse HTTP est du jSON
20.     response.headers['Content-Type'] = 'application/json; charset=utf-8'
21.     # on envoie la réponse HTTP
22.     return response, status_code
```

- ligne 16 : la fonction **[json_response]** attend deux paramètres :
    - **[réponse]** : le dictionnaire dont il faut envoyer la chaîne jSON au client web ;
    - **[status_code]** : le code de statut HTTP de la réponse ;
- ligne 18 : on fixe le corps jSON de la réponse ;
- ligne 20 : on ajoute l'entête HTTP qui dit au client web qu'il va recevoir du jSON ;
- ligne 22 : on envoie la réponse HTTP au code appelant. A charge pour lui de l'envoyer au client web ;

Le fichier **[__init__.py]** évolue de la façon suivante :

```
from .myutils import set_syspath, json_response
```

La nouvelle version de **[myutils]** est installée parmi les modules de portée machine avec la commande **[pip install .]** dans un terminal Pycharm :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\packages>pip install .
2.  Processing c:\data\st-2020\dev\python\cours-2020\python3-flask-2020\packages
3.  Using legacy setup.py install for myutils, since package 'wheel' is not installed.
4.  Installing collected packages: myutils
```

```
5.    Attempting uninstall: myutils
6.      Found existing installation: myutils 0.1
7.      Uninstalling myutils-0.1:
8.        Successfully uninstalled myutils-0.1
9.      Running setup.py install for myutils ... done
10. Successfully installed myutils-0.1
```

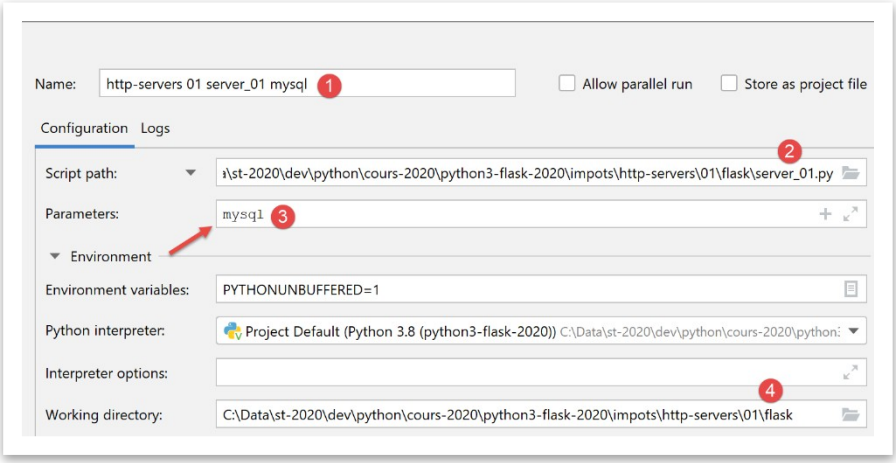
- ligne 1 : il faut être dans le dossier **[packages]** pour taper cette instruction ;

Le code du script **[server_01]** se poursuit de la façon suivante :

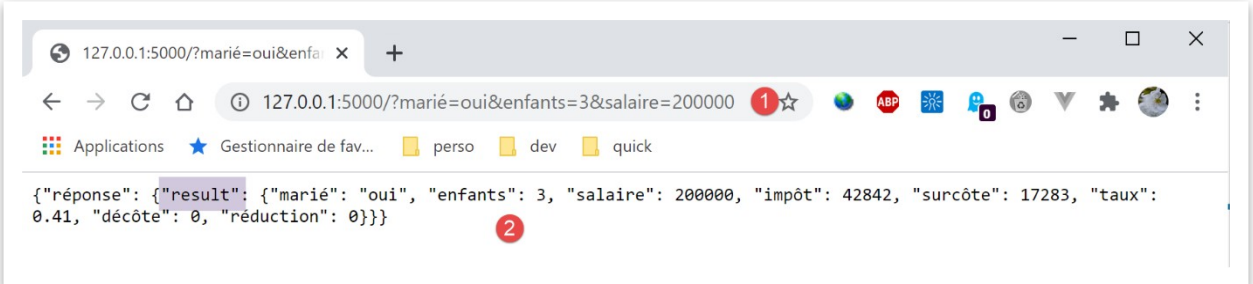
```
1.  …
2.      # des erreurs ?
3.      if erreurs:
4.          # on envoie une réponse d'erreur au client
5.          résultats = {"réponse": {"erreurs": erreurs}}
6.          return json_response(résultats, status.HTTP_400_BAD_REQUEST)
7.
8.      # pas d'erreurs, on peut travailler
9.      # calcul de l'impôt
10.     taxpayer = TaxPayer().fromdict({'id': 0, 'marié': marié, 'enfants': enfants, 'salaire': salaire})
11.     config["layers"]["métier"].calculate_tax(taxpayer, admindata)
12.     # on envoie la réponse au client
13.     return json_response({"réponse": {"result": taxpayer.asdict()}}, status.HTTP_200_OK)
14.
```

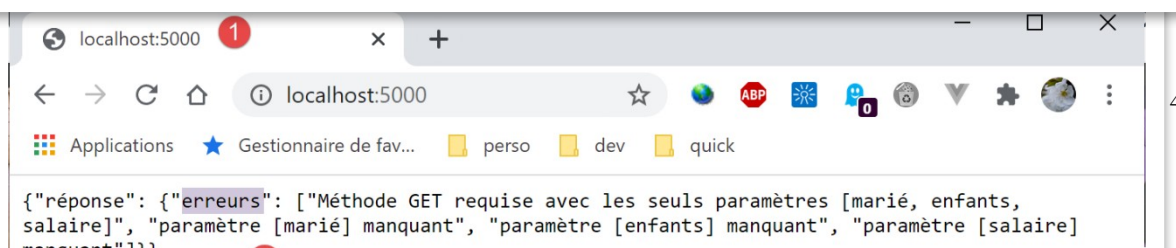
- ligne 10 : lorsqu'on est là, les paramètres attendus dans l'URL sont là et sont corrects ;
- ligne 10 : on crée l'objet **[TaxPayer]** qui modélise le contribuable ;
- ligne 11 : on demande à la couche **[métier]** de calculer l'impôt. On rappelle que les éléments calculés par la couche **[métier]** sont insérés dans l'objet **[taxpayer]** passé en paramètre ;
- ligne 13 : la réponse est envoyée au client web sous la forme d'une chaîne jSON. Celle-ci est la chaîne jSON d'un dictionnaire. Associé à la clé **[result]**, on y met le dictionnaire de l'objet **[taxpayer]**. On ne pouvait pas mettre l'objet **[taxpayer]** lui-même car celui-ci n'est pas sérialisable en jSON ;

On crée deux configurations d'exécution, une pour MySQL, l'autre pour PostgreSQL :



Voici quelques exemples d'exécution (vous avez lancé l'application **[server_01]** et le SGBD utilisé puis vous demandez l'URL http://localhost:5000/ avec un navigateur) :



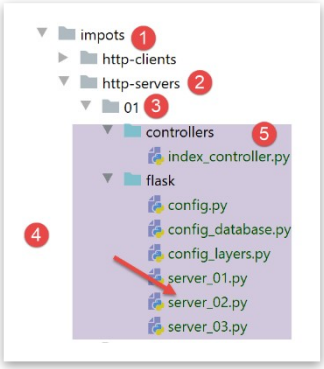


Voici un exemple d'exécution dans la console de Postman :

```
1.  GET /?mari%C3%A9=xx&enfants=yy&salaire=zz HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: e4c5df8c-4bd6-4250-b789-b7b164db4eff
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
9.
10. HTTP/1.0 400 BAD REQUEST
11. Content-Type: application/json; charset=utf-8
12. Content-Length: 134
13. Server: Werkzeug/1.0.1 Python/3.8.1
14. Date: Fri, 17 Jul 2020 06:15:44 GMT
15.
16. {"réponse": {"erreurs": ["paramètre marié [xx] invalide", "paramètre enfants [yy] invalide", "paramètre
    salaire [zz] invalide"]}}
```

- ligne 1 : une URL incorrecte est demandée ;
- ligne 10 : le serveur répond avec le statut 400 BAD REQUEST ;

23.2.2 Version 2



La version 2 du serveur isole le traitement de l'URL dans le module **[index_controller] [5]** :

```
1.  # import des dépendances
2.  import re
3.
4.  from flask_api import status
5.  from werkzeug.local import LocalProxy
6.
7.
8.  # URL paramétrée : /?marié=xx&enfants=yy&salaire=zz
9.  def execute(request: LocalProxy, config: dict) -> tuple:
10.     # dépendances
11.     from TaxPayer import TaxPayer
12.
13.     # au départ pas d'erreurs
14.     erreurs = []
15.     # la requête doit avoir trois paramètres
16.     if len(request.args) != 3:
```

```
17.         erreurs.append("Méthode GET requise avec les seuls paramètres [marié, enfants, salaire]")
18.
19.     # on récupère le statut marital de l'URL
20.     marié = request.args.get('marié')
21.     if marié is None:
22.         erreurs.append("paramètre [marié] manquant")
23.     else:
24.         marié = marié.strip().lower()
25.         erreur = marié != "oui" and marié != "non"
26.         if erreur:
27.             erreurs.append(f"paramétre marié [{marié}] invalide")
28.
29.     # on récupère le nombre d'enfants de l'URL
30.     enfants = request.args.get('enfants')
31.     if enfants is None:
32.         erreurs.append("paramètre [enfants] manquant")
33.     else:
34.         enfants = enfants.strip()
35.         match = re.match(r"^\d+", enfants)
36.         if not match:
37.             erreurs.append(f"paramétre enfants {enfants} invalide")
38.         else:
39.             enfants = int(enfants)
40.
41.     # on récupère le salaire de l'URL
42.     salaire = request.args.get('salaire')
43.     if salaire is None:
44.         erreurs.append("paramètre [salaire] manquant")
45.     else:
46.         salaire = salaire.strip()
47.         match = re.match(r"^\d+", salaire)
48.         if not match:
49.             erreurs.append(f"paramétre salaire {salaire} invalide")
50.         else:
51.             salaire = int(salaire)
52.
53.     # autres paramètres dans l'URL ?
54.     for key in request.args.keys():
55.         if not key in ['marié', 'enfants', 'salaire']:
56.             erreurs.append(f"paramètre [{key}] invalide")
57.
58.     # des erreurs ?
59.     if erreurs:
60.         # on envoie une réponse d'erreur au client
61.         résultats = {"réponse": {"erreurs": erreurs}}
62.         return résultats, status.HTTP_400_BAD_REQUEST
63.
64.     # pas d'erreurs, on peut travailler
65.     # calcul de l'impôt
66.     taxpayer = TaxPayer().fromdict({'marié': marié, 'enfants': enfants, 'salaire': salaire})
67.     config["layers"]["métier"].calculate_tax(taxpayer, config["admindata"])
68.     # on envoie la réponse au client
69.     return {"réponse": {"result": taxpayer.asdict()}}, status.HTTP_200_OK
```

- ligne 9 : la fonction **[execute]** reçoit deux paramètres :
  - **[request]** : la requête HTTP du client ;
  - **[config]** : le dictionnaire de configuration de l'application ;

Le script **[server_02]** est le suivant :

```
1.  # on attend un paramètre mysql ou pgres
2.  import sys
3.  syntaxe = f"{sys.argv[0]} mysql / pgres"
4.  erreur = len(sys.argv) != 2
5.  if not erreur:
6.      sgbd = sys.argv[1].lower()
7.      erreur = sgbd != "mysql" and sgbd != "pgres"
8.  if erreur:
9.      print(f"syntaxe : {syntaxe}")
10.     sys.exit()
11.
12. # on configure l'application
13. import config
14. config = config.configure({'sgbd': sgbd})
15.
16. # dépendances
```

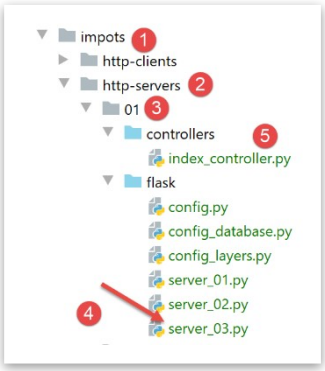
```
17. from ImpôtsError import ImpôtsError
18. from flask import request
19. from myutils import json_response
20. from flask import Flask
21. import index_controller
22.
23. # récupération des données de l'administration fiscale
24. try:
25.     # admindata sera une donnée de portée application en lecture seule
26.     config['admindata'] = config["layers"]["dao"].get_admindata()
27. except ImpôtsError as erreur:
28.     print(f"L'erreur suivante s'est produite : {erreur}")
29.     sys.exit(1)
30.
31. # application Flask
32. app = Flask(__name__)
33.
34.
35. # Home URL : /?marié=xx&enfant=yy&salaire=zz
36. @app.route('/', methods=['GET'])
37. def index():
38.     # on exécute la requête
39.     résultat, statusCode = index_controller.execute(request, config)
40.     # on envoie la réponse
41.     return json_response(résultat, statusCode)
42.
43.
44. # main uniquement
45. if __name__ == '__main__':
46.     # on lance le serveur
47.     app.config.update(ENV="development", DEBUG=True)
48.     app.run()
```

- ligne 36-41 : le traitement de la route / ;
- ligne 39 : utilisation de la fonction **[IndexController.execute]** ;

Nous utiliserons désormais cette technique : chaque route sera traitée par un module qui lui sera propre.

Les résultats d'exécution sont les mêmes que pour la version 1.

23.2.3 Version 3



La version 3 introduit la notion d'authentification.

Le script **[server_03]** devient le suivant :

```
1.  # on attend un paramètre mysql ou pgres
2.  import sys
3.  syntaxe = f"{sys.argv[0]} mysql / pgres"
4.  erreur = len(sys.argv) != 2
5.  if not erreur:
6.      sgbd = sys.argv[1].lower()
7.      erreur = sgbd != "mysql" and sgbd != "pgres"
8.  if erreur:
9.      print(f"syntaxe : {syntaxe}")
10.     sys.exit()
11.
```

```
12. # on configure l'application
13. import config
14. config = config.configure({'sgbd': sgbd})
15. 
16. # dépendances
17. from ImpôtsError import ImpôtsError
18. from flask import request
19. from myutils import json_response
20. from flask import Flask
21. from flask_httpauth import HTTPBasicAuth
22. import index_controller
23. 
24. # récupération des données de l'administration fiscale
25. try:
26.     # config['admindata'] sera une donnée de portée application en lecture seule
27.     config["admindata"] = config["layers"]["dao"].get_admindata()
28. except ImpôtsError as erreur:
29.     print(f"L'erreur suivante s'est produite : {erreur}")
30.     sys.exit(1)
31. 
32. # gestionnaire d'authentification
33. auth = HTTPBasicAuth()
34. 
35. 
36. # méthode d'authentification
37. @auth.verify_password
38. def verify_credentials(login: str, password: str) -> bool:
39.     # liste des utilisateurs
40.     users = config['users']
41.     # on parcourt cette liste
42.     for user in users:
43.         if user['login'] == login and user['password'] == password:
44.             return True
45.     # on n'a pas trouvé
46.     return False
47. 
48. 
49. # application Flask
50. app = Flask(__name__)
51. 
52. 
53. # Home URL : /?marié=xx&enfant=yy&salaire=zz
54. @app.route('/', methods=['GET'])
55. @auth.login_required
56. def index():
57.     # on exécute la requête
58.     résultat, statusCode = index_controller.execute(request, config)
59.     # on envoie la réponse
60.     return json_response(résultat, statusCode)
61. 
62. 
63. # main uniquement
64. if __name__ == '__main__':
65.     # on lance le serveur
66.     app.config.update(ENV="development", DEBUG=True)
67.     app.run()
```

- ligne 21 : on importe un gestionnaire d'authentification. Il existe divers types d'authentification auprès d'un serveur web. Celle que nous utilisons ici s'appelle **[HTTP Basic]**. Chaque type d'authentification suit un dialogue client / serveur précis ;
- ligne 33 : on crée une instance du gestionnaire d'authentification ;
- ligne 37 : l'annotation **[@auth.verify_password]** tague la fonction à exécuter lorsque le gestionnaire d'authentification veut vérifier les login / password transmis par le client selon le protocole **[HTTP Basic]** ;
- ligne 55 : l'annotation **[@auth.login_required]** tague une route pour laquelle le client web doit être authentifié. Si le client web n'a pas encore envoyé ses identifiants, le serveur web va automatiquement les lui demander selon le protocole HTTP basic ;

Le module **[flask_httpauth]** doit être installé :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\impots\http-servers\01\flask>pip install flask_httpauth
2.  Collecting flask_httpauth
3.    Downloading Flask_HTTPAuth-4.1.0-py2.py3-none-any.whl (5.8 kB)
4.  Requirement already satisfied: Flask in c:\data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\lib\site-packages
    (from flask_httpauth) (1.1.2)
5.  Requirement already satisfied: itsdangerous>=0.24 in c:\data\st-2020\dev\python\cours-2020\python3-
    flask-2020\venv\lib\site-packages (from Flask->flask_httpauth) (1.1.0)
```

```
6.  Requirement already satisfied: click>=5.1 in c:\data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\lib\site-
    packages (from Flask->flask_httpauth) (7.1.2)
7.  Requirement already satisfied: Jinja2>=2.10.1 in c:\data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\lib\site-
    packages (from Flask->flask_httpauth) (2.11.2)
8.  Requirement already satisfied: Werkzeug>=0.15 in c:\data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\lib\site-
    packages (from Flask->flask_httpauth) (1.0.1)
9.  Requirement already satisfied: MarkupSafe>=0.23 in c:\data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\lib\site-
    packages (from Jinja2>=2.10.1->Flask->flask_httpauth) (1.1.1
10. )
11. Installing collected packages: flask-httpauth
12. Successfully installed flask-httpauth-4.1.0
```

Voyons ce qui se passe avec la console Postman. Vous :

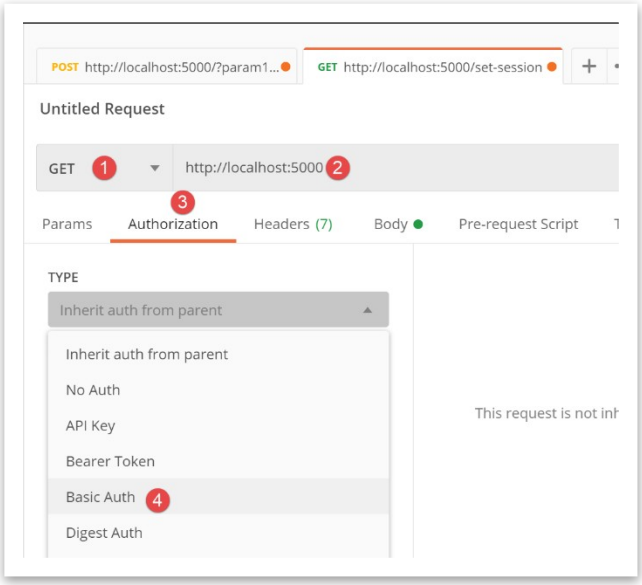
- créez une configuration d'exécution ;
- lancez l'application web ;
- lancez le SGBD de votre choix ;
- demandez l'URL **[/]** avec Postman ;

Le dialogue client / serveur dans la console Postman est le suivant :

```
1.  GET / HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.1
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: e65e2a28-4fe3-423b-88b3-b3e5a83092b1
6.  Host: localhost:5000
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
9.
10. HTTP/1.0 401 UNAUTHORIZED
11. Content-Type: text/html; charset=utf-8
12. Content-Length: 19
13. WWW-Authenticate: Basic realm="Authentication Required"
14. Server: Werkzeug/1.0.1 Python/3.8.1
15. Date: Fri, 17 Jul 2020 07:05:37 GMT
16.
17. Unauthorized Access
```

- ligne 10 : le serveur répond que nous ne sommes pas autorisés à accéder à l'URL **[/]** ;
- ligne 13 : il nous indique le protocole d'authentification à utiliser, ici le protocole dit Authentification Basic ;

Il est possible de configurer Postman pour qu'il envoie les identifiants de l'utilisateur selon le protocole Auth Basic :

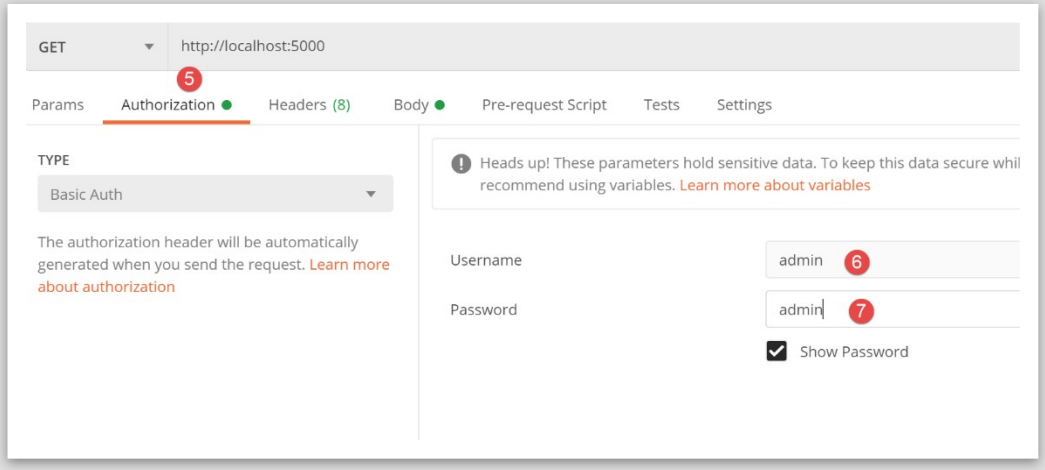


- en **[6-7]** nous mettons les identifiants présents dans le script **[config]** :

```
1.      config['users'] = [
2.          {
3.              "login": "admin",
4.              "password": "admin"
5.          }
6.      ]
```

Le dialogue client / serveur dans la console Postman devient le suivant :

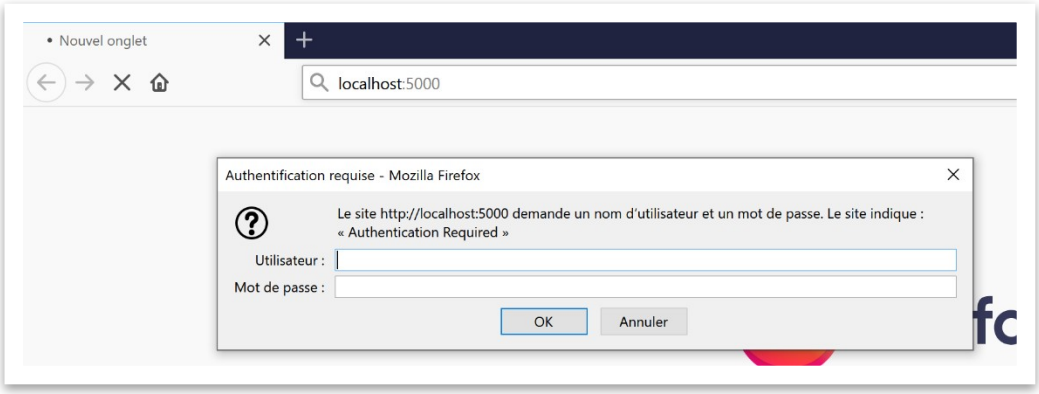
```
1.  GET / HTTP/1.1
2.  Authorization: Basic YWRtaW46YWRtaW4=
3.  User-Agent: PostmanRuntime/7.26.1
4.  Accept: */*
5.  Cache-Control: no-cache
6.  Postman-Token: 5ce20822-e87c-4eef-a2f4-b9eaec38d881
7.  Host: localhost:5000
8.  Accept-Encoding: gzip, deflate, br
9.  Connection: keep-alive
10.
11. HTTP/1.0 400 BAD REQUEST
12. Content-Type: application/json; charset=utf-8
13. Content-Length: 203
```




```
14. Server: Werkzeug/1.0.1 Python/3.8.1
15. Date: Fri, 17 Jul 2020 07:20:01 GMT
16.
17. {"réponse": {"erreurs": ["Méthode GET requise avec les seuls paramètres [marié, enfants, salaire]",
    "paramètre [marié] manquant", "paramètre [enfants] manquant", "paramètre [salaire] manquant"]}}
```

- ligne 2 : le client Postman envoie sous forme codée les identifiants de l'utilisateur **[admin / admin]** ;
- ligne 17 : le serveur répond correctement. Il signale des erreurs car on n'a pas envoyé les paramètres **[marié, enfants, salaire]** (ligne 1) mais il ne signale pas d'erreur d'authentification ;

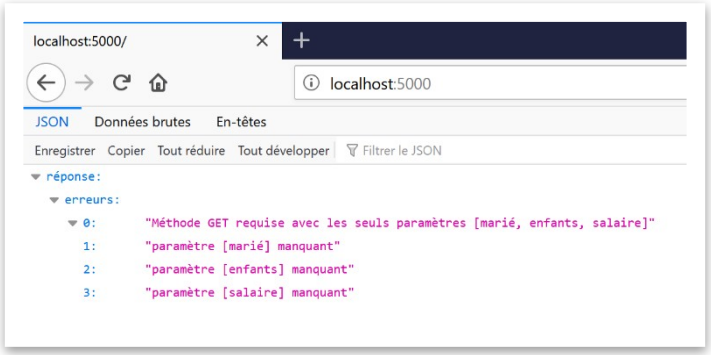
Maintenant demandons l'URL / avec un navigateur (Firefox ci-dessous) :



- comme avec Postman, Firefox a reçu la réponse HTTP du serveur avec les entêtes HTTP :

```
1.  HTTP/1.0 401 UNAUTHORIZED
2.  …
3.  WWW-Authenticate: Basic realm="Authentication Required"
4.  …
```

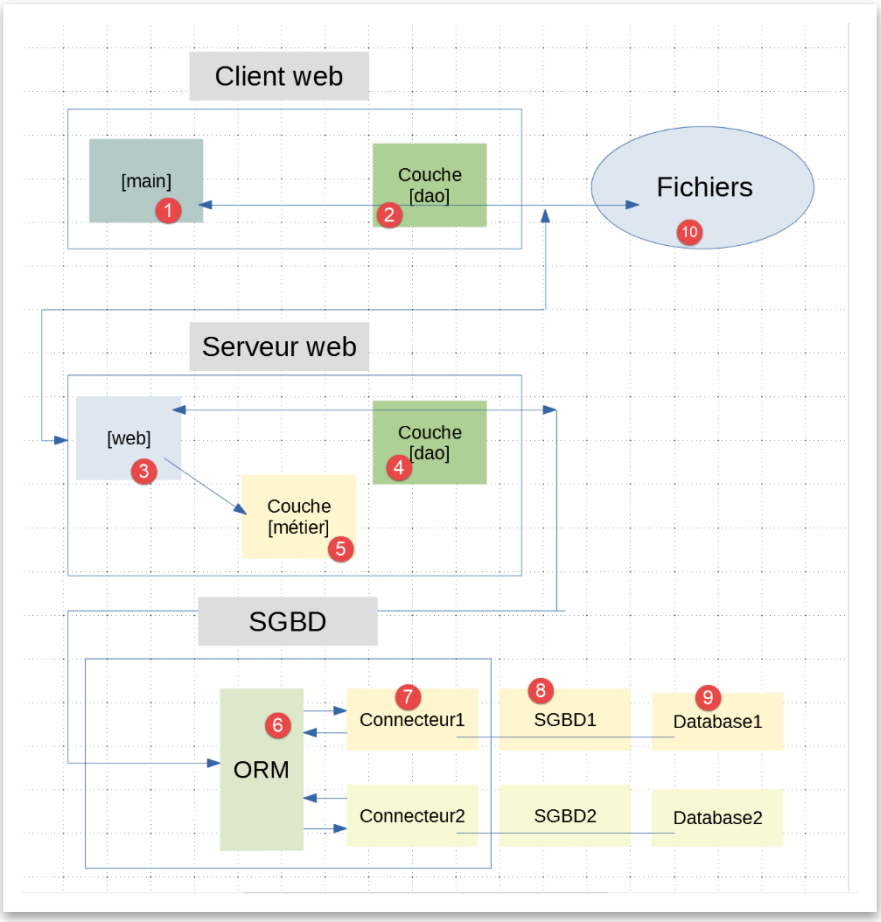
Firefox comme d'autres navigateurs n'arrêtent pas le dialogue lorsqu'il reçoit ces entêtes. Il demandet à l'utilisateur les identifiants réclamés par le serveur. Il suffit ci-dessus de taper admin / admin pour recevoir la réponse du serveur :



23.3 Le client web du serveur de calcul de l'impôt

23.3.1 Introduction

Dans le paragraphe précédent, le client web du serveur de calcul de l'impôt était un navigateur. Dans cette partie, le client web sera un script console. L'architecture devient la suivante :



- le client web est formé des couches **[1-2]** ;

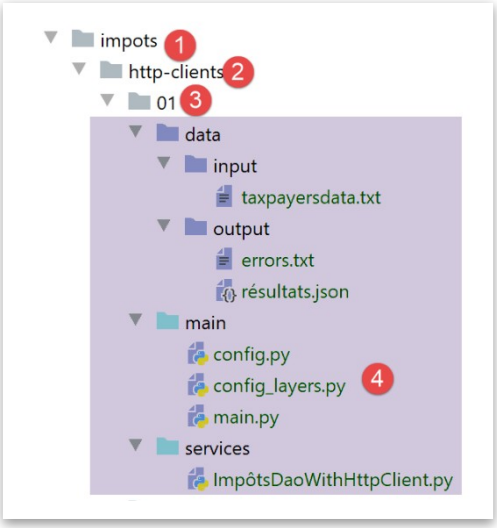
- le serveur web est formé des couches **[3-9]**. Il a été écrit dans le paragraphe précédent ;

Il nous faut donc écrire les couches **[1-2]**.

La couche **[dao] [2]** doit savoir communiquer avec le serveur web **[3]**. Nous connaissons maintenant le protocole HTTP et nous pourrions écrire, avec le module **[pycurl]** déjà étudié par exemple, un script communiquant avec le serveur web **[3]**. Cependant il existe des modules spécialisés dans les dialogues HTTP client / serveur. Nous allons utiliser l'un d'entre-eux, le module **[requests]** :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\impots\http-servers\01\flask>pip install requests
2.  Collecting requests
3.    Downloading requests-2.24.0-py2.py3-none-any.whl (61 kB)
4.       || 61 kB 137 kB/s
5.  Collecting idna<3,>=2.5
6.    Downloading idna-2.10-py2.py3-none-any.whl (58 kB)
7.       || 58 kB 692 kB/s
8.  Collecting chardet<4,>=3.0.2
9.    Downloading chardet-3.0.4-py2.py3-none-any.whl (133 kB)
10.      || 133 kB 1.3 MB/s
11. Collecting urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1
12.   Downloading urllib3-1.25.9-py2.py3-none-any.whl (126 kB)
13.      || 126 kB 1.1 MB/s
14. Collecting certifi>=2017.4.17
15.   Downloading certifi-2020.6.20-py2.py3-none-any.whl (156 kB)
16.      || 156 kB 1.1 MB/s
17. Installing collected packages: idna, chardet, urllib3, certifi, requests
18. Successfully installed certifi-2020.6.20 chardet-3.0.4 idna-2.10 requests-2.24.0 urllib3-1.25.9
```

L'arborescence des scripts du client web est la suivante :



Le script va implémenter l'application de calcul de l'impôt en mode batch décrite dès la |version 1|. La dernière version de cette application est la |version 5|. On rappelle son fonctionnement :

- les contribuables dont on va calculer l'impôt sont rassemblés dans le fichier texte **[taxpayersdata.txt]** :

```
1.  # données valides : id, marié, enfants, salaire
2.  1,oui,2,55555
3.  2,oui,2,50000
4.  3,oui,3,50000
5.  4,non,2,100000
6.  5,non,3,100000
7.  6,oui,3,100000
8.  7,oui,5,100000
9.  8,non,0,100000
10. 9,oui,2,30000
11. 10,non,0,200000
12. 11,oui,3,200000
13. # on crée des lignes erronées
14. # pas assez de valeurs
15. 11,12
16. # des valeurs erronées
17. x,x,x,x
```

- les résultats vont dans deux fichiers :
  - le fichier texte **[errors.txt]** rassemble les erreurs détectées dans le fichier des contribuables :

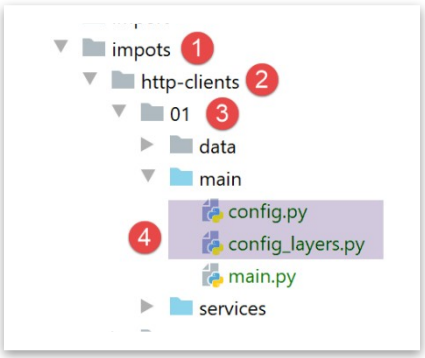
```
Analyse du fichier C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\impots\http-clients\01\main/../data/input/taxpayersdata.txt

Ligne 15, not enough values to unpack (expected 4, got 2)
Ligne 17, MyException[1, L'identifiant d'une entité <class 'TaxPayer.TaxPayer'> doit être un entier >=0]
```

  - le fichier jSON **[résultats.json]** rassemble les résultats des calculs de l'impôt des différents contribuables :

```
[
  {
    "id": 0,
    "marié": "oui",
    "enfants": 2,
    "salaire": 55555,
    "impôt": 2814,
    "surcôte": 0,
    "taux": 0.14,
    "décôte": 0,
    "réduction": 0
  },
  {
    "id": 1,
    "marié": "oui",
    "enfants": 2,
    "salaire": 50000,
    "impôt": 1384,
    "surcôte": 0,
    "taux": 0.14,
    "décôte": 384,
    "réduction": 347
  },
  …
]
```

23.3.2 Configuration du client web



La configuration se fait à l'aide de deux scripts :

- **[config]** qui assure la totalité de la configuration hors couches de l'architecture ;
- **[config_layers]** qui assure la configuration des couches de l'architecture ;

Le script **[config]** est le suivant :

```
1.  def configure(config: dict) -> dict:
2.      import os
3.
4.      # étape 1 ------
5.
6.      # dossier de ce fichier
7.      script_dir = os.path.dirname(os.path.abspath(__file__))
8.
9.      # chemin racine
```

```
10.     root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
11.
12.     # dépendances absolues
13.     absolute_dependencies = [
14.         # dossiers du projet
15.         # BaseEntity, MyException
16.         f"{root_dir}/classes/02/entities",
17.         # InterfaceImpôtsDao, InterfaceImpôtsMétier, InterfaceImpôtsUi
18.         f"{root_dir}/impots/v04/interfaces",
19.         # AbstractImpôtsdao, ImpôtsConsole, ImpôtsMétier
20.         f"{root_dir}/impots/v04/services",
21.         # ImpotsDaoWithAdminDataInDatabase
22.         f"{root_dir}/impots/v05/services",
23.         # AdminData, ImpôtsError, TaxPayer
24.         f"{root_dir}/impots/v04/entities",
25.         # Constantes, tranches
26.         f"{root_dir}/impots/v05/entities",
27.         # ImpôtsDaoWithHttpClient
28.         f"{script_dir}/../services",
29.         # scripts de configuration
30.         script_dir,
31.     ]
32.
33.     # on fixe le syspath
34.     from myutils import set_syspath
35.     set_syspath(absolute_dependencies)
36.
37.     # étape 2 ------
38.     # configuration de l'application avec des constantes
39.     config.update({
40.         "taxpayersFilename": f"{script_dir}/../data/input/taxpayersdata.txt",
41.         "resultsFilename": f"{script_dir}/../data/output/résultats.json",
42.         "errorsFilename": f"{script_dir}/../data/output/errors.txt",
43.         "server": {
44.             "urlServer": "http://127.0.0.1:5000/",
45.             "authBasic": True,
46.             "user": {
47.                 "login": "admin",
48.                 "password": "admin"
49.             }
50.         }
51.     }
52.     )
53.
54.     # étape 3 ------
55.     # instanciation des couches
56.     import config_layers
57.     config['layers'] = config_layers.configure(config)
58.
59.     # on rend la configuation
60.     return config
```

- ligne 1 : la fonction **[configure]** reçoit en paramètre le dictionnaire à remplir avec les informations de configuration. Celui-ci peut être déjà pré-rempli ou vide. Ici, il sera vide ;
- lignes 40-42 : les noms absolus des trois fichiers texte gérés par la couche **[dao]** ;
- lignes 43-50 : associées à la clé **[server]**, les informations que doit connaître la couche **[dao]** sur le serveur web avec lequel elle doit communiquer :
  - ligne 44 : l'URL du service web ;
  - ligne 45 : la clé **[authBasic]** vaut True si l'accès à l'URL nécessite une authentification de type Basic ;
  - lignes 46-49 : les identifiants de l'utilisateur qui va s'authentifier si l'authentification est demandée ;
- lignes 56-57 : on instancie les couches, ici l'unique couche **[dao]** et on met les références des couches dans **[config]** associées à la clé **[layers]** ;

Le script **[config_layers]** est le suivant :

```
1.  def configure(config: dict) -> dict:
2.      # instanciation des couches de l'applicatuon
3.
4.      # couche dao
5.      from ImpôtsDaoWithHttpClient import ImpôtsDaoWithHttpClient
6.      dao = ImpôtsDaoWithHttpClient(config)
7.
8.      # on rend la configuation des couches
9.      return {
10.         "dao": dao
```

```
11.     }
```

- ligne 1 : la fonction **[configure]** reçoit le dictionnaire qui configure l'application ;
- lignes 4-6 : la couche **[dao]** est instanciée. Ligne 6, on lui passe la configuration de l'application dans lequel elle trouvera les informations dont elle a besoin ;
- lignes 8-11 : on rend un dictionnaire dans lequel on a mis la référence de la couche **[dao]** ;

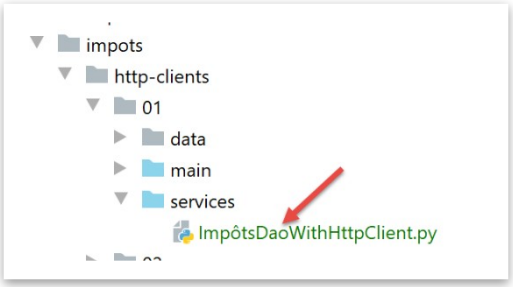
23.3.3 Le script principal [main]

Le script principal **[main]** est une variante de celui de la |version 5| :

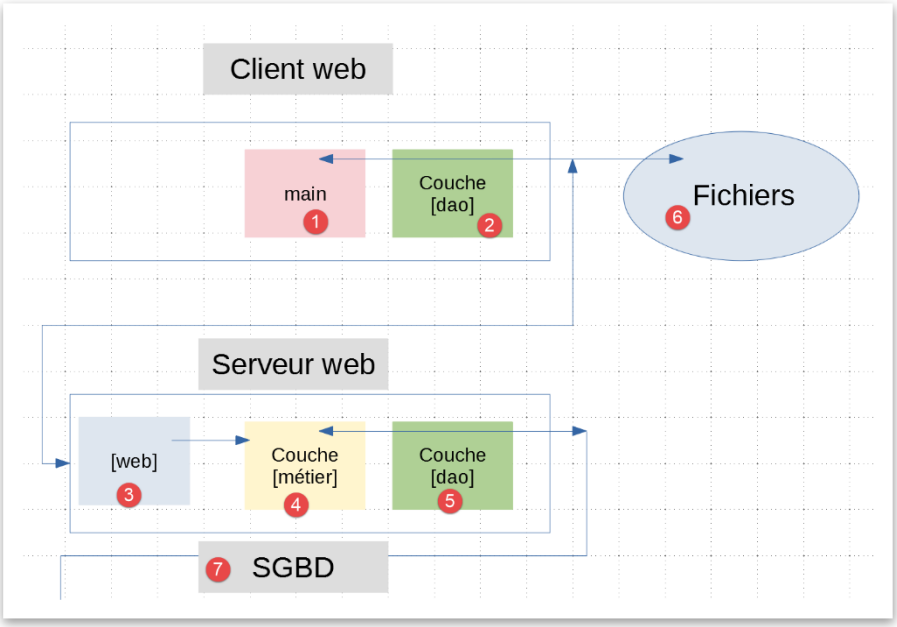
```
1.  # on configure l'application
2.  import config
3.  config = config.configure({})
4.
5.  # dépendances
6.  from ImpôtsError import ImpôtsError
7.
8.  # code
9.  try:
10.     # on récupère la couche [dao]
11.     dao = config["layers"]["dao"]
12.     # lecture des données des contribuables
13.     taxpayers = dao.get_taxpayers_data()["taxpayers"]
14.     # des contribuables ?
15.     if not taxpayers:
16.         raise ImpôtsError(f"Pas de contribuables valides dans le fichier {config['taxpayersFilename']}")
17.     # calcul de l'impôt des contribuables
18.     for taxpayer in taxpayers:
19.         # taxpayer est à la fois un paramètre d'entrée et de sortie
20.         # taxpayer va être modifié
21.         dao.calculate_tax(taxpayer)
22.     # écriture des résultats dans un fichier texte
23.     dao.write_taxpayers_results(taxpayers)
24. except ImpôtsError  as erreur:
25.     # affichage de l'erreur
26.     print(f"L'erreur suivante s'est produite : {erreur}")
27. finally:
28.     # terminé
29.     print("Travail terminé...")
```

- lignes 2-3 : l'application est configurée ;
- ligne 13 : la couche **[dao]** fournit la liste des contribuables dont on doit calculer l'impôt ;
- ligne 21 : la couche **[dao]** calcule l'impôt de chacun d'eux ;
- ligne 23 : les résultats sont enregistrés dans un fichier jSON ;

23.3.4 Implémentation de la couche [dao]



Revenons sur l'architecture client / serveur utilisée :



- en **[2, 6]**, on voit que la couche **[dao]** a deux rôles :
    - elle accède au système de fichiers à la fois pour lire les données des contribuables et écrire les résultats des calculs de l'impôt. Nous avons déjà une classe |AbstractImpôtsDao| qui sait faire cela. Elle a été utilisée dès la |version 4| ;
    - elle dialogue avec le serveur web **[3]** ;

Dans la |version 5|, le script principal **[main] [1]** dialoguait directement avec la couche **[métier] [4]**. On voudrait ne pas changer ce script. Pour cela, on va faire en sorte que la couche **[dao] [2]** implémente l'interface de la couche **[métier] [4]**. Ainsi le script principal **[main]** aura l'impression de dialoguer directement avec la couche **[métier] [4]** et pourra ignorer complètement que celle-ci se situe sur une autre machine.

Une définition de la classe implémentant la couche **[dao] [2]** pourrait être celle-ci :

```
class ImpôtsDaoWithHttpClient(AbstractImpôtsDao, InterfaceImpôtsMétier):
```

- la classe **[ImpôtsDaoWithHttpClient]** :
    - hérite de la classe **[AbstractImpôtsDao]** ce qui va lui permettre de gérer le dialogue avec le système de fichiers **[6]** ;
    - implémente l'interface **[InterfaceImpôtsMétier]** pour ne pas avoir à changer le script principal **[main]** de la |version 5| ;

Le code complet de la classe **[ImpôtsDaoWithHttpClient]** est le suivant :

```
1.  # imports
2.  import requests
3.  from flask_api import status
4.
5.  from AbstractImpôtsDao import AbstractImpôtsDao
6.  from AdminData import AdminData
7.  from ImpôtsError import ImpôtsError
8.  from InterfaceImpôtsMétier import InterfaceImpôtsMétier
9.  from TaxPayer import TaxPayer
10.
11.
12. class ImpôtsDaoWithHttpClient(AbstractImpôtsDao, InterfaceImpôtsMétier):
13.
14.     # constructeur
15.     def __init__(self, config: dict):
16.         # initialisation parent
17.         AbstractImpôtsDao.__init__(self, config)
18.         # mémorisation paramètres
19.         self.__config_server = config["server"]
20.
21.     # méthode inutilisée de [AbstractImpôtsDao]
```

```
22.     def get_admindata(self) -> AdminData:
23.         pass
24.
25.     # calcul de l'impôt
26.     def calculate_tax(self: object, taxpayer: TaxPayer, admindata: AdminData = None):
27.         # on laisse remonter les exceptions
28.         # paramètres du get
29.         params = {"marié": taxpayer.marié, "enfants": taxpayer.enfants, "salaire": taxpayer.salaire}
30.         # connexion avec authentification Auth Basic ?
31.         if self.__config_server['authBasic']:
32.             response = requests.get(
33.                 # URL du serveur interrogé
34.                 self.__config_server['urlServer'],
35.                 # paramètres de l'URL
36.                 params=params,
37.                 # authentification Basic
38.                 auth=(
39.                     self.__config_server["user"]["login"],
40.                     self.__config_server["user"]["password"]))
41.         else:
42.             # connexion sans authentification Auth Basic
43.             response = requests.get(self.__config_server['urlServer'], params=params)
44.         # vérification
45.         print(response.text)
46.         # code de statut de la réponse HTTP
47.         status_code = response.status_code
48.         # on met la réponse jSON dans un dictionnaire
49.         résultat = response.json()
50.         # erreur si code de statut différent de 200 OK
51.         if status_code != status.HTTP_200_OK:
52.             # on sait que les erreurs ont été associées à la clé [erreurs] de la réponse
53.             raise ImpôtsError(87, résultat['réponse']['erreurs'])
54.         # on sait que le résultat a été associé à la clé [result] de la réponse
55.         # on modifie le paramètre d'entrée avec ce résultat
56.         taxpayer.fromdict(résultat["réponse"]["result"])
```

- lignes 21-23 : la classe **[AbstractImpôtsDao]** (ligne 12) possède une méthode abstraite **[get_admindata]**. Nous sommes obligés de l'implémenter même si on ne s'en sert pas (admindata est géré par le serveur pas par le client) ;
- ligne 26 : la méthode **[calculate_tax]** appartient à l'interface **[InterfaceImpôtsMétier]** (ligne 12). il nous faut l'implémenter ;
- ligne 15 : le constructeur reçoit en unique paramètre le dictionnaire de la configuration de l'application ;
- lignes 16-17 : la classe parent **[AbstractImpôtsDao]** est initialisée en lui passant, là également, la configuration de l'application. Elle y trouvera les noms des trois fichiers texte qu'elle a à gérer ;
- lignes 18-19 : on mémorise localement dans la classe les informations concernant le serveur web de calcul de l'impôt ;
- ligne 26 : la méthode **[calculate_tax]** reçoit en paramètre un objet de type |Taxpayer|. Pour respecter la signature de la méthode **[InterfaceImpôtsMétier.calculate_tax]**, elle reçoit également un paramètre **[admindata]** qui est censé encapsuler les données de l'administration fiscale. Côté client, on n'a pas ces données. Ce paramètre restera toujours à **[None]**. Cette contorsion fait dire que la classe **[ImpôtsMétier]** a été initialement mal écrite :
  - la signature de **[calculate_tax]** aurait dû être simplement :

```
def calculate_tax(self, taxpayer: TaxPayer)
```

  et le paramètre **[admindata : AdminData]** aurait dû être passé au constructeur de la classe ;
- ligne 27 : le code de la méthode **[calculate_tax]** n'a pas été encapsulé dans un try / catch / finally. Cela veut dire que les éventuelles exceptions ne seront pas gérées et remonteront au code appelant, en l'occurrence le script **[main]**. Celui-ci arrête bien toutes les exceptions remontant de la couche **[dao]** ;
- ligne 28 : le calcul de l'impôt se fait côté serveur. Il va donc falloir communiquer avec lui. On le fait avec le module **[requests]** importé ligne 2 ;
- lignes 31-43 : pour envoyer une requête GET au serveur web, on utilise la méthode **[requests.get]** :
  - lignes 33-34 : le 1er paramètre de la méthode est l'URL à contacter ;
  - lignes 35-40 : les deux autres paramètres sont des paramètres nommés dont l'ordre n'importe pas ;
  - lignes 35-36 : la valeur du paramètre nommé **[params]** doit être un dictionnaire contenant les informations à mettre dans l'URL sous la forme **[/url ?param1=valeur1&param2=valeur2&…]** ;
    - ligne 29 : le dictionnaire contenant les trois paramètres **[marié, enfants, salaire]** que le serveur web attend. On n'a pas à s'occuper de l'encodage (appelé urlencoded) que doivent subir ces paramètres. **[requests]** s'en occupe ;
  - lignes 37-40 : le paramètre nommé **[auth]** est un tuple de deux éléments (login, password). Il représente les identifiants d'une authentification de type Basic ;
- lignes 44-45 : ces deux lignes n'ont qu'un but pédagogique (on les mettra en commentaires lorsque le débogage sera terminé) :
  - **[response]** représente la réponse HTTP du serveur ;

    - **[response.text]** représente le texte du document encapsulé dans cette réponse. En phase de débogage, il est utile de vérifier ce que le serveur nous a envoyé ;
- ligne 47 : **[response.status_code]** est le code de statut HTTP de la réponse reçue. Notre serveur n'en envoie que trois :
    - 200 OK
    - 400 BAD REQUEST
    - 500 INTERNAL SERVER ERROR
- ligne 49 : notre serveur envoie toujours du jSON même en cas d'erreur. La fonction **[response.json()]** crée un dictionnaire à partir de la chaîne jSON reçue. Rappelons les deux formes possibles pour la chaîne jSON :

```
{"réponse": {"erreurs": ["Méthode GET requise avec les seuls paramètres [marié, enfants, salaire]",
"paramètre [marié] manquant", "paramètre [enfants] manquant", "paramètre [salaire] manquant"]}}
```

```
{"réponse": {"result": {"id": 0, "marié": "oui", "enfants": 3, "salaire": 200000, "impôt": 42842,
"surcôte": 17283, "taux": 0.41, "décôte": 0, "réduction": 0}}}
```

- lignes 51-53 : si le code de statut n'est pas 200 alors on lance une exception avec les messages d'erreur encapsulés dans la réponse ;
- ligne 56 : on récupère le dictionnaire produit par le calcul de l'impôt et on l'utilise pour mettre à jour le paramètre d'entrée **[taxpayer]** ;

23.3.5 Exécution

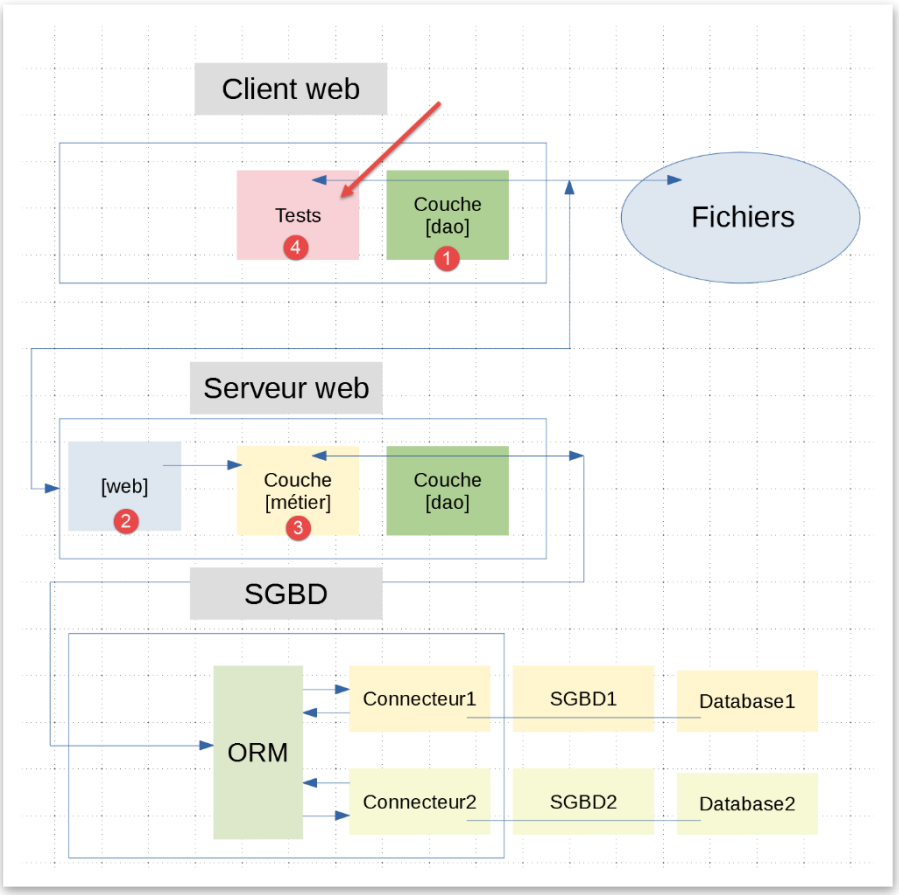
Pour exécuter le client :

- lancer le serveur **[server_03]** avec le SGBD de votre choix ;
- exécuter le script **[main]** du client ;

Les résultats seront trouvés dans le dossier **[data/output]**. Ce sont les mêmes que pour la version 5.

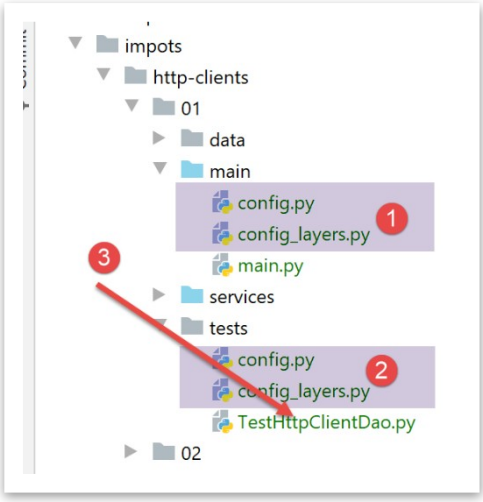
23.4 Tests de la couche [dao]

Revenons à l'architecture de l'application client / serveur :



- dans le client écrit, nous nous sommes débrouillés pour que la couche **[dao] [1]** offre la même interface que la couche **[métier] [3]**. Nous allons donc utiliser en **[4]**, la classe de tests |TestDaoMétier| déjà étudiée pour tester la couche **[métier] [3]** ;

La classe de test sera exécutée dans l'environnement suivant :



- la configuration **[2]** est identique à la configuration **[1]** que nous venons d'étudier ;

La classe de test **[TestHttpClientDao]** est la suivante :

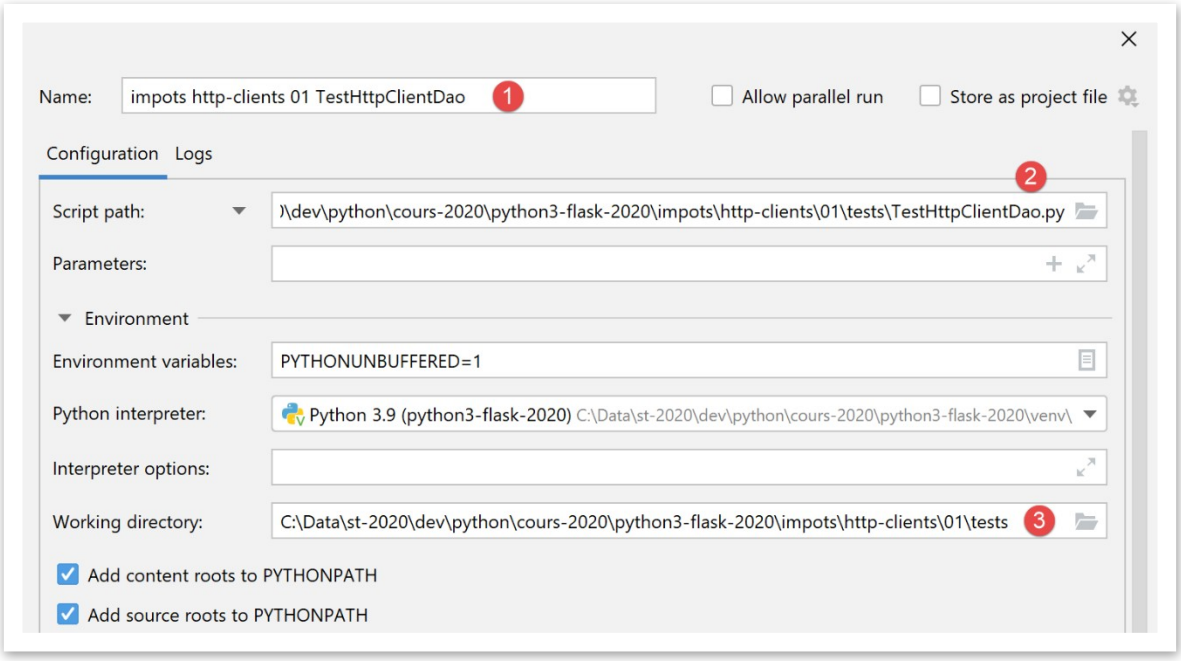
```
1.  import unittest
2.
3.
4.  class TestHttpClientDao(unittest.TestCase):
5.
6.      def test_1(self) -> None:
7.          from TaxPayer import TaxPayer
8.
9.          # {'marié': 'oui', 'enfants': 2, 'salaire': 55555,
10.         # 'impôt': 2814, 'surcôte': 0, 'décôte': 0, 'réduction': 0, 'taux': 0.14}
11.         taxpayer = TaxPayer().fromdict({"marié": "oui", "enfants": 2, "salaire": 55555})
12.         dao.calculate_tax(taxpayer)
13.         # vérification
14.         self.assertAlmostEqual(taxpayer.impôt, 2815, delta=1)
15.         self.assertEqual(taxpayer.décôte, 0)
16.         self.assertEqual(taxpayer.réduction, 0)
17.         self.assertAlmostEqual(taxpayer.taux, 0.14, delta=0.01)
18.         self.assertEqual(taxpayer.surcôte, 0)
19.
20.     …
21.
22.     def test_11(self) -> None:
23.         from TaxPayer import TaxPayer
24.
25.         # {'marié': 'oui', 'enfants': 3, 'salaire': 200000,
26.         # 'impôt': 42842, 'surcôte': 17283, 'décôte': 0, 'réduction': 0, 'taux': 0.41}
27.         taxpayer = TaxPayer().fromdict({'marié': 'oui', 'enfants': 3, 'salaire': 200000})
28.         dao.calculate_tax(taxpayer)
29.         # vérifications
30.         self.assertAlmostEqual(taxpayer.impôt, 42842, 1)
31.         self.assertEqual(taxpayer.décôte, 0)
32.         self.assertEqual(taxpayer.réduction, 0)
33.         self.assertAlmostEqual(taxpayer.taux, 0.41, delta=0.01)
34.         self.assertAlmostEqual(taxpayer.surcôte, 17283, delta=1)
35.
36.
37. if __name__ == '__main__':
38.
39.     # on configure l'application
40.     import config
41.     config = config.configure({})
42.
43.     # couche dao
44.     dao = config['layers']['dao']
```

```
45.
46.     # on exécute les méthodes de test
47.     print("tests en cours...")
48.     unittest.main()
```

Cette classe est analogue à |celle| déjà étudiée dans la version 4 de l'application.

- lignes 40-41 : on configure l'environnement des tests ;
- ligne 44 : on récupère une référence sur la couche **[dao]** ;
- lignes 47-48 : on exécute les tests ;

Pour exécuter les tests, on crée une |configuration d'exécution| :



- on crée une configuration d'exécution pour un script console pas pour un test UnitTest ;

Lorsqu'on exécute cette configuration, on obtient les résultats suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/http-clients/01/tests/TestHttpClientDao.py
2.  tests en cours...
3.  {"réponse": {"result": {"marié": "oui", "enfants": 2, "salaire": 55555, "impôt": 2814, "surcôte": 0,
    "taux": 0.14, "décôte": 0, "réduction": 0}}}
4.  ....{"réponse": {"result": {"marié": "non", "enfants": 0, "salaire": 200000, "impôt": 64210, "surcôte":
    7498, "taux": 0.45, "décôte": 0, "réduction": 0}}}
5.  {"réponse": {"result": {"marié": "oui", "enfants": 3, "salaire": 200000, "impôt": 42842, "surcôte": 17283,
    "taux": 0.41, "décôte": 0, "réduction": 0}}}
6.  {"réponse": {"result": {"marié": "oui", "enfants": 2, "salaire": 50000, "impôt": 1384, "surcôte": 0,
    "taux": 0.14, "décôte": 384, "réduction": 347}}}
7.  {"réponse": {"result": {"marié": "oui", "enfants": 3, "salaire": 50000, "impôt": 0, "surcôte": 0, "taux":
    0.14, "décôte": 720, "réduction": 0}}}
8.  ...{"réponse": {"result": {"marié": "non", "enfants": 2, "salaire": 100000, "impôt": 19884, "surcôte":
    4480, "taux": 0.41, "décôte": 0, "réduction": 0}}}
9.  {"réponse": {"result": {"marié": "non", "enfants": 3, "salaire": 100000, "impôt": 16782, "surcôte": 7176,
    "taux": 0.41, "décôte": 0, "réduction": 0}}}
10. {"réponse": {"result": {"marié": "oui", "enfants": 3, "salaire": 100000, "impôt": 9200, "surcôte": 2180,
    "taux": 0.3, "décôte": 0, "réduction": 0}}}
11. {"réponse": {"result": {"marié": "oui", "enfants": 5, "salaire": 100000, "impôt": 4230, "surcôte": 0,
    "taux": 0.14, "décôte": 0, "réduction": 0}}}
12. {"réponse": {"result": {"marié": "non", "enfants": 0, "salaire": 100000, "impôt": 22986, "surcôte": 0,
    "taux": 0.41, "décôte": 0, "réduction": 0}}}
13. ....
14. {"réponse": {"result": {"marié": "oui", "enfants": 2, "salaire": 30000, "impôt": 0, "surcôte": 0, "taux":
    0.0, "décôte": 0, "réduction": 0}}}
15. ----------------------------------------------------------------------
16. Ran 11 tests in 0.130s
17.
```

```
18. OK
19.
20. Process finished with exit code 0
```

Les 11 tests ont réussi.

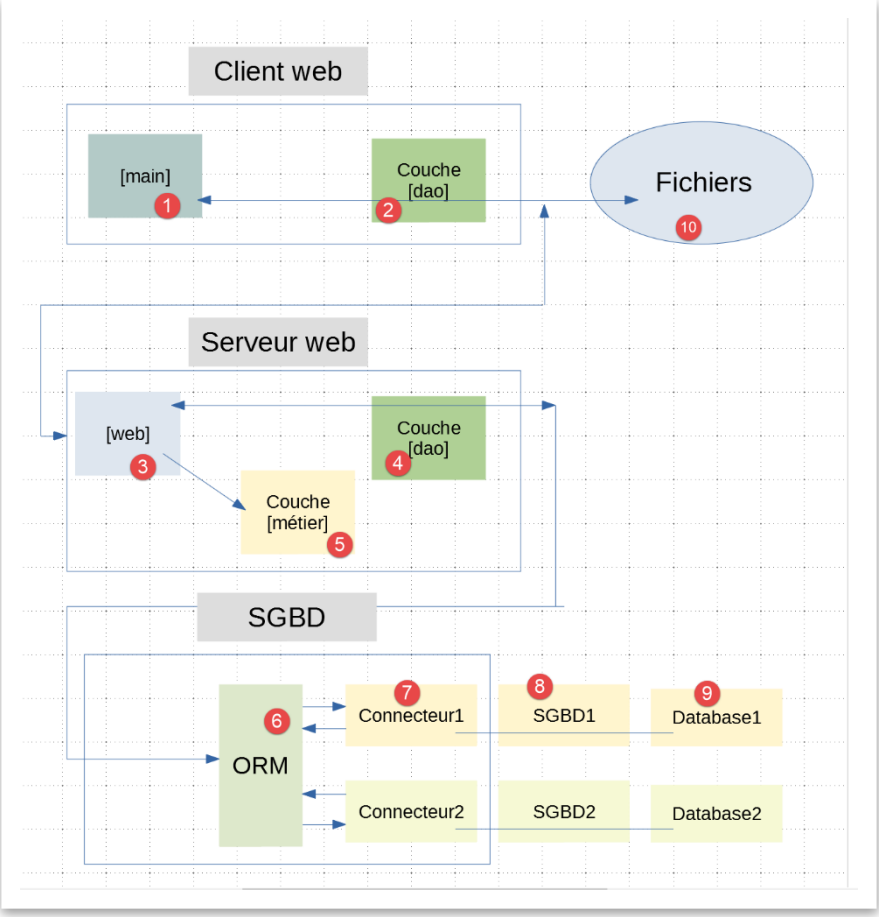
24 Exercice d'application : version 7

24.1 Introduction

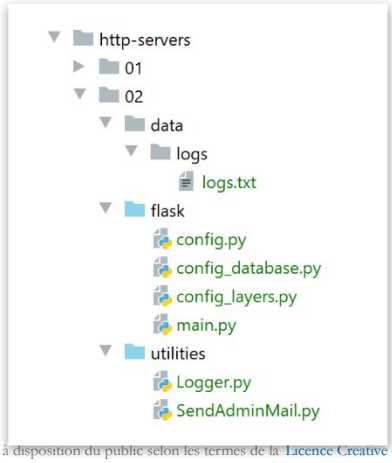
La version 7 de l'application de calcul de l'impôt est identique à la version 6 aux détails près suivants :

- le client web va lancer simultanément plusieurs requêtes HTTP. Dans la version précédente ces requêtes étaient lancées séquentiellement. Le serveur ne traitait alors à tout moment qu'une unique requête ;
- le serveur sera multi-threadé : il pourra traiter plusieurs requêtes simultanément ;
- pour suivre l'exécution de ces requêtes, on va doter le serveur web d'un logueur avec lequel on loguera dans un fichier texte les moments importants du traitement des requêtes ;
- le serveur enverra un mail à l'administrateur de l'application lorsqu'il rencontrera un problème qui l'empêchera de se lancer, typiquement un problème avec la base de données associée au serveur web ;

L'architecture de l'application ne change pas :

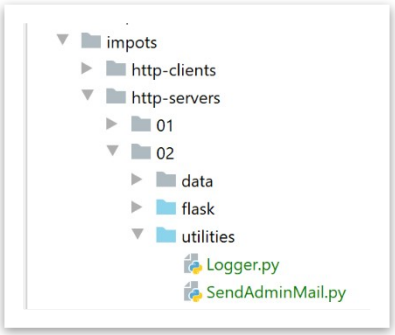


L'arborescence des scripts est la suivante :



Le dossier **[http-servers/02]** est d'abord obtenue par recopie du dossier **[http-servers/01]**. On y fait ensuite des modifications.

24.2 Les utilitaires



24.2.1 La classe [Logger]

La classe **[Logger]** va permettre de loguer dans un fichier texte certaines actions du serveur web :

```
1.  import codecs
2.  import threading
3.  from datetime import date, datetime
4.  from threading import current_thread
5.
6.  from ImpôtsError import ImpôtsError
7.
8.
9.  class Logger:
10.     # attribut de classe
11.     verrou = threading.RLock()
12.
13.     # constructeur
14.     def __init__(self, logs_filename: str):
15.         try:
16.             # on ouvre le fichier en mode append (a)
17.             self.__resource = codecs.open(logs_filename, "a", "utf-8")
18.         except BaseException as erreur:
19.             raise ImpôtsError(18, f"{erreur}")
20.
21.     # écriture d'un log
22.     def write(self, message: str):
23.         # date / heure du moment
24.         today = date.today()
25.         now = datetime.time(datetime.now())
26.         # nom du thread
27.         thread_name = current_thread().name
28.         # on ne veut pas être dérangé pendant qu'on écrira dans le fichier de logs
29.         # on demande l'objet de synchronisation (= le verrou) de la classe - un seul thread l'obtiendra
30.         Logger.verrou.acquire()
31.         try:
32.             # écriture du log
33.             self.__resource.write(f"{today} {now}, {thread_name} : {message}")
34.             # écriture immédiate - sinon le texte ne sera écrit que lors de la fermeture du flux d'écriture
35.             # or on veut suivre les logs dans le temps
36.             self.__resource.flush()
37.         finally:
38.             # on libère l'objet de synchronisation (= le verrou) pour qu'un autre thread puisse l'obtenir
39.             Logger.verrou.release()
40.
41.     # libération des ressources
42.     def close(self):
43.         # fermeture du fichier
44.         if self.__resource:
45.             self.__resource.close()
```

- lignes 10-11 : on définit un attribut de classe. Un attribut de classe est une propriété partagée par toutes les instances de la classe. On la référence par la notation **[Classe.attribut_de_classe]** (lignes 30, 39). L'attribut de classe **[verrou]** sera un objet de synchronisation pour tous les threads exécutant le code des lignes 31-36 ;

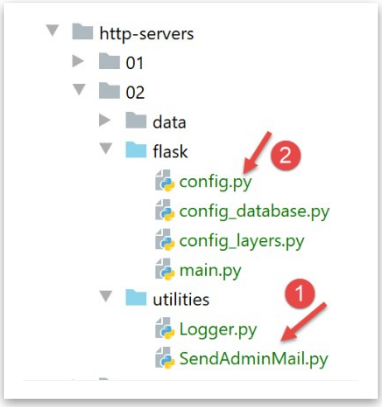
- lignes 14-19 : le constructeur reçoit le nom absolu du fichier de logs. Ce fichier est alors ouvert et le descripteur de fichier récupéré est mémorisé dans la classe ;
- ligne 17 : le fichier de logs est ouvert en mode 'append' (a). Chaque ligne écrite le sera à la fin du fichier ;
- lignes 22-39 : la méthode **[write]** permet d'écrire dans le fichier de logs un message passé en paramètre. A celui-ci sont accolées deux informations :
  - ligne 24 : la date du jour ;
  - ligne 25 : l'heure du moment ;
  - ligne 27 : le nom du thread qui écrit le log. Il ne faut pas oublier ici qu'une application web sert plusieurs utilisateurs à la fois. Toute requête se voit attribuer un thread pour l'exécuter. Si ce thread est mis en pause, typiquement pour une opération d' entrée / sortie (réseau, fichiers, base de données), alors le processeur sera donné à un autre thread. A cause de ces interruptions possibles, on ne peut pas être sûrs qu'un thread va réussir à écrire une ligne dans le fichier de logs sans être interrompu. On risque alors de voir des logs de deux threads différents se mélanger. Le risque est faible, peut-être même nul, mais on a néanmoins décidé de montrer comment synchroniser l'accès de deux threads à une ressource commune, ici le fichier de logs ;
- ligne 30 : avant d'écrire, le thread demande la clé de la porte d'entrée. La clé demandée est celle créée ligne 11. Elle est effectivement unique : un attribut de classe est unique pour toutes les instances de la classe ;
  - au temps T1, un thread Thread1 obtient la clé. Il peut alors exécuter la ligne 33 ;
  - au temps T2, le thread Thread1 est mis en pause avant même d'avoir terminé l'écriture du log ;
  - au temps T3, le thread Thread2 qui a obtenu le processeur doit lui aussi écrire un log. Il arrive ainsi à la ligne 30 où il demande la clé de la porte d'entrée. On lui répond qu'un autre thread l'a déjà. Il est alors automatiquement mis en pause. Il en sera ainsi de tous les threads qui demanderont cette clé ;
  - au temps T4, le thread Thread1 qui avait été mis en pause retrouve le processeur. Il termine alors l'écriture du log ;
- lignes 32-36 : l'écriture dans le fichier de logs se fait en deux temps :
  - ligne 33 : le descripteur de fichier obtenu ligne 17 travaille avec un buffer. L'opération **[write]** de la ligne 33 écrit dans ce buffer mais pas directement dans le fichier. Le buffer est ensuite vidé dans le fichier dans certaines conditions :
    - le buffer est plein ;
    - le descripteur de fichier subit une opération **[close]** ou **[flush]** ;
  - ligne 36 : on force l'écriture de la ligne de log dans le fichier. On fait cela parce qu'on veut voir s'intercaler entre eux les logs des différents threads. Si on ne fait pas ça, les logs d'un thread seront tous écrits en même temps lors de la fermeture du descripteur, ligne 45. Il serait alors beaucoup plus difficile de voir que certains threads ont été arrêtés : il faudrait regarder les heures dans les logs ;
- ligne 39 : le thread Thread1 rend la clé qu'on lui avait donnée. Elle va pouvoir être donnée à un autre thread ;
- ligne 22 : la méthode **[write]** est donc synchronisée : un seul thread à la fois écrit dans le fichier de logs. La clé du dispositif est la ligne 30 : quoiqu'il arrive, seul un thread récupère la clé de passage à la ligne suivante. Il la garde tant qu'il ne la rend pas (ligne 39) ;
- lignes 41-45 : la méthode **[close]** permet de libérer les ressources allouées au descripteur du fichier de logs ;

Les logs écrits dans le fichier de logs auront l'allure suivante :

```
2020-07-22 20:03:52.992152, Thread-2 : …
```

24.2.2 La classe [SendAdminMail]

La classe **[SendAminMail]** permet d'envoyer un message à l'administrateur de l'application lorsque celle-ci 'plante'.



La classe **[SendAdminMail]** est configurée dans le script **[config] [2]** de la façon suivante :

```
1.          # config serveur SMTP
2.          "adminMail": {
3.              # serveur SMTP
4.              "smtp-server": "localhost",
```

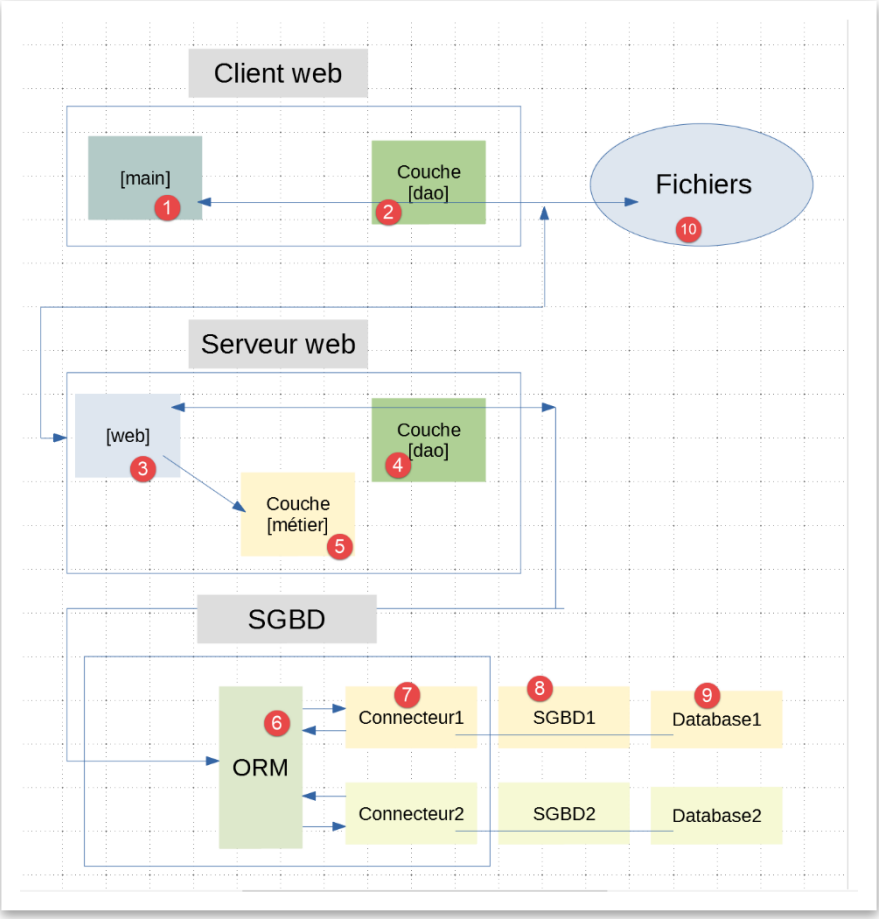
```
5.              # port du serveur SMTP
6.              "smtp-port": "25",
7.              # administrateur
8.              "from": "guest@localhost.com",
9.              "to": "guest@localhost.com",
10.             # sujet du mail
11.             "subject": "plantage du serveur de calcul d'impôts",
12.             # tls à True si le serveur SMTP requiert une autorisation, à False sinon
13.             "tls": False
14.         }
```

La classe **[SendAminMail]** reçoit le dictionnaire des lignes 2-13 ainsi que la configuration de l'envoi du mail. La classe est la suivante :

```
1.  # imports
2.  import smtplib
3.  from email.mime.text import MIMEText
4.  from email.utils import formatdate
5.
6.
7.  class SendAdminMail:
8.
9.      # ------------------------------------------------------------------------
10.     @staticmethod
11.     def send(config: dict, message: str, verbose: bool = False):
12.         # envoie message au serveur smtp config['smtp-server'] sur le port config[smtp-port]
13.         # si config['tls'] est vrai, le support TLS sera utilisé
14.         # le mail est envoyé de la part de config['from']
15.         # pour le destinataire config['to']
16.         # le message a le sujet config['subject']
17.         # on trouve la référence d'un logueur dans config['logger']
18.
19.         # on récupère le logueur dans la config - peut être égal à None
20.         logger = config["logger"]
21.         # serveur SMTP
22.         server = None
23.         # on envoie le message
24.         try:
25.             # le serveur SMTP
26.             server = smtplib.SMTP(config["smtp-server"])
27.             # mode verbose
28.             server.set_debuglevel(verbose)
29.             # connexion sécurisée ?
30.             if config['tls']:
31.                 # début dialogue de sécurisation
32.                 server.starttls()
33.                 # authentification
34.                 server.login(config["user"], config["password"])
35.             # construction d'un message Multipart - c'est ce message qui sera envoyé
36.             msg = MIMEText(message)
37.             msg['From'] = config["from"]
38.             msg['To'] = config["to"]
39.             msg['Date'] = formatdate(localtime=True)
40.             msg['Subject'] = config["subject"]
41.             # on envoie le message
42.             server.send_message(msg)
43.             # log - le logger peut ne pas exister
44.             if logger:
45.                 logger.write(f"[SendAdminMail] Message envoyé à [{config['to']}] : [{message}]\n")
46.         except BaseException as erreur:
47.             # log- le logger peutne pas exister
48.             if logger:
49.                 logger.write(
50.                     f"[SendAdminMail] Erreur [{erreur}] lors de l'envoi à [{config['to']}] du message
    [{message}] : \n")
51.         finally:
52.             # on a fini - on libère les ressources mobilisées par la fonction
53.             if server:
54.                 server.quit()
```

- lignes 24-54 : on retrouve le code déjà étudié dans l'exemple |smtp/02| ;
- ligne 20 : on récupère la référence d'un logueur. Celui-ci est utilisé aux lignes 45 et 49 ;

24.3 Le serveur web



24.3.1 Configuration

La configuration du serveur est très semblable à celle du serveur étudié précédemment. Seul le fichier **[config.py]** évolue légèrement :

```
1.  def configure(config: dict) -> dict:
2.      import os
3.
4.      # étape 1 ------
5.
6.      # dossier de ce fichier
7.      script_dir = os.path.dirname(os.path.abspath(__file__))
8.
9.      # chemin racine
10.     root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
11.
12.     # dépendances absolues
13.     absolute_dependencies = [
14.         # dossiers du projet
15.         # BaseEntity, MyException
16.         f"{root_dir}/classes/02/entities",
17.         # InterfaceImpôtsDao, InterfaceImpôtsMétier, InterfaceImpôtsUi
18.         f"{root_dir}/impots/v04/interfaces",
19.         # AbstractImpôtsdao, ImpôtsConsole, ImpôtsMétier
20.         f"{root_dir}/impots/v04/services",
21.         # ImpotsDaoWithAdminDataInDatabase
22.         f"{root_dir}/impots/v05/services",
23.         # AdminData, ImpôtsError, TaxPayer
24.         f"{root_dir}/impots/v04/entities",
25.         # Constantes, tranches
26.         f"{root_dir}/impots/v05/entities",
27.         # IndexController
28.         f"{root_dir}/impots/http-servers/01/controllers",
```

```
29.         # scripts [config_database, config_layers]
30.         script_dir,
31.         # Logger, SendAdminMail
32.         f"{script_dir}/../utilities",
33.     ]
34.     # on fixe le syspath
35.     from myutils import set_syspath
36.     set_syspath(absolute_dependencies)
37.
38.     # étape 2 ------
39.     # configuration de l'application
40.     config.update({
41.         # utilisateurs autorisés à utiliser l'application
42.         "users": [
43.             {
44.                 "login": "admin",
45.                 "password": "admin"
46.             }
47.         ],
48.         # fichier de logs
49.         "logsFilename": f"{script_dir}/../data/logs/logs.txt",
50.         # config serveur SMTP
51.         "adminMail": {
52.             # serveur SMTP
53.             "smtp-server": "localhost",
54.             # port du serveur SMTP
55.             "smtp-port": "25",
56.             # administrateur
57.             "from": "guest@localhost.com",
58.             "to": "guest@localhost.com",
59.             # sujet du mail
60.             "subject": "plantage du serveur de calcul d'impôts",
61.             # tls à True si le serveur SMTP requiert une autorisation, à False sinon
62.             "tls": False
63.         },
64.         # durée pause thread en secondes
65.         "sleep_time": 0
66.     })
67.
68.     # étape 3 ------
69.     # configuration base de données
70.     import config_database
71.     config["database"] = config_database.configure(config)
72.
73.     # étape 4 ------
74.     # instanciation des couches de l'application
75.     import config_layers
76.     config['layers'] = config_layers.configure(config)
77.
78.     # on rend la configuration
79.     return config
```

- lignes 40-66 : on ajoute dans le dictionnaire de configuration du serveur, les éléments concernant le logueur (ligne 49) et celles concernant l'envoi d'un mail d'alerte à l'administrateur de l'application (lignes 51-63) ;
- ligne 65 : pour mieux voir les threads en action on va imposer à certains de s'arrêter. **[sleep_time]** est la durée de l'arrêt exprimée en secondes ;
- lignes 27-28 : on notera qu'on utilise le contrôleur **[index_controller]** de la version 6 précédente ;

24.3.2 Le script principal [main]

Le script principal **[main]** est le suivant :

```
1.  # on attend un paramètre mysql ou pgres
2.  import sys
3.  syntaxe = f"{sys.argv[0]} mysql / pgres"
4.  erreur = len(sys.argv) != 2
5.  if not erreur:
6.      sgbd = sys.argv[1].lower()
7.      erreur = sgbd != "mysql" and sgbd != "pgres"
8.  if erreur:
9.      print(f"syntaxe : {syntaxe}")
10.     sys.exit()
11.
12. # on configure l'application
13. import config
```

```
14. config = config.configure({'sgbd': sgbd})
15.
16. # dépendances
17. from flask import request, Flask
18. from flask_httpauth import HTTPBasicAuth
19. import json
20. import index_controller
21. from flask_api import status
22. from SendAdminMail import SendAdminMail
23. from myutils import json_response
24. from Logger import Logger
25. import threading
26. import time
27. from random import randint
28. from ImpôtsError import ImpôtsError
29.
30. # gestionnaire d'authentification
31. auth = HTTPBasicAuth()
32.
33.
34. @auth.verify_password
35. def verify_password(login, password):
36.     # liste des utilisateurs
37.     users = config['users']
38.     # on parcourt cette liste
39.     for user in users:
40.         if user['login'] == login and user['password'] == password:
41.             return True
42.     # on n'a pas trouvé
43.     return False
44.
45.
46. # envoi d'un mail à l'administrateur
47. def send_adminmail(config: dict, message: str):
48.     # on envoie un mail à l'administrateur de l'application
49.     config_mail = config["adminMail"]
50.     config_mail["logger"] = config['logger']
51.     SendAdminMail.send(config_mail, message)
52.
53.
54. # vérification du fichier de logs
55. logger = None
56. erreur = False
57. message_erreur = None
58. try:
59.     # logueur
60.     logger = Logger(config["logsFilename"])
61. except BaseException as exception:
62.     # log console
63.     print(f"L'erreur suivante s'est produite : {exception}")
64.     # on note l'erreur
65.     erreur = True
66.     message_erreur = f"{exception}"
67. # on mémorise le logueur dans la config
68. config['logger'] = logger
69. # gestion de l'erreur
70. if erreur:
71.     # mail à l'administrateur
72.     send_adminmail(config, message_erreur)
73.     # fin de l'application
74.     sys.exit(1)
75.
76. # log de démarrage
77. log = "[serveur] démarrage du serveur"
78. logger.write(f"{log}\n")
79. print(log)
80.
81. # récupération des données de l'administration fiscale
82. erreur = False
83. try:
84.     # admindata sera une donnée de portée application en lecture seule
85.     config["admindata"] = config["layers"]["dao"].get_admindata()
86.     # log de réussite
87.     logger.write("[serveur] connexion à la base de données réussie\n")
88. except ImpôtsError as ex:
89.     # on note l'erreur
90.     erreur = True
```

```
91.     # log d'erreur
92.     log = f"L'erreur suivante s'est produite : {ex}"
93.     # console
94.     print(log)
95.     # fichier de logs
96.     logger.write(f"{log}\n")
97.     # mail à l'administrateur
98.     send_adminmail(config, log)
99.
100.# le thread principal n'a plus besoin du logger
101.logger.close()
102.
103.# s'il y a eu erreur on s'arrête
104.if erreur:
105.    sys.exit(2)
106.
107.# l'application Flask peut démarrer
108.app = Flask(__name__)
109.
110.
111.# Home URL
112.@app.route('/', methods=['GET'])
113.@auth.login_required
114.def index():
115.    …
116.
117.
118.# main uniquement
119.if __name__ == '__main__':
120.    # on lance le serveur
121.    app.config.update(ENV="development", DEBUG=True)
122.    app.run(threaded=True)
```

- lignes 1-10 : le script attend un paramètre **[mysql / pgres]** qui lui indique le SGBD à utiliser ;
- lignes 12-14 : l'application est configurée (Python Path, couches, base de données) ;
- lignes 16-28 : les dépendances nécessaires à l'application ;
- lignes 30-43 : gestion de l'authentification ;
- lignes 46-51 : une fonction qui envoie un mail à l'administrateur de l'application ;
  - la fonction attend deux paramètres :
    - config : un dictionnaire ayant les clés **[adminMail]** et **[logger]** ;
    - le message à envoyer ;
  - lignes 49-50 : on prépare la configuration de l'envoi ;
  - on envoie le mail ;
- lignes 54-74 : on vérifie la présence du fichier de logs ;
- ligne 70-74 : si on n'a pas réussi à ouvrir le fichier de logs, on envoie un mail à l'administrateur et on s'arrête ;
- lignes 76-79 : on logue le démarrage du serveur ;
- lignes 81-98 : on va chercher les données de l'administration fiscale en base de données ;
- lignes 88-98 : si on n'a pas réussi à obtenir ces données, on logue l'erreur aussi bien sur la console que dans le fichier de logs ;
- lignes 100-101 : le thread principal ne fera plus de logs (les threads créés n'utiliseront pas le même descripteur de fichier) ;
- lignes 103-105 : si on n'a pas pu se connecter à la base de données, on s'arrête ;
- ligne 122 : on lance le serveur en mode multithreadé ;

La fonction **[index]** (ligne 114) est la suivante :

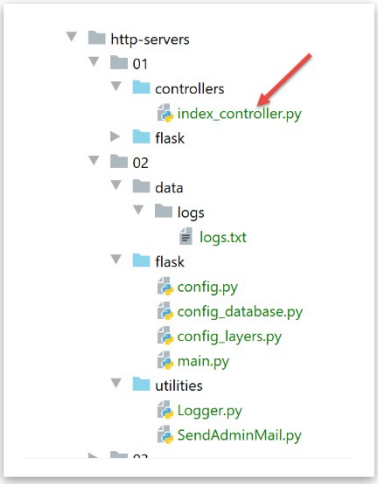
```
1.  # Home URL
2.  @app.route('/', methods=['GET'])
3.  @auth.login_required
4.  def index():
5.      logger = None
6.      try:
7.          # logger
8.          logger = Logger(config["logsFilename"])
9.          # on le mémorise dans une config associée au thread
10.         thread_config = {"logger": logger}
11.         thread_name = threading.current_thread().name
12.         config[thread_name] = {"config": thread_config}
13.         # on logue la requête
14.         logger.write(f"[index] requête : {request}\n")
15.         # on interrompt le thread si cela a été demandé
16.         sleep_time = config["sleep_time"]
17.         if sleep_time != 0:
18.             # la pause est aléatoire pour que certains threads soient interrompus et d'autres pas
```

```
19.             aléa = randint(0, 1)
20.             if aléa == 1:
21.                 # log avant pause
22.                 logger.write(f"[index] mis en pause du thread pendant {sleep_time} seconde(s)\n")
23.                 # pause
24.                 time.sleep(sleep_time)
25.         # on fait exécuter la requête par un contrôleur
26.         résultat, status_code = index_controller.execute(request, config)
27.         # y-a-t-il eu une erreur fatale ?
28.         if status_code == status.HTTP_500_INTERNAL_SERVER_ERROR:
29.             # on envoie un mail à l'administrateur de l'application
30.             config_mail = config["adminMail"]
31.             config_mail["logger"] = logger
32.             SendAdminMail.send(config_mail, json.dumps(résultat, ensure_ascii=False))
33.         # on logue la réponse
34.         logger.write(f"[index] {résultat}\n")
35.         # on envoie la réponse
36.         return json_response(résultat, status_code)
37.     except BaseException as erreur:
38.         # on logue l'erreur si c'est possible
39.         if logger:
40.             logger.write(f"[index] {erreur}")
41.         # on prépare la réponse au client
42.         résultat = {"réponse": {"erreurs": [f"{erreur}"]}}
43.         # on envoie la réponse
44.         return json_response(résultat, status.HTTP_500_INTERNAL_SERVER_ERROR)
45.     finally:
46.         # on ferme le fichier de logs s'il a été ouvert
47.         if logger:
48.             logger.close()
```

- ligne 4 : la fonction exécutée lorsqu'un utilisateur demande l'URL /. Parce que le serveur est multi-threadé (ligne 112), un thread va être créé pour exécuter la fonction. Ce thread peut à tout moment être interrompu et mis en pause pour reprendre son exécution un peu plus tard. Il faut toujours se souvenir de ce point lorsque le code accède à une ressource partagée par tous les threads. Une telle ressource est ici le fichier de logs : tous les threads écrivent dedans ;
- ligne 8 : on crée une instance de logueur. Donc tous les threads auront une instance différence du logueur. Néanmoins tous ces logueurs pointent sur le même fichier de logs. Il est quand même important de noter que lorsqu'un thread ferme son logueur, cela n'a pas d'incidence sur les logueurs des autres threads ;
- lignes 9-12 : on mémorise le logueur dans le dictionnaire **[config]** de l'application associé à une clé portant le nom du thread. Ainsi s'il y a n threads exécutés simultanément, on aura la création de n entrées dans le dictionnaire **[config]**. **[config]** est une ressource partagée entre tous les threads. Il peut donc y avoir un besoin de synchronisation. J'ai fait ici une hypothèse. J'ai supposé que si deux threads créaient simultanément leur entrée dans le fichier **[config]** et que l'un d'eux était interrompu par l'autre, cela n'avait pas d'incidence. Celui interrompu pouvait ultérieurement terminer la création de l'entrée. Si l'expérience montrait que cette hypothèse était fausse, il faudrait synchroniser l'accès à la ligne 12 ;
- ligne 10 : on met le logueur dans un dictionnaire ;
- ligne 11 : **[threading.current_thread()]** est le thread qui exécute cette ligne, donc le thread qui exécute la fonction **[index]**. On note son nom. Chaque thread a un nom unique ;
- ligne 12 : on mémorise la configuration du thread. A partir de maintenant, nous procèderons toujours ainsi : s'il y a des informations qui ne peuvent être partagées entre les threads, elles seront mises quand même dans la configuration générale, mais associées au nom du thread ;
- ligne 14 : on logue la requête qu'on est en train d'exécuter ;
- lignes 15-24 : de façon aléatoire on met certains threads en pause afin qu'ils laissent le processeur à un autre thread ;
  - ligne 16 : on récupère la durée de la pause (en secondes) dans la configuration ;
  - ligne 17 : il n'y a pause que si la durée de pause est différente de 0 ;
  - ligne 19 : un nombre entier aléatoire dans l'intervalle **[0, 1]**. Donc seules les valeurs 0 et 1 sont possibles ;
  - ligne 20 : la pause du thread ne se fait que si le nombre aléatoire est 1 ;
  - ligne 22 : on logue le fait que le thread va être interrompu ;
  - ligne 24 : on interrompt le thread pendant **[sleep_time]** secondes ;
- ligne 26 : lorsque le thread se réveille, il fait exécuter la requête par le module **[index_controller]** ;
- lignes 28-32 : si cette exécution provoque une erreur de type **[500 INTERNAL SERVER ERROR]**, on envoie un mail à l'administrateur ;
  - lignes 30-31 : on configure le dictionnaire **[config_mail]** qu'on va passer à la classe **[SendAdminMail]** ;
  - ligne 32 : le message envoyé à l'administrateur est la chaîne jSON du résultat qui va être envoyé au client ;
- lignes 33-34 : on logue la réponse qu'on va envoyer au client (ligne 36) ;
- lignes 37-44 : traitement d'une éventuelle exception ;
- lignes 39-40 : si le logueur existe on logue l'erreur qui s'est produite ;
- lignes 47-48 : on ferme le logueur s'il existe. Au final, le thread crée un logueur au début de la requête et le ferme lorsque celle-ci a été traitée ;

24.3.3 Le contrôleur [index_controller]

Le contrôleur **[index_controller]** qui exécute les requêtes est celui de la version précédente :



24.3.4 Exécution

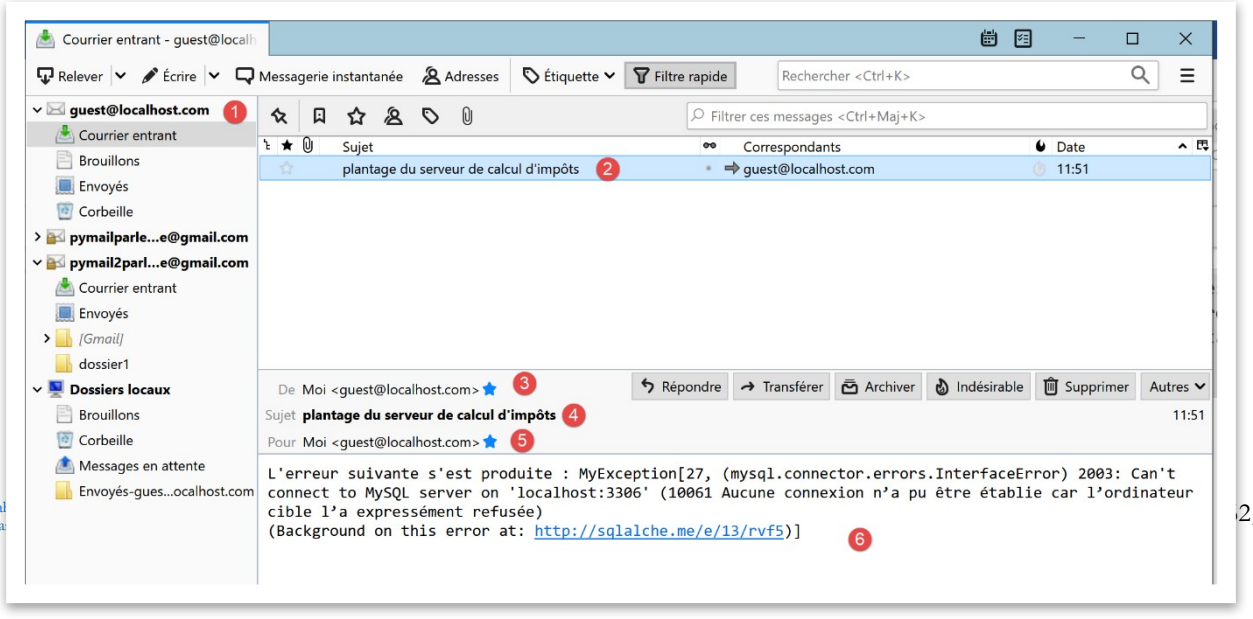
On lance le serveur Flask, le serveur de mails |hMailServer| ainsi que le lecteur de courrier |Thunderbird|. On ne lance pas le SGBD. Le serveur s'arrête avec les logs console suivants :

```
1. C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
   C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/http-servers/02/flask/main.py mysql
2. [serveur] démarrage du serveur
3. L'erreur suivante s'est produite : MyException[27, (mysql.connector.errors.InterfaceError) 2003: Can't
   connect to MySQL server on 'localhost:3306' (10061 Aucune connexion n'a pu être établie car l'ordinateur
   cible l'a expressément refusée)
4. (Background on this error at: http://sqlalche.me/e/13/rvf5)]
5.
6. Process finished with exit code 2
```

Le fichier de logs **[logs.txt]** est lui le suivant :

```
1. 2020-07-23 11:51:38.324752, MainThread : [serveur] démarrage du serveur
2. 2020-07-23 11:51:40.355510, MainThread : L'erreur suivante s'est produite : MyException[27,
   (mysql.connector.errors.InterfaceError) 2003: Can't connect to MySQL server on 'localhost:3306' (10061
   Aucune connexion n'a pu être établie car l'ordinateur cible l'a expressément refusée)
3. (Background on this error at: http://sqlalche.me/e/13/rvf5)]
4. 2020-07-23 11:51:42.464206, MainThread : [SendAdminMail] Message envoyé à [guest@localhost.com] : [L'erreur
   suivante s'est produite : MyException[27, (mysql.connector.errors.InterfaceError) 2003: Can't connect to
   MySQL server on 'localhost:3306' (10061 Aucune connexion n'a pu être établie car l'ordinateur cible l'a
   expressément refusée)
5. (Background on this error at: http://sqlalche.me/e/13/rvf5)]]
```

Avec Thunderbird, on vérifie les mails de l'administrateur **[guest@localhost.com]** :

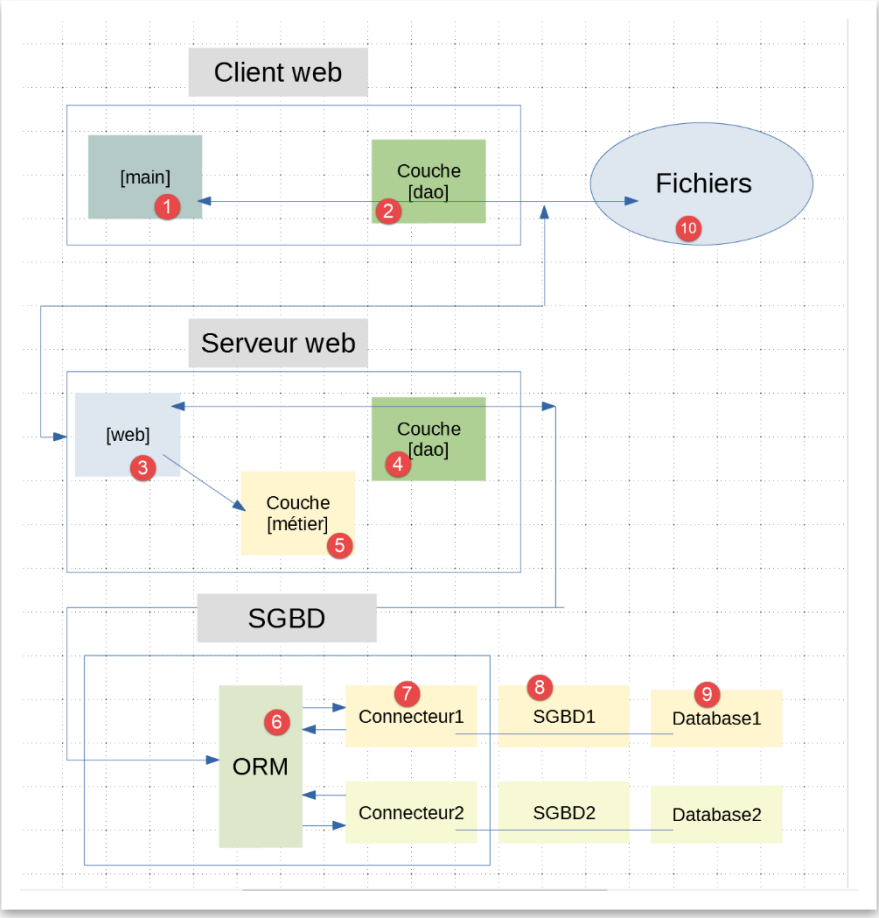


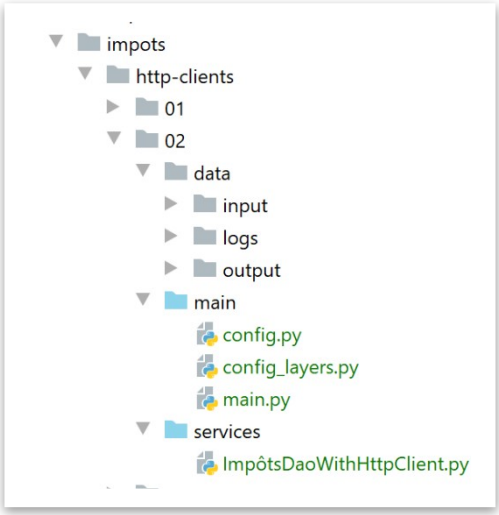
Puis on lance le SGBD et on demande l'URL [http://127.0.0.1:5000/?mari%C3%A9=oui&enfants=3&salaire=200000]. Les logs deviennent les suivants :

```
1.  2020-07-23 11:56:38.891753, MainThread : [serveur] démarrage du serveur
2.  2020-07-23 11:56:38.987999, MainThread : [serveur] connexion à la base de données réussie
3.  2020-07-23 11:56:40.586747, MainThread : [serveur] démarrage du serveur
4.  2020-07-23 11:56:40.655254, MainThread : [serveur] connexion à la base de données réussie
5.  2020-07-23 11:56:54.528360, Thread-2 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=3&salaire=200000' [GET]>
6.  2020-07-23 11:56:54.530653, Thread-2 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 3,
    'salaire': 200000, 'impôt': 42842, 'surcôte': 17283, 'taux': 0.41, 'décôte': 0, 'réduction': 0}}}
```

- lignes 1-4 : on rappelle qu'il y a deux démarrages du serveur parce que le mode **[Debug=True]** provoque un second démarrage ;
- lignes 5-6 : les logs nous donnent une idée du temps d'exécution d'une requête, ici 2,293 millisecondes ;

24.4 Le client web





Le dossier **[http-clients/02]** est obtenu par recopie du dossier **[http-clients/01]**. On procède ensuite à quelques modifications.

24.4.1 La configuration

La configuration **[config]** de l'application **[http-clients/02]** est le même que celle de l'application **[http-clients/01]** à quelques détails près :

```
1.  def configure(config: dict) -> dict:
2.      import os
3.
4.      # étape 1 ------
5.
6.      # dossier de ce fichier
7.      script_dir = os.path.dirname(os.path.abspath(__file__))
8.
9.      # chemin racine
10.     root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
11.
12.     # dépendances absolues
13.     absolute_dependencies = [
14.         # dossiers du projet
15.         # BaseEntity, MyException
16.         f"{root_dir}/classes/02/entities",
17.         # InterfaceImpôtsDao, InterfaceImpôtsMétier, InterfaceImpôtsUi
18.         f"{root_dir}/impots/v04/interfaces",
19.         # AbstractImpôtsdao, ImpôtsConsole, ImpôtsMétier
20.         f"{root_dir}/impots/v04/services",
21.         # ImpotsDaoWithAdminDataInDatabase
22.         f"{root_dir}/impots/v05/services",
23.         # AdminData, ImpôtsError, TaxPayer
24.         f"{root_dir}/impots/v04/entities",
25.         # Constantes, tranches
26.         f"{root_dir}/impots/v05/entities",
27.         # ImpôtsDaoWithHttpClient
28.         f"{script_dir}/../services",
29.         # scripts de configuration
30.         script_dir,
31.         # Logger
32.         f"{root_dir}/impots/http-servers/02/utilities",
33.     ]
34.
35.     # on fixe le syspath
36.     from myutils import set_syspath
37.     set_syspath(absolute_dependencies)
38.
39.     # étape 2 ------
40.     # configuration de l'application avec des constantes
41.     config.update({
42.         # fichier des contribuables
43.         "taxpayersFilename": f"{script_dir}/../data/input/taxpayersdata.txt",
44.         # fichier des résultats
45.         "resultsFilename": f"{script_dir}/../data/output/résultats.json",
46.         # fichier des erreurs
```

```
47.         "errorsFilename": f"{script_dir}/../data/output/errors.txt",
48.         # fichier de logs
49.         "logsFilename": f"{script_dir}/../data/logs/logs.txt",
50.         # le serveur de calcul de l'impôt
51.         "server": {
52.             "urlServer": "http://127.0.0.1:5000/",
53.             "authBasic": True,
54.             "user": {
55.                 "login": "admin",
56.                 "password": "admin"
57.             }
58.         },
59.         # mode debug
60.         "debug": True
61.     }
62.     )
63.
64.     # étape 3 ------
65.     # instanciation des couches
66.     import config_layers
67.     config['layers'] = config_layers.configure(config)
68.
69.     # on rend la configuation
70.     return config
```

- lignes 31-32 : on va utiliser le même logueur |Logger| que celui utilisé pour le serveur ;
- ligne 49 : le chemin absolu du fichier de logs ;
- ligne 60 : le mode **[debug=True]** sert à écrire les réponses du serveur web dans le fichier de logs ;

24.4.2 La couche [dao]

Le code de la classe **[ImpôtsDaoWithHttpClient]** évolue légèrement :

```
1.  # imports
2.
3.  import requests
4.  from flask_api import status
5.
6.  …
7.
8.
9.  class ImpôtsDaoWithHttpClient(AbstractImpôtsDao, InterfaceImpôtsMétier):
10.
11.     # constructeur
12.     def __init__(self, config: dict):
13.         # initialisation parent
14.         AbstractImpôtsDao.__init__(self, config)
15.         # mémorisation éléments de la configuration
16.         # config générale
17.         self.__config = config
18.         # serveur
19.         self.__config_server = config["server"]
20.         # mode debug
21.         self.__debug = config["debug"]
22.         # logger
23.         self.__logger = None
24.
25.     # méthode inutilisée
26.     def get_admindata(self) -> AdminData:
27.         pass
28.
29.     # calcul de l'impôt
30.     def calculate_tax(self, taxpayer: TaxPayer, admindata: AdminData = None):
31.         # on laisse remonter les exceptions
32.         …
33.         # mode debug ?
34.         if self.__debug:
35.             # logueur
36.             if not self.__logger:
37.                 self.__logger = self.__config['logger']
38.             # on logue
39.             self.__logger.write(f"{response.text}\n")
40.         # code de statut de la réponse HTTP
41.         status_code = response.status_code
42.         …
```

- lignes 17 : on mémorise la configuration générale. On verra ultérieurement que lorsque le constructeur de la classe **[ImpôtsDaoWithHttpClient]** s'exécute, le dictionnaire **[config]** ne contient pas encore la clé **[logger]** utilisée ligne 37. C'est pour cette raison qu'on ne peut pas initialiser **[self.__logger]** (ligne 23) dans le constructeur ;
- ligne 21 : on a ajouté dans la configuration une clé **[debug]** qui contrôle le log des lignes 33-39 ;
- ligne 34 : si on est en mode **[debug]** ;
- lignes 36-37 : initialisation éventuelle de la propriété **[self.__logger]**. Lorsque la méthode **[calculate_tax]** est utilisée, la clé **[logger]** fait partie du dictionnaire **[config]** ;
- ligne 39 : on logue le document texte associé à la réponse HTTP du serveur ;

La couche **[dao]** va être exécutée simultanément par plusieurs threads. Or ici on crée un unique exemplaire de cette couche (cf. config_layers). Il faut donc vérifier que le code n'implique pas l'accès en écriture à des données partagées, typiquement les propriétés de la classe **[ImpôtsDaoWithHttpClient]** qui implémente la couche **[dao]**. Or ci-dessus la ligne 37 modifie une propriété de l'instance de classe. Ici ça ne porte pas à conséquence car tous les threads partagent le même logueur. Si cela n'avait pas été le cas, l'accès à la ligne 37 aurait du être synchronisé.

24.4.3 Le script principal

Le script principal **[main]** évolue de la façon suivante :

```
1.  # on configure l'application
2.
3.  import config
4.  config = config.configure({})
5.
6.  # dépendances
7.  from ImpôtsError import ImpôtsError
8.  import random
9.  import sys
10. import threading
11. from Logger import Logger
12.
13.
14. # exécution de la couche [dao] dans un thread
15. # taxpayers est une liste de contribuables
16. def thread_function(dao, logger, taxpayers: list):
17.     …
18.
19.
20. # liste des threads du client
21. threads = []
22. logger = None
23. # code
24. try:
25.     # logger
26.     logger = Logger(config["logsFilename"])
27.     # on le mémorise dans la config
28.     config["logger"] = logger
29.     # on récupère la couche [dao]
30.     dao = config["layers"]["dao"]
31.     # lecture des données des contribuables
32.     taxpayers = dao.get_taxpayers_data()["taxpayers"]
33.     # des contribuables ?
34.     if not taxpayers:
35.         raise ImpôtsError(36, f"Pas de contribuables valides dans le fichier {config['taxpayersFilename']}")
36.     # calcul de l'impôt des contribuables avec plusieurs threads
37.     i = 0
38.     l_taxpayers = len(taxpayers)
39.     while i < len(taxpayers):
40.         # chaque thread va traiter de 1 à 4 contribuables
41.         nb_taxpayers = min(l_taxpayers - i, random.randint(1, 4))
42.         # la liste des contribuables traités par le thread
43.         thread_taxpayers = taxpayers[slice(i, i + nb_taxpayers)]
44.         # on incrémente i pour le thread suivant
45.         i += nb_taxpayers
46.         # on crée le thread
47.         thread = threading.Thread(target=thread_function, args=(dao, logger, thread_taxpayers))
48.         # on l'ajoute à la liste des threads du script principal
49.         threads.append(thread)
50.         # on lance le thread - cette opération est asynchrone - on n'attend pas le résultat du thread
51.         thread.start()
52.     # le thread principal attend la fin de tous les threads qu'il a lancés
53.     for thread in threads:
54.         thread.join()
55.     # ici tous les threads ont fini leur travail - chacun a modifié un ou plusieurs objets [taxpayer]
56.     # on enregistre les résultats dans le fichier jSON
57.     dao.write_taxpayers_results(taxpayers)
58. except BaseException as erreur:
59.     # affichage de l'erreur
60.     print(f"L'erreur suivante s'est produite : {erreur}")
61. finally:
```

```
62.      # on ferme le logueur
63.      if logger:
64.          logger.close()
65.      # on a fini
66.      print("Travail terminé...")
67.      # fin des threads qui pourraient encore exister si on s'est arrêté sur erreur
68.      sys.exit()
```

- le script principal se distingue de celui du client précédent par le fait qu'il va générer plusieurs threads d'exécution pour effectuer les requêtes au serveur. Le client de la version 6 faisait toutes ses requêtes séquentiellement. La requête n° i n'était faite qu'une fois la réponse à la requête n° **[i-1]** reçue. Ici on veut voir comment le serveur va se comporter lorsqu'il reçoit plusieurs requêtes simultanées. Pour cela nous avons besoin des threads ;
- ligne 21 : les threads générés vont être mis dans une liste. Il faut comprendre que le script **[main]** est lui aussi exécuté par un thread qui s'appelle **[MainThread]**. Ce thread principal va créer d'autres threads qui seront chargés de calculer l'impôt d'un ou plusieurs contribuables ;
- ligne 26 : on crée un logueur. Celui-ci sera partagé par tous les threads ;
- ligne 32 : on récupère tous les contribuables dont il faut calculer l'impôt ;
- lignes 39-51 : on va répartir ces contribuables sur plusieurs threads ;
- lignes 40-41 : chaque thread va traiter de 1 à 4 contribuables. Ce nombre est fixé de façon aléatoire ;
  - **[random.randint(1, 4)]** donne aléatoirement un nombre dans la liste **[1, 2, 3, 4]** ;
  - le thread ne peut avoir plus de **[l-i]** contribuables où **[l-i]** représente le nombre de contribuables à qui on n'a pas encore attribué de thread ;
  - on prend donc le min des deux valeurs ;
- ligne 43 : une fois que **[nb_taxpayers]**, le nombre de contribuables traités par le thread, est connu, on prend ceux-ci dans la liste des contribuables :
  - **[slice(10,12)]** est l'ensemble des indices **[10, 11, 12]** ;
  - **[taxpayers[slice(10,12)]**] est la liste **[taxpayers[10]**, taxpayers**[11]**, taxpayers**[12]** ;
- ligne 45 : on incrémente la valeur de i qui contrôle la boucle de la ligne 39 ;
- ligne 47 : on crée un thread :
  - **[target=thread_function]** fixe la fonction qu'exécutera le thread. C'est la fonction des lignes 16-17. Elle attend trois paramètres ;
  - **[ags]** est la liste des trois paramètres attendus par la fonction **[thread_function]** ;

  Créer un thread ne l'exécute pas. Ca crée un objet et c'est tout ;
- lignes 48-49 : le thread qui vient d'être créé est ajouté à la liste des threads créés par le thread principal ;
- ligne 51 : le thread est lancé. Il va alors être exécuté en parallèle des autres threads actifs. Ici, il va exécuter la fonction **[thread_function]** avec les arguments qu'on lui a donnés ;
- lignes 53-54 : le thread principal attend chacun des threads qu'il a lancés. Prenons un exemple :
  - le thread principal a lancé trois threads **[th1, th2, th3]** ;
  - le thread principal se met en attente de chacun des threads (lignes 53-54) dans l'ordre de la boucle for : **[th1, th2, th3]** ;
  - supposons que les threads se terminent dans l'ordre **[th2, th1, th3]** ;
  - le thread principal attend la fin de th1. Lorsque th2 se termine, il ne se passe rien ;
  - lorsque th1 se termine, le thread principal se met en attente de th2. Or celui-ci est terminé. Le thread principal passe alors au thread suivant et attend th3 ;
  - lorsque th3 se termine, le thread principal a terminé son attente et passe alors à l'exécution de la ligne 57 ;
- la ligne 57 écrit les résultats obtenus dans le fichier des résultats. On a là un bon exemple des références d'objets :
  - ligne 43 : la liste **[thread_payers]** associée à un thread contient des copies des références d'objets contenus dans la liste **[taxpayers]** ;
  - on sait que le calcul de l'impôt va modifier les objets pointés par les références de la liste **[thread_payers]**. Ces objets vont se voir enrichis des résultats du calcul de l'impôt. Néanmoins les références elles ne sont pas modifiées. Donc les références de la liste initiale **[taxpayers]** 'voient' ou 'pointent sur' les objets modifiés ;

La fonction **[thread_function]** exécutée par les threads est la suivante :

```
1.  # exécution de la couche [dao] dans un thread
2.  # taxpayers est une liste de contribuables
3.  def thread_function(dao, logger, taxpayers: list):
4.      # log début du thread
5.      thread_name = threading.current_thread().name
6.      logger.write(f"début du thread [{thread_name}] avec {len(taxpayers)} contribuable(s)\n")
7.      # on calcule l'impôt des contribuables
8.      for taxpayer in taxpayers:
9.          # log
10.         logger.write(f"début du calcul de l'impôt de {taxpayer}\n")
11.         # calcul synchrone de l'impôt
12.         dao.calculate_tax(taxpayer)
13.         # log
14.         logger.write(f"fin du calcul de l'impôt de {taxpayer}\n")
15.     # log fin du thread
```

```
16.     logger.write(f"fin du thread [{thread_name}]\n")
```

- les fonctions exécutées simultanément par plusieurs threads sont souvent délicates à écrire : on doit toujours vérifier que le code n'essaie pas de modifier une donnée partagée entre threads. Lorsque ce dernier cas se produit, il faut mettre en place un accès synchronisé à la donnée partagée qui va être modifiée ;
- ligne 3 : la fonction reçoit trois paramètres :
  - **[dao]** : une référence sur la couche **[dao]**. Cette donnée est partagée ;
  - **[logger]** : une référence sur le logueur. Cette donnée est partagée ;
  - **[taxpayers]** : une liste de contribuables. Cette donnée n'est pas partagée : chaque thread gère une liste différente ;
- examinons les deux références **[dao, logger]** :
  - on a vu que l'objet pointé par la référence **[dao]** avait une référence **[self.__logger]** qui était modifiée par les threads mais pour y mettre une valeur commune à tous les threads ;
  - la référence **[logger]** pointe sur un descripteur de fichier. On a vu qu'il pouvait y avoir un problème lors de l'écriture des logs dans le fichier. Pour cette raison l'écriture dans le fichier a été synchronisée ;
- lignes 5-6 : on logue le nom du thread et le nombre de contribuables qu'il doit gérer ;
- lignes 8-14 : calcul de l'impôt des contribuables ;
- ligne 16 : on logue la fin du thread ;

24.4.4 Exécution

Lançons le serveur web comme dans le paragraphe précédent (serveur web, SGBD, hMailServer, Thunderbird), puis exécutons le script **[main]** du client. Dans les fichiers **[data/output/errors.txt, data/output/résultats.json]** on a les mêmes résultats que dans la version précédente. Dans le fichier **[data/logs/logs.txt]**, on a les logs suivants :

```
1.  2020-07-24 10:05:20.942404, Thread-1 : début du thread [Thread-1] avec 1 contribuable(s)
2.  2020-07-24 10:05:20.943458, Thread-1 : début du calcul de l'impôt de {"id": 1, "marié": "oui", "enfants":
    2, "salaire": 55555}
3.  2020-07-24 10:05:20.943458, Thread-2 : début du thread [Thread-2] avec 3 contribuable(s)
4.  2020-07-24 10:05:20.946502, Thread-3 : début du thread [Thread-3] avec 1 contribuable(s)
5.  2020-07-24 10:05:20.946502, Thread-2 : début du calcul de l'impôt de {"id": 2, "marié": "oui", "enfants":
    2, "salaire": 50000}
6.  2020-07-24 10:05:20.947003, Thread-3 : début du calcul de l'impôt de {"id": 5, "marié": "non", "enfants":
    3, "salaire": 100000}
7.  2020-07-24 10:05:20.947003, Thread-4 : début du thread [Thread-4] avec 3 contribuable(s)
8.  2020-07-24 10:05:20.950324, Thread-4 : début du calcul de l'impôt de {"id": 6, "marié": "oui", "enfants":
    3, "salaire": 100000}
9.  2020-07-24 10:05:20.948449, Thread-5 : début du thread [Thread-5] avec 3 contribuable(s)
10. 2020-07-24 10:05:20.953645, Thread-5 : début du calcul de l'impôt de {"id": 9, "marié": "oui", "enfants":
    2, "salaire": 30000}
11. 2020-07-24 10:05:20.976143, Thread-1 : {"réponse": {"result": {"marié": "oui", "enfants": 2, "salaire":
    55555, "impôt": 2814, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0}}}
12. 2020-07-24 10:05:20.976695, Thread-1 : fin du calcul de l'impôt de {"id": 1, "marié": "oui", "enfants": 2,
    "salaire": 55555, "impôt": 2814, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0}
13. 2020-07-24 10:05:20.976695, Thread-1 : fin du thread [Thread-1]
14. 2020-07-24 10:05:21.973914, Thread-2 : {"réponse": {"result": {"marié": "oui", "enfants": 2, "salaire":
    50000, "impôt": 1384, "surcôte": 0, "taux": 0.14, "décôte": 384, "réduction": 347}}}
15. 2020-07-24 10:05:21.973914, Thread-2 : fin du calcul de l'impôt de {"id": 2, "marié": "oui", "enfants": 2,
    "salaire": 50000, "impôt": 1384, "surcôte": 0, "taux": 0.14, "décôte": 384, "réduction": 347}
16. 2020-07-24 10:05:21.973914, Thread-2 : début du calcul de l'impôt de {"id": 3, "marié": "oui", "enfants":
    3, "salaire": 50000}
17. 2020-07-24 10:05:21.977130, Thread-4 : {"réponse": {"result": {"marié": "oui", "enfants": 3, "salaire":
    100000, "impôt": 9200, "surcôte": 2180, "taux": 0.3, "décôte": 0, "réduction": 0}}}
18. 2020-07-24 10:05:21.977130, Thread-4 : fin du calcul de l'impôt de {"id": 6, "marié": "oui", "enfants": 3,
    "salaire": 100000, "impôt": 9200, "surcôte": 2180, "taux": 0.3, "décôte": 0, "réduction": 0}
19. 2020-07-24 10:05:21.977130, Thread-4 : début du calcul de l'impôt de {"id": 7, "marié": "oui", "enfants":
    5, "salaire": 100000}
20. 2020-07-24 10:05:21.982634, Thread-3 : {"réponse": {"result": {"marié": "non", "enfants": 3, "salaire":
    100000, "impôt": 16782, "surcôte": 7176, "taux": 0.41, "décôte": 0, "réduction": 0}}}
21. 2020-07-24 10:05:21.982634, Thread-5 : {"réponse": {"result": {"marié": "oui", "enfants": 2, "salaire":
    30000, "impôt": 0, "surcôte": 0, "taux": 0.0, "décôte": 0, "réduction": 0}}}
22. 2020-07-24 10:05:21.983134, Thread-3 : fin du calcul de l'impôt de {"id": 5, "marié": "non", "enfants": 3,
    "salaire": 100000, "impôt": 16782, "surcôte": 7176, "taux": 0.41, "décôte": 0, "réduction": 0}
23. 2020-07-24 10:05:21.983134, Thread-5 : fin du calcul de l'impôt de {"id": 9, "marié": "oui", "enfants": 2,
    "salaire": 30000, "impôt": 0, "surcôte": 0, "taux": 0.0, "décôte": 0, "réduction": 0}
24. 2020-07-24 10:05:21.983134, Thread-3 : fin du thread [Thread-3]
25. 2020-07-24 10:05:21.983763, Thread-5 : début du calcul de l'impôt de {"id": 10, "marié": "non", "enfants":
    0, "salaire": 200000}
26. 2020-07-24 10:05:22.008562, Thread-5 : {"réponse": {"result": {"marié": "non", "enfants": 0, "salaire":
    200000, "impôt": 64210, "surcôte": 7498, "taux": 0.45, "décôte": 0, "réduction": 0}}}
27. 2020-07-24 10:05:22.008562, Thread-5 : fin du calcul de l'impôt de {"id": 10, "marié": "non", "enfants": 0,
    "salaire": 200000, "impôt": 64210, "surcôte": 7498, "taux": 0.45, "décôte": 0, "réduction": 0}
28. 2020-07-24 10:05:22.009062, Thread-5 : début du calcul de l'impôt de {"id": 11, "marié": "oui", "enfants":
    3, "salaire": 200000}
```

```
29. 2020-07-24 10:05:22.016848, Thread-5 : {"réponse": {"result": {"marié": "oui", "enfants": 3, "salaire":
    200000, "impôt": 42842, "surcôte": 17283, "taux": 0.41, "décôte": 0, "réduction": 0}}}
30. 2020-07-24 10:05:22.017349, Thread-5 : fin du calcul de l'impôt de {"id": 11, "marié": "oui", "enfants": 3,
    "salaire": 200000, "impôt": 42842, "surcôte": 17283, "taux": 0.41, "décôte": 0, "réduction": 0}
31. 2020-07-24 10:05:22.017349, Thread-5 : fin du thread [Thread-5]
32. 2020-07-24 10:05:23.008486, Thread-2 : {"réponse": {"result": {"marié": "oui", "enfants": 3, "salaire":
    50000, "impôt": 0, "surcôte": 0, "taux": 0.14, "décôte": 720, "réduction": 0}}}
33. 2020-07-24 10:05:23.008486, Thread-2 : fin du calcul de l'impôt de {"id": 3, "marié": "oui", "enfants": 3,
    "salaire": 50000, "impôt": 0, "surcôte": 0, "taux": 0.14, "décôte": 720, "réduction": 0}
34. 2020-07-24 10:05:23.009749, Thread-2 : début du calcul de l'impôt de {"id": 4, "marié": "non", "enfants":
    2, "salaire": 100000}
35. 2020-07-24 10:05:23.011722, Thread-4 : {"réponse": {"result": {"marié": "oui", "enfants": 5, "salaire":
    100000, "impôt": 4230, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0}}}
36. 2020-07-24 10:05:23.013723, Thread-4 : fin du calcul de l'impôt de {"id": 7, "marié": "oui", "enfants": 5,
    "salaire": 100000, "impôt": 4230, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0}
37. 2020-07-24 10:05:23.013723, Thread-4 : début du calcul de l'impôt de {"id": 8, "marié": "non", "enfants":
    0, "salaire": 100000}
38. 2020-07-24 10:05:23.024135, Thread-2 : {"réponse": {"result": {"marié": "non", "enfants": 2, "salaire":
    100000, "impôt": 19884, "surcôte": 4480, "taux": 0.41, "décôte": 0, "réduction": 0}}}
39. 2020-07-24 10:05:23.024135, Thread-2 : fin du calcul de l'impôt de {"id": 4, "marié": "non", "enfants": 2,
    "salaire": 100000, "impôt": 19884, "surcôte": 4480, "taux": 0.41, "décôte": 0, "réduction": 0}
40. 2020-07-24 10:05:23.025178, Thread-2 : fin du thread [Thread-2]
41. 2020-07-24 10:05:23.025178, Thread-4 : {"réponse": {"result": {"marié": "non", "enfants": 0, "salaire":
    100000, "impôt": 22986, "surcôte": 0, "taux": 0.41, "décôte": 0, "réduction": 0}}}
42. 2020-07-24 10:05:23.026191, Thread-4 : fin du calcul de l'impôt de {"id": 8, "marié": "non", "enfants": 0,
    "salaire": 100000, "impôt": 22986, "surcôte": 0, "taux": 0.41, "décôte": 0, "réduction": 0}
43. 2020-07-24 10:05:23.026191, Thread-4 : fin du thread [Thread-4]
```

- ces logs montrent que cinq threads ont été lancés pour calculer l'impôt de 11 contribuables. Ces cinq threads ont lancé des requêtes simultanées au serveur de calcul de l'impôt. Il faut comprendre comment ça marche :
  - le thread **[Thread-1]** est lancé le premier. Lorsqu'il a le processeur, il avance dans le code jusqu'à envoyer sa requête HTTP. Comme il doit attendre le résultat de celle-ci, il est automatiquement mis en attente. Il perd alors le processeur et un autre thread obtient celui-ci ;
  - lignes 1-10 : le même processus se répète pour chacun des 5 threads. Ainsi les 5 threads sont lancés avant même que le thread **[Thread-1]** ait reçu sa réponse ligne 11 ;
- les threads ne se terminent pas dans l'ordre où ils ont été lancés. Ainsi c'est le thread **[Thread-3]** qui termine le premier, ligne 23 ;

Côté serveur, les logs dans le fichier **[data/logs/logs.txt]** sont les suivants :

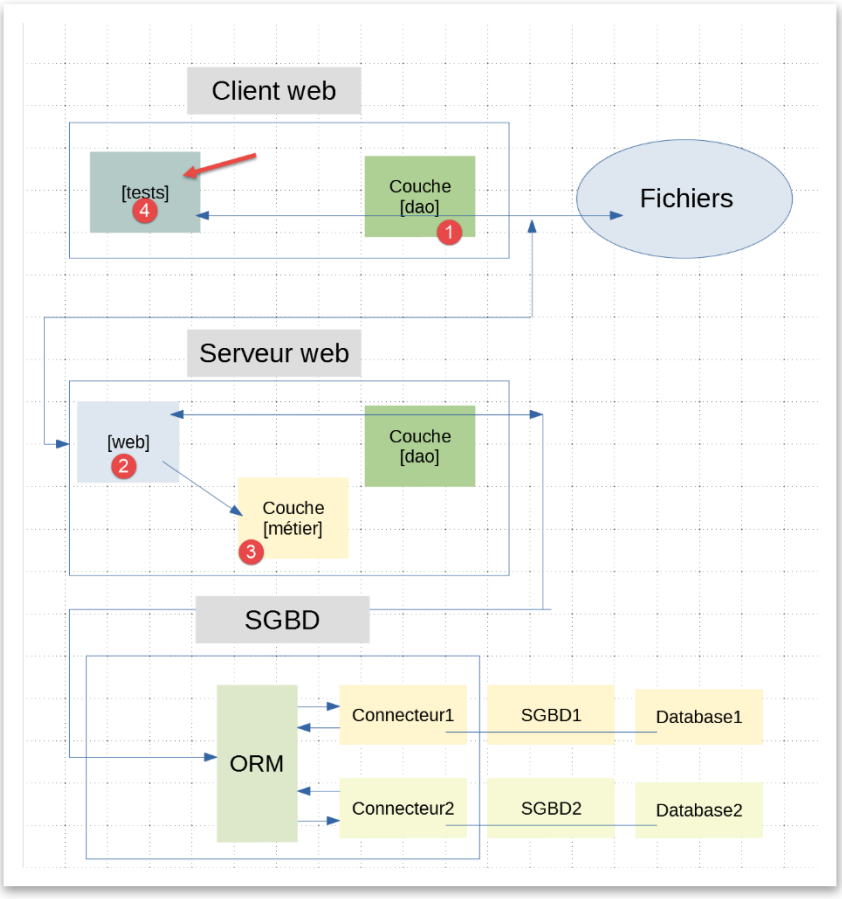
```
1.  2020-07-24 10:05:01.692980, MainThread : [serveur] démarrage du serveur
2.  2020-07-24 10:05:01.877251, MainThread : [serveur] connexion à la base de données réussie
3.  2020-07-24 10:05:03.596162, MainThread : [serveur] démarrage du serveur
4.  2020-07-24 10:05:03.661160, MainThread : [serveur] connexion à la base de données réussie
5.  2020-07-24 10:05:20.968053, Thread-2 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=2&salaire=50000' [GET]>
6.  2020-07-24 10:05:20.969132, Thread-2 : [index] mis en pause du thread pendant 1 seconde(s)
7.  2020-07-24 10:05:20.970316, Thread-3 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=3&salaire=100000' [GET]>
8.  2020-07-24 10:05:20.970316, Thread-3 : [index] mis en pause du thread pendant 1 seconde(s)
9.  2020-07-24 10:05:20.971335, Thread-4 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=2&salaire=55555' [GET]>
10. 2020-07-24 10:05:20.972563, Thread-4 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 2,
    'salaire': 55555, 'impôt': 2814, 'surcôte': 0, 'taux': 0.14, 'décôte': 0, 'réduction': 0}}}
11. 2020-07-24 10:05:20.974796, Thread-5 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=non&enfants=3&salaire=100000' [GET]>
12. 2020-07-24 10:05:20.974796, Thread-5 : [index] mis en pause du thread pendant 1 seconde(s)
13. 2020-07-24 10:05:20.976143, Thread-6 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=2&salaire=30000' [GET]>
14. 2020-07-24 10:05:20.976143, Thread-6 : [index] mis en pause du thread pendant 1 seconde(s)
15. 2020-07-24 10:05:21.970615, Thread-2 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 2,
    'salaire': 50000, 'impôt': 1384, 'surcôte': 0, 'taux': 0.14, 'décôte': 384, 'réduction': 347}}}
16. 2020-07-24 10:05:21.973914, Thread-3 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 3,
    'salaire': 100000, 'impôt': 9200, 'surcôte': 2180, 'taux': 0.3, 'décôte': 0, 'réduction': 0}}}
17. 2020-07-24 10:05:21.977130, Thread-6 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 2,
    'salaire': 30000, 'impôt': 0, 'surcôte': 0, 'taux': 0.0, 'décôte': 0, 'réduction': 0}}}
18. 2020-07-24 10:05:21.977130, Thread-5 : [index] {'réponse': {'result': {'marié': 'non', 'enfants': 3,
    'salaire': 100000, 'impôt': 16782, 'surcôte': 7176, 'taux': 0.41, 'décôte': 0, 'réduction': 0}}}
19. 2020-07-24 10:05:22.001693, Thread-7 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=3&salaire=50000' [GET]>
20. 2020-07-24 10:05:22.003013, Thread-7 : [index] mis en pause du thread pendant 1 seconde(s)
21. 2020-07-24 10:05:22.003013, Thread-8 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=5&salaire=100000' [GET]>
22. 2020-07-24 10:05:22.003013, Thread-8 : [index] mis en pause du thread pendant 1 seconde(s)
```

```
23. 2020-07-24 10:05:22.005871, Thread-9 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=non&enfants=0&salaire=200000' [GET]>
24. 2020-07-24 10:05:22.006370, Thread-9 : [index] {'réponse': {'result': {'marié': 'non', 'enfants': 0,
    'salaire': 200000, 'impôt': 64210, 'surcôte': 7498, 'taux': 0.45, 'décôte': 0, 'réduction': 0}}}
25. 2020-07-24 10:05:22.014170, Thread-10 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=3&salaire=200000' [GET]>
26. 2020-07-24 10:05:22.014170, Thread-10 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 3,
    'salaire': 200000, 'impôt': 42842, 'surcôte': 17283, 'taux': 0.41, 'décôte': 0, 'réduction': 0}}}
27. 2020-07-24 10:05:23.003533, Thread-7 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 3,
    'salaire': 50000, 'impôt': 0, 'surcôte': 0, 'taux': 0.14, 'décôte': 720, 'réduction': 0}}}
28. 2020-07-24 10:05:23.006434, Thread-8 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 5,
    'salaire': 100000, 'impôt': 4230, 'surcôte': 0, 'taux': 0.14, 'décôte': 0, 'réduction': 0}}}
29. 2020-07-24 10:05:23.018026, Thread-11 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=non&enfants=2&salaire=100000' [GET]>
30. 2020-07-24 10:05:23.019074, Thread-11 : [index] {'réponse': {'result': {'marié': 'non', 'enfants': 2,
    'salaire': 100000, 'impôt': 19884, 'surcôte': 4480, 'taux': 0.41, 'décôte': 0, 'réduction': 0}}}
31. 2020-07-24 10:05:23.021447, Thread-12 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=non&enfants=0&salaire=100000' [GET]>
32. 2020-07-24 10:05:23.022447, Thread-12 : [index] {'réponse': {'result': {'marié': 'non', 'enfants': 0,
    'salaire': 100000, 'impôt': 22986, 'surcôte': 0, 'taux': 0.41, 'décôte': 0, 'réduction': 0}}}
```

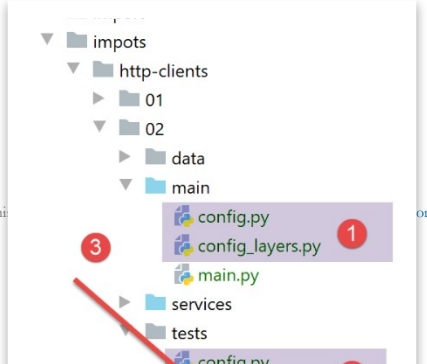
- on voit que 11 threads ont traité les 11 contribuables ;
- certains threads ont été mis en attente (lignes 6, 8, 12, 14, 20, 22) et d'autres pas (lignes 9, 23, 25, 29, 31) ;

24.5 Tests de la couche `[dao]`

Comme nous l'avons fait dans la |version précédente| nous testons la couche **`[dao]`** du client. Le principe est exactement le même :



La classe de test sera exécutée dans l'environnement suivant :



- la configuration **[2]** est identique à la configuration **[1]** que nous venons d'étudier ;

La classe de test **[TestHttpClientDao]** est la suivante :

```
1.  import unittest
2.
3.  from Logger import Logger
4.
5.
6.  class TestHttpClientDao(unittest.TestCase):
7.
8.      def test_1(self) -> None:
9.          from TaxPayer import TaxPayer
10.
11.         # {'marié': 'oui', 'enfants': 2, 'salaire': 55555,
12.         # 'impôt': 2814, 'surcôte': 0, 'décôte': 0, 'réduction': 0, 'taux': 0.14}
13.         taxpayer = TaxPayer().fromdict({"marié": "oui", "enfants": 2, "salaire": 55555})
14.         dao.calculate_tax(taxpayer)
15.         # vérification
16.         self.assertAlmostEqual(taxpayer.impôt, 2815, delta=1)
17.         self.assertEqual(taxpayer.décôte, 0)
18.         self.assertEqual(taxpayer.réduction, 0)
19.         self.assertAlmostEqual(taxpayer.taux, 0.14, delta=0.01)
20.         self.assertEqual(taxpayer.surcôte, 0)
21.
22.     …
23.
24.
25. if __name__ == '__main__':
26.     # on configure l'application
27.     import config
28.     config = config.configure({})
29.
30.     # logger
31.     logger = Logger(config["logsFilename"])
32.     # on le mémorise dans la config
33.     config["logger"] = logger
34.     # on récupère la couche [dao]
35.     dao = config["layers"]["dao"]
36.
37.     # on exécute les méthodes de test
38.     print("tests en cours...")
39.     unittest.main()
```

- on crée une |configuration d'exécution| pour ce test ;
- on lance le serveur web avec tout son environnement ;
- on exécute le test ;

Les résultats sont les suivants :

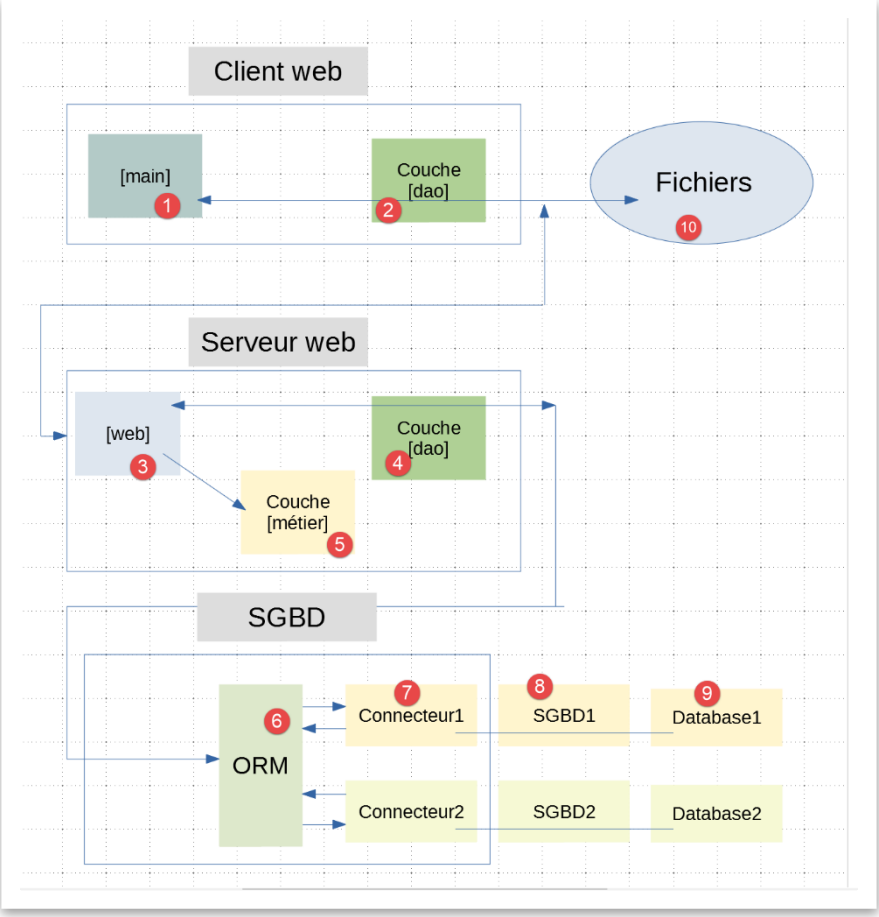
```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/http-clients/02/tests/TestHttpClientDao.py
2.  tests en cours...
3.  ..........
4.  ----------------------------------------------------------------------
5.  Ran 11 tests in 6.128s
6.
7.  OK
8.
9.  Process finished with exit code 0
```

25 Exercice d'application : version 8

25.1 Introduction

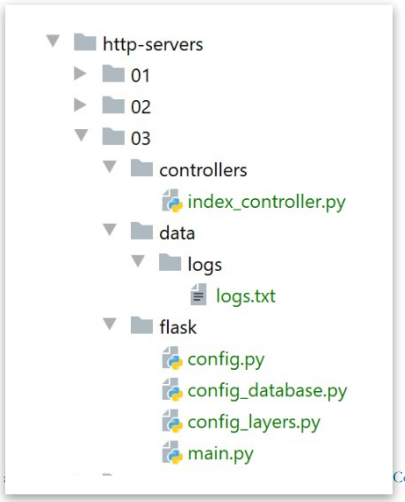
Nous allons écrire une nouvelle application client / serveur. La nouveauté du serveur est qu'il va gérer une session. Au lieu de mettre les données de l'administration fiscale dans un objet de portée **[application]**, on va les mettre dans un objet de portée **[session]**. Ce faisant, on régresse dans les performances du code. Lorsqu'un objet peut être partagé en lecture seule par tous les utilisateurs, il est préférable d'en faire un objet de portée **[application]** plutôt que de portée **[session]**. On gagne au minimum de la bande passante puisque qu'on diminue ainsi la taille du cookie de session. Mais nous voulons montrer une application client / serveur où le client et le serveur s'échangent un cookie de session.

L'architecture de l'application ne change pas :



25.2 Le serveur web

L'arborescence des scripts du serveur est la suivante :



Le dossier **[http-servers/03]** est obtenu initialement par recopie du dossier **[http-servers/02]**. On procède ensuite à des modifications.

25.2.1 La configuration

Elle est la même que dans la |version précédente| avec quelques modifications dans le script **[config]** :

```
1.  # dépendances absolues
2.      absolute_dependencies = [
3.          # dossiers du projet
4.          # BaseEntity, MyException
5.          f"{root_dir}/classes/02/entities",
6.          # InterfaceImpôtsDao, InterfaceImpôtsMétier, InterfaceImpôtsUi
7.          f"{root_dir}/impots/v04/interfaces",
8.          # AbstractImpôtsdao, ImpôtsConsole, ImpôtsMétier
9.          f"{root_dir}/impots/v04/services",
10.         # ImpotsDaoWithAdminDataInDatabase
11.         f"{root_dir}/impots/v05/services",
12.         # AdminData, ImpôtsError, TaxPayer
13.         f"{root_dir}/impots/v04/entities",
14.         # Constantes, Tranches
15.         f"{root_dir}/impots/v05/entities",
16.         # index_controller
17.         f"{script_dir}/../controllers",
18.         # scripts [config_database, config_layers]
19.         script_dir,
20.         # Logger, SendAdminMail
21.         f"{root_dir}/impots/http-servers/02/utilities",
22.     ]
```

- ligne 17 : on va réécrire un contrôleur pour la fonction **[index]** qui traite l'URL / ;
- ligne 21 : on utilise les utilitaires de la |version précédente| ;

25.2.2 Le script principal [main]

Le nouveau script **[main]** amène quelques modifications au script principal **[main]** de la version précédente :

```
1.  # l'application Flask peut démarrer
2.  app = Flask(__name__)
3.  # clé secrète de la session
4.  app.secret_key = os.urandom(12).hex()
```

- ligne 4 : on crée une clé secrète pour l'application. On sait que celle-ci est nécessaire pour gérer les sessions ;

Ensuite, les données fiscale ne sont plus demandées dans le code de **[main]**. Les lignes suivantes sont supprimées :

```
1.  # récupération des données de l'administration fiscale
2.  erreur = False
3.  try:
4.      # admindata sera une donnée de portée application en lecture seule
5.      config["admindata"] = config["layers"]["dao"].get_admindata()
6.      # log de réussite
7.      logger.write("[serveur] connexion à la base de données réussie\n")
8.  except ImpôtsError as ex:
9.      # on note l'erreur
10.     erreur = True
11.     # log d'erreur
12.     log = f"L'erreur suivante s'est produite : {ex}"
13.     # console
14.     print(log)
15.     # fichier de logs
16.     logger.write(f"{log}\n")
17.     # mail à l'administrateur
18.     send_adminmail(config, log)
```

Par ailleurs, le contrôleur **[index_controller]** admet un paramètre supplémentaire, la session Flask :

```
1.  from flask import request, Flask, session
2.  ….
3.          # on fait exécuter la requête par un contrôleur
4.          résultat, status_code = index_controller.execute(request, session, config)
```

25.2.3 Le contrôleur `[index_controller]`

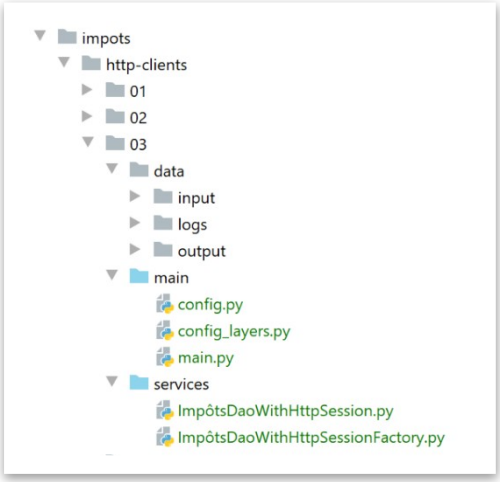
Le contrôleur **`[index_controller]`** gère désormais une session :

```
1.  # import des dépendances
2.  import os
3.  import re
4.  import threading
5.
6.  from flask_api import status
7.  from werkzeug.local import LocalProxy
8.
9.  # URL paramétrée : /?marié=xx&enfants=yy&salaire=zz
10. from AdminData import AdminData
11. from ImpôtsError import ImpôtsError
12.
13.
14. def execute(request: LocalProxy, session: LocalProxy, config: dict) -> tuple:
15.     # dépendances
16.     from TaxPayer import TaxPayer
17.
18.     # au départ pas d'erreurs
19.     erreurs = []
20.     …
21.
22.     # des erreurs ?
23.     if erreurs:
24.         # on rend une réponse d'erreur au client
25.         return {"réponse": {"erreurs": erreurs}}, status.HTTP_400_BAD_REQUEST
26.
27.     # pas d'erreurs, on peut travailler
28.     # on récupère la config associée au thread
29.     thread_name = threading.current_thread().name
30.     logger = config[thread_name]["config"]["logger"]
31.     # on exécute la requête
32.     réponse = None
33.     try:
34.         # le cas le + simple - admindata est déjà en session
35.         if session.get('client_id') is not None:
36.             # on récupère les informations de la session
37.             client_id = session.get('client_id')
38.             admindata = AdminData().fromdict(session.get('admindata'))
39.             # log
40.             logger.write(f"[index_controller] client [{client_id}], données fiscales prises en session\n")
41.         else:
42.             # récupération des données de l'administration fiscale
43.             admindata = config["layers"]["dao"].get_admindata()
44.             # mise en session de admindata
45.             session['admindata'] = admindata.asdict()
46.             # on donne un n° au client et on met ce n° en session
47.             # cela va nous permettre de le suivre dans les logs du serveur
48.             client_id = os.urandom(12).hex()
49.             session['client_id'] = client_id
50.             # log
51.             logger.write(f"[index_controller] client [{client_id}], données fiscales prises dans la couche dao\n")
52.         # calcul de l'impôt
53.         taxpayer = TaxPayer().fromdict({'marié': marié, 'enfants': enfants, 'salaire': salaire})
54.         config["layers"]["métier"].calculate_tax(taxpayer, admindata)
55.         # on rend la réponse au client
56.         return {"réponse": {"result": taxpayer.asdict()}}, status.HTTP_200_OK
57.     except (BaseException, ImpôtsError) as erreur:
58.         # on rend la réponse au client
59.         return {"réponse": {"erreurs": [f"{erreur}"]}}, status.HTTP_500_INTERNAL_SERVER_ERROR
```

- ligne 14 : le contrôleur reçoit la session courante du client web ;
- lignes 35-38 : si le client a une session, alors celle-ci contient deux clés :
  - **`[client_id]`** : un n° de client (ligne 37) ;
  - **`[admindata]`** : les données de l'administration fiscale sous le forme d'un dictionnaire (ligne 38) ;
- ligne 35 : on regarde si la session a une des deux clés attendues ;
- lignes 42-51 : cas où la session du client n'a pas encore été initialisée ;
  - ligne 43 : on récupère les données de l'administration fiscale auprès de la couche **`[dao]`** ;
  - ligne 45 : ces données sont mises dans la session sous la forme d'un dictionnaire ;
  - ligne 48 : on affecte un n° aléatoire au client. Ce n° sera différent pour chaque client ;
  - ligne 49 : ce n° est mis en session ;
  - ligne 51 : on logue le fait que les données de l'administration fiscale ont été obtenues par la couche **`[dao]`**. Les accès à la couche **`[dao]`** sont en général coûteux. C'est pourquoi il faut les limiter. L'idée ici est d'obtenir une fois les données fiscales auprès de la couche **`[dao]`**, de les mettre en session et d'aller les chercher là lors des requêtes ultérieures du même client. On rappelle que ce n'est pas la meilleure solution. Les données fiscales de l'administration étant les mêmes pour tous les clients, leur place est dans un objet de portée application ;

- lignes 35-40 : cas où la session du client a été initialisée lors d'une précédente requête ;
  - ligne 37 : on récupère le n° du client dans la session ;
  - ligne 38 : on récupère les données fiscales de l'administration dans la session ;
  - ligne 40 : on logue le fait que le client a obtenu les données fiscales de l'administration dans la session ;

25.3 Le client web



25.3.1 La couche [dao]

25.3.1.1 La classe [ImpôtsDaoWithHttpSession]

La couche **[dao]** est implémentée par la classe **[ImpôtsDaoWithHttpSession]** suivante :

```
1.  # imports
2.
3.  import requests
4.  from flask_api import status
5.  from myutils import decode_flask_session
6.
7.  from AbstractImpôtsDao import AbstractImpôtsDao
8.  from AdminData import AdminData
9.  from ImpôtsError import ImpôtsError
10. from InterfaceImpôtsMétier import InterfaceImpôtsMétier
11. from TaxPayer import TaxPayer
12.
13.
14. class ImpôtsDaoWithHttpSession(AbstractImpôtsDao, InterfaceImpôtsMétier):
15.
16.     # constructeur
17.     def __init__(self, config: dict):
18.         # initialisation parent
19.         AbstractImpôtsDao.__init__(self, config)
20.         # mémorisation éléments de la configuration
21.         # config générale
22.         self.__config = config
23.         # serveur
24.         self.__config_server = config["server"]
25.         # mode debug
26.         self.__debug = config["debug"]
27.         # logger
28.         self.__logger = None
29.         # cookies
30.         self.__cookies = None
31.
32.     # méthode inutilisée
33.     def get_admindata(self) -> AdminData:
34.         pass
35.
36.     # calcul de l'impôt
37.     def calculate_tax(self, taxpayer: TaxPayer, admindata: AdminData = None):
38.         # on laisse remonter les exceptions
39.         # paramètres du get
40.         params = {"marié": taxpayer.marié, "enfants": taxpayer.enfants, "salaire": taxpayer.salaire}
41.         # connexion avec authentification Auth Basic ?
42.         if self.__config_server['authBasic']:
43.             response = requests.get(
44.                 # URL du serveur interrogé
```

```
45.                 self.__config_server['urlServer'],
46.                 # paramètres de l'URL
47.                 params=params,
48.                 # authentification Basic
49.                 auth=(
50.                     self.__config_server["user"]["login"],
51.                     self.__config_server["user"]["password"]),
52.                 cookies=self.__cookies)
53.
54.         else:
55.             # connexion sans authentification Auth Basic
56.             response = requests.get(self.__config_server['urlServer'], params=params, cookies=self.__cookies)
57.         # on récupère les cookies de la réponse s'il y en a
58.         if response.cookies:
59.             self.__cookies = response.cookies
60.             # on récupère le cookie de session
61.             session_cookie = response.cookies.get('session')
62.             # on le décode pour le loguer
63.             if session_cookie:
64.                 # logueur
65.                 if not self.__logger:
66.                     self.__logger = self.__config['logger']
67.                 # on logue
68.                 self.__logger.write(f"cookie de session={decode_flask_session(session_cookie)}\n")
69.
70.         # mode debug ?
71.         if self.__debug:
72.             # logueur
73.             if not self.__logger:
74.                 self.__logger = self.__config['logger']
75.             # on logue
76.             self.__logger.write(f"{response.text}\n")
77.         # code de statut de la réponse HTTP
78.         status_code = response.status_code
79.         # on met la réponse jSON dans un dictionnaire
80.         résultat = response.json()
81.         # erreur si code de statut différent de 200 OK
82.         if status_code != status.HTTP_200_OK:
83.             # on sait que les erreurs ont été associées à la clé [erreurs] de la réponse
84.             raise ImpôtsError(87, résultat['réponse']['erreurs'])
85.         # on sait que le résultat a été associé à la clé [result] de la réponse
86.         # on modifie le paramètre d'entrée avec ce résultat
87.         taxpayer.fromdict(résultat["réponse"]["result"])
```

- ligne 30 : la couche **[dao]** va gérer un dictionnaire de cookies ;
- ligne 58 : la propriété **[response.cookies]** est un dictionnaire rassemblant les cookies envoyés par le serveur dans les entêtes HTTP **[Set-Cookie]** ;
- ligne 59 : ces cookies sont mémorisés dans la couche **[dao]**. Ils seront renvoyés au serveur lors des requêtes ultérieures du même client ;
- lignes 60-68 : bien que ce ne soit pas indispensable on récupère le cookie de session. Dans le dictionnaire des cookies envoyés par le serveur, le cookie de session est associé à la clé **[session]** ;
- lignes 62-68 : on décode le cookie de session pour le loguer ;
- ligne 68 : nous reviendrons ultérieurement sur la fonction **[decode_flask_session]** qui décode le cookie de session ;
- lignes 52 et 57 : à chaque requête du même client, les cookies envoyés par le serveur lui sont renvoyés. C'est de cette façon que la session Flask est maintenue entre le client et le serveur ;

Il faut se souvenir maintenant que la couche **[dao]** va être exécutée simultanément par plusieurs threads. Il faut donc regarder toutes les propriétés de l'instance de classe et voir si l'accès simultané à ces propriétés pose problème. Ici nous avons ajouté la propriété **[self.__cookies]**, ligne 30. Cette propriété est modifiée à la ligne 59. On a donc un accès en écriture à une donnée partagée par tous les threads. Or cet accès pose problème : chaque thread représentant un client donné a son propre cookie de session. En effet, dedans il y a un n° de client (=thread) unique pour chaque client. Si on ne fait rien, le thread T2 peut écraser les cookies du thread T1.

On a déjà vu une méthode pour gérer ce problème : on peut créer dans le fichier **[config]** passé en paramètre au constructeur (ligne 17), des clés différentes pour chaque thread. On peut par exemple utiliser comme clé le nom du thread :

- ligne 59, on pourrait écrire :

```
config[thread_name]['cookies']=cookies
```

- ligne 52, on pourrait alors écrire :

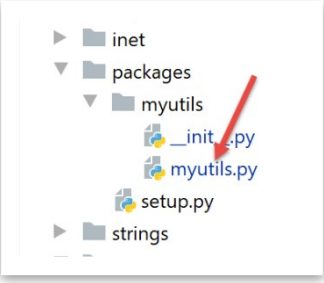
```
cookies=config[thread_name]['cookies']
```

Nous allons ici utiliser une technique différente : chaque thread (=client) aura sa propre couche **[dao]**. Ainsi la ligne 59 ne pose plus problème car les cookies utilisés sont alors ceux d'un unique client.

Pour cela, nous allons créer une nouvelle classe **[ImpôtsDaoWithHttpSessionFactory]**.

25.3.1.2 La fonction de décodage de la session Flask

La fonction **[decode_flask_session]** est définie dans le script **[myutils]** :



Nous avons déjà étudié le script |myutils|. Ce script est un module de portée machine que les différents scripts de ce cours peuvent importer avec l'instruction :

```
import myutils
```

On y définit la fonction **[decode_flask_session]** de la façon suivante :

```
1.  def decode_flask_session(cookie: str) -> str:
2.      # source : https://www.kirsle.net/wizards/flask-session.cgi
3.      compressed = False
4.      payload = cookie
5.
6.      if payload.startswith('.'):
7.          compressed = True
8.          payload = payload[1:]
9.
10.     data = payload.split(".")[0]
11.
12.     data = base64_decode(data)
13.     if compressed:
14.         data = zlib.decompress(data)
15.
16.     return data.decode("utf-8")
```

- ligne 2 : l'URL où j'ai trouvé cette fonction ;
- ligne 1 : le paramètre **[cookie]** est la chaîne de caractères associée à la clé **[session]** dans le dictionnaire des cookies reçus par un client web ;
- lignes 3-16 : je ne commenterai pas ce code que je ne maîtrise pas ;

On ajoute une nouvelle importation dans le fichier **[__init__.py]** :

```
from .myutils import set_syspath, json_response, decode_flask_session
```

La nouvelle version de **[myutils]** est installée parmi les modules de portée machine avec la commande **[pip install .]** dans un terminal Pycharm :

```
60. (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\packages>pip install .
61. Processing c:\data\st-2020\dev\python\cours-2020\python3-flask-2020\packages
62. Using legacy setup.py install for myutils, since package 'wheel' is not installed.
63. Installing collected packages: myutils
64.   Attempting uninstall: myutils
65.     Found existing installation: myutils 0.1
66.     Uninstalling myutils-0.1:
67.       Successfully uninstalled myutils-0.1
68.     Running setup.py install for myutils ... done
69. Successfully installed myutils-0.1
```

- ligne 1 : il faut être dans le dossier **[packages]** pour taper cette instruction ;

25.3.1.3 La classe [ImpôtsDaoWithHttpSessionFactory]

La classe **[ImpôtsDaoWithHttpSessionFactory]** est la suivante :

```
1.  from ImpôtsDaoWithHttpSession import ImpôtsDaoWithHttpSession
2.
3.
4.  class ImpôtsDaoWithHttpSessionFactory:
5.
6.      def __init__(self, config: dict):
7.          # on mémorise le paramètre
8.          self.__config = config
9.
10.     def new_instance(self):
11.         # on rend une instance de la couche [dao]
12.         return ImpôtsDaoWithHttpSession(self.__config)
```

- la classe **[ImpôtsDaoWithHttpSessionFactory]** permet de créer une nouvelle implémentation de la couche **[dao]** avec la méthode **[new_instance]** des lignes 10-12 ;

25.3.2 La configuration

Le script **[config_layers]** qui configure les couches du client web est modifié de la façon suivante :

```
1.  def configure(config: dict) -> dict:
2.      # instanciation des couches de l'applicatuon
3.
4.      # couche dao
5.      from ImpôtsDaoWithHttpSessionFactory import ImpôtsDaoWithHttpSessionFactory
6.      dao_factory = ImpôtsDaoWithHttpSessionFactory(config)
7.
8.      # on rend la configuation des couches
9.      return {
10.         "dao_factory": dao_factory
11.     }
```

- lignes 5-6 : au lieu d'instancier une couche **[dao]** unique comme c'était fait précédemment, on instancie une 'factory' de cette couche (factory=usine de production d'objets, ici la couche **[dao]**) ;
- lignes 9-11 : on rend la configuration des couches ;

25.3.3 Le script principal du client

Le script **[main]** évolue de la façon suivante par rapport à celui de la version précédente :

```
1.  # on configure l'application
2.
3.  import config
4.  config = config.configure({})
5.
6.  # dépendances
7.  from ImpôtsError import ImpôtsError
8.  import random
9.  import sys
10. import threading
11. from Logger import Logger
12.
13.
14. # exécution de la couche [dao] dans un thread
15. # taxpayers est une liste de contribuables
16. def thread_function(thread_dao, logger, taxpayers: list):
17.     …
18.
19.
20. # liste des threads du client
21. threads = []
22. logger = None
23. # code
24. try:
25.     …
26.     l_taxpayers = len(taxpayers)
27.     while i < len(taxpayers):
28.         …
29.         # chaque thread doit avoir sa propre couche [dao] pour gérer correctement son cookie de session
30.         thread_dao = dao_factory.new_instance()
31.         # on crée le thread
32.         thread = threading.Thread(target=thread_function, args=(thread_dao, logger, thread_taxpayers))
33.         # on l'ajoute à la liste des threads du script principal
```

```
34.             threads.append(thread)
35.             # on lance le thread - cette opération est asynchrone - on n'attend pas le résultat du thread
36.             thread.start()
37.         # le thread principal attend la fin de tous les threads qu'il a lancés
38.         …
39. except BaseException as erreur:
40.     # affichage de l'erreur
41.     print(f"L'erreur suivante s'est produite : {erreur}")
42. finally:
43.     # on ferme le logueur
44.     if logger:
45.         logger.close()
46.     # on a fini
47.     print("Travail terminé...")
48.     # fin des threads qui pourraient encore exister si on s'est arrêté sur erreur
49.     sys.exit()
```

- lignes 29-30 : chaque thread a sa couche **[dao]** ;

25.3.4 Exécution du client

Le serveur web est lancé, le SGBD est lancé, le serveur de mails **[hMailServer]** est lancé. Puis on lance le script **[main]** du client web. Les logs de l'exécution dans **[data/logs/logs.txt]** sont alors les suivants :

```
1.  2020-07-25 10:21:46.478511, Thread-1 : début du thread [Thread-1] avec 1 contribuable(s)
2.  2020-07-25 10:21:46.479111, Thread-1 : début du calcul de l'impôt de {"id": 1, "marié": "oui", "enfants":
    2, "salaire": 55555}
3.  2020-07-25 10:21:46.479111, Thread-2 : début du thread [Thread-2] avec 1 contribuable(s)
4.  2020-07-25 10:21:46.480195, Thread-3 : début du thread [Thread-3] avec 2 contribuable(s)
5.  2020-07-25 10:21:46.480195, Thread-2 : début du calcul de l'impôt de {"id": 2, "marié": "oui", "enfants":
    2, "salaire": 50000}
6.  2020-07-25 10:21:46.481137, Thread-4 : début du thread [Thread-4] avec 3 contribuable(s)
7.  2020-07-25 10:21:46.481137, Thread-3 : début du calcul de l'impôt de {"id": 3, "marié": "oui", "enfants":
    3, "salaire": 50000}
8.  2020-07-25 10:21:46.482279, Thread-5 : début du thread [Thread-5] avec 3 contribuable(s)
9.  2020-07-25 10:21:46.482622, Thread-6 : début du thread [Thread-6] avec 1 contribuable(s)
10. 2020-07-25 10:21:46.482622, Thread-4 : début du calcul de l'impôt de {"id": 5, "marié": "non", "enfants":
    3, "salaire": 100000}
11. 2020-07-25 10:21:46.485923, Thread-5 : début du calcul de l'impôt de {"id": 8, "marié": "non", "enfants":
    0, "salaire": 100000}
12. 2020-07-25 10:21:46.485923, Thread-6 : début du calcul de l'impôt de {"id": 11, "marié": "oui", "enfants":
    3, "salaire": 200000}
13. 2020-07-25 10:21:46.725910, Thread-4 : cookie de session={"admindata":
    {"abattement_dixpourcent_max":12502.0,"abattement_dixpourcent_min":437.0,"coeffn":
    [0.0,1394.96,5798.0,13913.7,20163.4],"coeffr":[0.0,0.14,0.3,0.41,0.45],"id":1,"limites":
    [9964.0,27519.0,73779.0,156244.0,90000.0],"plafond_decote_celibataire":1196.0,"plafond_decote_couple":1970.
    0,"plafond_impot_celibataire_pour_decote":1595.0,"plafond_impot_couple_pour_decote":2627.0,"plafond_qf_demi
    _part":1551.0,"plafond_revenus_celibataire_pour_reduction":21037.0,"plafond_revenus_couple_pour_reduction":
    42074.0,"valeur_reduc_demi_part":3797.0},"client_id":"fa3c83b82761c83e13217967"}
14. 2020-07-25 10:21:46.725910, Thread-4 : {"réponse": {"result": {"marié": "non", "enfants": 3, "salaire":
    100000, "impôt": 16782, "surcôte": 7176, "taux": 0.41, "décôte": 0, "réduction": 0}}}
15. 2020-07-25 10:21:46.725910, Thread-4 : fin du calcul de l'impôt de {"id": 5, "marié": "non", "enfants": 3,
    "salaire": 100000, "impôt": 16782, "surcôte": 7176, "taux": 0.41, "décôte": 0, "réduction": 0}
16. 2020-07-25 10:21:46.726960, Thread-4 : début du calcul de l'impôt de {"id": 6, "marié": "oui", "enfants":
    3, "salaire": 100000}
17. 2020-07-25 10:21:47.514108, Thread-3 : cookie de session={"admindata":
    {"abattement_dixpourcent_max":12502.0,"abattement_dixpourcent_min":437.0,"coeffn":
    [0.0,1394.96,5798.0,13913.7,20163.4],"coeffr":[0.0,0.14,0.3,0.41,0.45],"id":1,"limites":
    [9964.0,27519.0,73779.0,156244.0,24999.5],"plafond_decote_celibataire":1196.0,"plafond_decote_couple":1970.
    0,"plafond_impot_celibataire_pour_decote":1595.0,"plafond_impot_couple_pour_decote":2627.0,"plafond_qf_demi
    _part":1551.0,"plafond_revenus_celibataire_pour_reduction":21037.0,"plafond_revenus_couple_pour_reduction":
    42074.0,"valeur_reduc_demi_part":3797.0},"client_id":"700e3f5dc808c7c48f0c9007"}
18. 2020-07-25 10:21:47.514610, Thread-3 : {"réponse": {"result": {"marié": "oui", "enfants": 3, "salaire":
    50000, "impôt": 0, "surcôte": 0, "taux": 0.14, "décôte": 720, "réduction": 0}}}
19. 2020-07-25 10:21:47.514939, Thread-3 : fin du calcul de l'impôt de {"id": 3, "marié": "oui", "enfants": 3,
    "salaire": 50000, "impôt": 0, "surcôte": 0, "taux": 0.14, "décôte": 720, "réduction": 0}
20. 2020-07-25 10:21:47.514939, Thread-3 : début du calcul de l'impôt de {"id": 4, "marié": "non", "enfants":
    2, "salaire": 100000}
21. 2020-07-25 10:21:47.527211, Thread-5 : cookie de session={"admindata":
    {"abattement_dixpourcent_max":12502.0,"abattement_dixpourcent_min":437.0,"coeffn":
    [0.0,1394.96,5798.0,13913.7,20163.4],"coeffr":[0.0,0.14,0.3,0.41,0.45],"id":1,"limites":
    [9964.0,27519.0,73779.0,156244.0,90000.0],"plafond_decote_celibataire":1196.0,"plafond_decote_couple":1970.
    0,"plafond_impot_celibataire_pour_decote":1595.0,"plafond_impot_couple_pour_decote":2627.0,"plafond_qf_demi
    _part":1551.0,"plafond_revenus_celibataire_pour_reduction":21037.0,"plafond_revenus_couple_pour_reduction":
    42074.0,"valeur_reduc_demi_part":3797.0},"client_id":"9e14a5d4a3057f69ab95ab2d"}
```

```
22. 2020-07-25 10:21:47.527211, Thread-2 : cookie de session={"admindata":
    {"abattement_dixpourcent_max":12502.0,"abattement_dixpourcent_min":437.0,"coeffn":
    [0.0,1394.96,5798.0,13913.7,20163.4],"coeffr":[0.0,0.14,0.3,0.41,0.45],"id":1,"limites":
    [9964.0,27519.0,73779.0,156244.0,22500.0],"plafond_decote_celibataire":1196.0,"plafond_decote_couple":1970.
    0,"plafond_impot_celibataire_pour_decote":1595.0,"plafond_impot_couple_pour_decote":2627.0,"plafond_qf_demi
    _part":1551.0,"plafond_revenus_celibataire_pour_reduction":21037.0,"plafond_revenus_couple_pour_reduction":
    42074.0,"valeur_reduc_demi_part":3797.0},"client_id":"a06e8fd70a44c9e311f4dce0"}
23. 2020-07-25 10:21:47.527211, Thread-5 : {"réponse": {"result": {"marié": "non", "enfants": 0, "salaire":
    100000, "impôt": 22986, "surcôte": 0, "taux": 0.41, "décôte": 0, "réduction": 0}}}
24. 2020-07-25 10:21:47.527211, Thread-1 : cookie de session={"admindata":
    {"abattement_dixpourcent_max":12502.0,"abattement_dixpourcent_min":437.0,"coeffn":
    [0.0,1394.96,5798.0,13913.7,20163.4],"coeffr":[0.0,0.14,0.3,0.41,0.45],"id":1,"limites":
    [9964.0,27519.0,73779.0,156244.0,90000.0],"plafond_decote_celibataire":1196.0,"plafond_decote_couple":1970.
    0,"plafond_impot_celibataire_pour_decote":1595.0,"plafond_impot_couple_pour_decote":2627.0,"plafond_qf_demi
    _part":1551.0,"plafond_revenus_celibataire_pour_reduction":21037.0,"plafond_revenus_couple_pour_reduction":
    42074.0,"valeur_reduc_demi_part":3797.0},"client_id":"28c38df998f67685b3a482b8"}
25. 2020-07-25 10:21:47.527211, Thread-2 : {"réponse": {"result": {"marié": "oui", "enfants": 2, "salaire":
    50000, "impôt": 1384, "surcôte": 0, "taux": 0.14, "décôte": 384, "réduction": 347}}}
26. 2020-07-25 10:21:47.528341, Thread-5 : fin du calcul de l'impôt de {"id": 8, "marié": "non", "enfants": 0,
    "salaire": 100000, "impôt": 22986, "surcôte": 0, "taux": 0.41, "décôte": 0, "réduction": 0}
27. 2020-07-25 10:21:47.528341, Thread-1 : {"réponse": {"result": {"marié": "oui", "enfants": 2, "salaire":
    55555, "impôt": 2814, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0}}}
28. 2020-07-25 10:21:47.528842, Thread-2 : fin du calcul de l'impôt de {"id": 2, "marié": "oui", "enfants": 2,
    "salaire": 50000, "impôt": 1384, "surcôte": 0, "taux": 0.14, "décôte": 384, "réduction": 347}
29. 2020-07-25 10:21:47.529349, Thread-5 : début du calcul de l'impôt de {"id": 9, "marié": "oui", "enfants":
    2, "salaire": 30000}
30. 2020-07-25 10:21:47.529699, Thread-1 : fin du calcul de l'impôt de {"id": 1, "marié": "oui", "enfants": 2,
    "salaire": 55555, "impôt": 2814, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0}
31. 2020-07-25 10:21:47.529699, Thread-2 : fin du thread [Thread-2]
32. 2020-07-25 10:21:47.531905, Thread-1 : fin du thread [Thread-1]
33. 2020-07-25 10:21:47.536121, Thread-6 : cookie de session={"admindata":
    {"abattement_dixpourcent_max":12502.0,"abattement_dixpourcent_min":437.0,"coeffn":
    [0.0,1394.96,5798.0,13913.7,20163.4],"coeffr":[0.0,0.14,0.3,0.41,0.45],"id":1,"limites":
    [9964.0,27519.0,73779.0,156244.0,93749.0],"plafond_decote_celibataire":1196.0,"plafond_decote_couple":1970.
    0,"plafond_impot_celibataire_pour_decote":1595.0,"plafond_impot_couple_pour_decote":2627.0,"plafond_qf_demi
    _part":1551.0,"plafond_revenus_celibataire_pour_reduction":21037.0,"plafond_revenus_couple_pour_reduction":
    42074.0,"valeur_reduc_demi_part":3797.0},"client_id":"38499b63076516c02f2770ec"}
34. 2020-07-25 10:21:47.537161, Thread-3 : {"réponse": {"result": {"marié": "non", "enfants": 2, "salaire":
    100000, "impôt": 19884, "surcôte": 4480, "taux": 0.41, "décôte": 0, "réduction": 0}}}
35. 2020-07-25 10:21:47.537161, Thread-6 : {"réponse": {"result": {"marié": "oui", "enfants": 3, "salaire":
    200000, "impôt": 42842, "surcôte": 17283, "taux": 0.41, "décôte": 0, "réduction": 0}}}
36. 2020-07-25 10:21:47.538156, Thread-3 : fin du calcul de l'impôt de {"id": 4, "marié": "non", "enfants": 2,
    "salaire": 100000, "impôt": 19884, "surcôte": 4480, "taux": 0.41, "décôte": 0, "réduction": 0}
37. 2020-07-25 10:21:47.538557, Thread-6 : fin du calcul de l'impôt de {"id": 11, "marié": "oui", "enfants": 3,
    "salaire": 200000, "impôt": 42842, "surcôte": 17283, "taux": 0.41, "décôte": 0, "réduction": 0}
38. 2020-07-25 10:21:47.538828, Thread-3 : fin du thread [Thread-3]
39. 2020-07-25 10:21:47.538828, Thread-6 : fin du thread [Thread-6]
40. 2020-07-25 10:21:47.546198, Thread-5 : {"réponse": {"result": {"marié": "oui", "enfants": 2, "salaire":
    30000, "impôt": 0, "surcôte": 0, "taux": 0.0, "décôte": 0, "réduction": 0}}}
41. 2020-07-25 10:21:47.546198, Thread-5 : fin du calcul de l'impôt de {"id": 9, "marié": "oui", "enfants": 2,
    "salaire": 30000, "impôt": 0, "surcôte": 0, "taux": 0.0, "décôte": 0, "réduction": 0}
42. 2020-07-25 10:21:47.546198, Thread-5 : début du calcul de l'impôt de {"id": 10, "marié": "non", "enfants":
    0, "salaire": 200000}
43. 2020-07-25 10:21:47.739643, Thread-4 : {"réponse": {"result": {"marié": "oui", "enfants": 3, "salaire":
    100000, "impôt": 9200, "surcôte": 2180, "taux": 0.3, "décôte": 0, "réduction": 0}}}
44. 2020-07-25 10:21:47.739643, Thread-4 : fin du calcul de l'impôt de {"id": 6, "marié": "oui", "enfants": 3,
    "salaire": 100000, "impôt": 9200, "surcôte": 2180, "taux": 0.3, "décôte": 0, "réduction": 0}
45. 2020-07-25 10:21:47.740668, Thread-4 : début du calcul de l'impôt de {"id": 7, "marié": "oui", "enfants":
    5, "salaire": 100000}
46. 2020-07-25 10:21:48.557469, Thread-5 : {"réponse": {"result": {"marié": "non", "enfants": 0, "salaire":
    200000, "impôt": 64210, "surcôte": 7498, "taux": 0.45, "décôte": 0, "réduction": 0}}}
47. 2020-07-25 10:21:48.558715, Thread-5 : fin du calcul de l'impôt de {"id": 10, "marié": "non", "enfants": 0,
    "salaire": 200000, "impôt": 64210, "surcôte": 7498, "taux": 0.45, "décôte": 0, "réduction": 0}
48. 2020-07-25 10:21:48.558715, Thread-5 : fin du thread [Thread-5]
49. 2020-07-25 10:21:48.753025, Thread-4 : {"réponse": {"result": {"marié": "oui", "enfants": 5, "salaire":
    100000, "impôt": 4230, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0}}}
50. 2020-07-25 10:21:48.753318, Thread-4 : fin du calcul de l'impôt de {"id": 7, "marié": "oui", "enfants": 5,
    "salaire": 100000, "impôt": 4230, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0}
51. 2020-07-25 10:21:48.753540, Thread-4 : fin du thread [Thread-4]
```

- on a au total 6 threads donc 6 clients (lignes 1, 3, 4, 6, 8, 9) qui interrogent simultanément le serveur de calcul de l'impôt ;
- nous allons suivre le thread **[Thread-4]** qui gère 3 contribuables (ligne 6). Il va faire séquentiellement trois requêtes au serveur de calcul de l'impôt ;
- ligne 10 : la 1ère requête de **[Thread-4]** ;
- ligne 13 : **[Thread-4]** a reçu la réponse à sa 1ère requête. Dedans elle trouve un cookie de session dans lequel il y a le n° **[fa3c83b82761c83e13217967]** que lui a attribué le serveur ;

- ligne 14 : l'impôt du 1er contribuable ;
- ligne 16 : **[Thread-4]** fait une requête pour le 2ième contribuable ;
- ligne 43 : **[Thread-4]** reçoit l'impôt du 2ième contribuable ;
- ligne 45 : **[Thread-4]** fait une requête pour le 3ième contribuable ;
- ligne 49 : **[Thread-4]** reçoit l'impôt du 3ième contribuable ;
- ligne 51 : **[Thread-4]** a terminé son travail ;

Maintenant, regardons comment les 3 requêtes de **[Thread-4]** ont été traitées côté serveur. On va pouvoir le suivre dans les logs du serveur grâce à son n° de client **[fa3c83b82761c83e13217967]**.

Les logs **[data/logs/logs.txt]** côté serveur sont les suivants :

```
1.  2020-07-25 10:21:39.187366, MainThread : [serveur] démarrage du serveur
2.  2020-07-25 10:21:40.439093, MainThread : [serveur] démarrage du serveur
3.  2020-07-25 10:21:46.502011, Thread-2 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=3&salaire=50000' [GET]>
4.  2020-07-25 10:21:46.504049, Thread-2 : [index] mis en pause du thread pendant 1 seconde(s)
5.  2020-07-25 10:21:46.505452, Thread-3 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=2&salaire=55555' [GET]>
6.  2020-07-25 10:21:46.506257, Thread-3 : [index] mis en pause du thread pendant 1 seconde(s)
7.  2020-07-25 10:21:46.507292, Thread-4 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=non&enfants=0&salaire=100000' [GET]>
8.  2020-07-25 10:21:46.507292, Thread-4 : [index] mis en pause du thread pendant 1 seconde(s)
9.  2020-07-25 10:21:46.508301, Thread-5 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=2&salaire=50000' [GET]>
10. 2020-07-25 10:21:46.509293, Thread-5 : [index] mis en pause du thread pendant 1 seconde(s)
11. 2020-07-25 10:21:46.511808, Thread-6 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=non&enfants=3&salaire=100000' [GET]>
12. 2020-07-25 10:21:46.517604, Thread-7 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=3&salaire=200000' [GET]>
13. 2020-07-25 10:21:46.517604, Thread-7 : [index] mis en pause du thread pendant 1 seconde(s)
14. 2020-07-25 10:21:46.719504, Thread-6 : [index_controller] client [fa3c83b82761c83e13217967], données
    fiscales prises dans la couche dao
15. 2020-07-25 10:21:46.720003, Thread-6 : [index] {'réponse': {'result': {'marié': 'non', 'enfants': 3,
    'salaire': 100000, 'impôt': 16782, 'surcôte': 7176, 'taux': 0.41, 'décôte': 0, 'réduction': 0}}}
16. 2020-07-25 10:21:46.736108, Thread-8 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=3&salaire=100000' [GET]>
17. 2020-07-25 10:21:46.736108, Thread-8 : [index] mis en pause du thread pendant 1 seconde(s)
18. 2020-07-25 10:21:47.506709, Thread-2 : [index_controller] client [700e3f5dc808c7c48f0c9007], données
    fiscales prises dans la couche dao
19. 2020-07-25 10:21:47.507216, Thread-2 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 3,
    'salaire': 50000, 'impôt': 0, 'surcôte': 0, 'taux': 0.14, 'décôte': 720, 'réduction': 0}}}
20. 2020-07-25 10:21:47.507216, Thread-3 : [index_controller] client [28c38df998f67685b3a482b8], données
    fiscales prises dans la couche dao
21. 2020-07-25 10:21:47.508442, Thread-4 : [index_controller] client [9e14a5d4a3057f69ab95ab2d], données
    fiscales prises dans la couche dao
22. 2020-07-25 10:21:47.508940, Thread-3 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 2,
    'salaire': 55555, 'impôt': 2814, 'surcôte': 0, 'taux': 0.14, 'décôte': 0, 'réduction': 0}}}
23. 2020-07-25 10:21:47.510506, Thread-4 : [index] {'réponse': {'result': {'marié': 'non', 'enfants': 0,
    'salaire': 100000, 'impôt': 22986, 'surcôte': 0, 'taux': 0.41, 'décôte': 0, 'réduction': 0}}}
24. 2020-07-25 10:21:47.511513, Thread-5 : [index_controller] client [a06e8fd70a44c9e311f4dce0], données
    fiscales prises dans la couche dao
25. 2020-07-25 10:21:47.514939, Thread-5 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 2,
    'salaire': 50000, 'impôt': 1384, 'surcôte': 0, 'taux': 0.14, 'décôte': 384, 'réduction': 347}}}
26. 2020-07-25 10:21:47.520727, Thread-7 : [index_controller] client [38499b63076516c02f2770ec], données
    fiscales prises dans la couche dao
27. 2020-07-25 10:21:47.523162, Thread-7 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 3,
    'salaire': 200000, 'impôt': 42842, 'surcôte': 17283, 'taux': 0.41, 'décôte': 0, 'réduction': 0}}}
28. 2020-07-25 10:21:47.530835, Thread-9 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=non&enfants=2&salaire=100000' [GET]>
29. 2020-07-25 10:21:47.531736, Thread-9 : [index_controller] client [700e3f5dc808c7c48f0c9007], données
    fiscales prises en session
30. 2020-07-25 10:21:47.531905, Thread-9 : [index] {'réponse': {'result': {'marié': 'non', 'enfants': 2,
    'salaire': 100000, 'impôt': 19884, 'surcôte': 4480, 'taux': 0.41, 'décôte': 0, 'réduction': 0}}}
31. 2020-07-25 10:21:47.541899, Thread-10 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=2&salaire=30000' [GET]>
32. 2020-07-25 10:21:47.542488, Thread-10 : [index_controller] client [9e14a5d4a3057f69ab95ab2d], données
    fiscales prises en session
33. 2020-07-25 10:21:47.542488, Thread-10 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 2,
    'salaire': 30000, 'impôt': 0, 'surcôte': 0, 'taux': 0.0, 'décôte': 0, 'réduction': 0}}}
34. 2020-07-25 10:21:47.553628, Thread-11 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=non&enfants=0&salaire=200000' [GET]>
35. 2020-07-25 10:21:47.553628, Thread-11 : [index] mis en pause du thread pendant 1 seconde(s)
36. 2020-07-25 10:21:47.736910, Thread-8 : [index_controller] client [fa3c83b82761c83e13217967], données
    fiscales prises en session
```

```
37. 2020-07-25 10:21:47.737191, Thread-8 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 3,
    'salaire': 100000, 'impôt': 9200, 'surcôte': 2180, 'taux': 0.3, 'décôte': 0, 'réduction': 0}}}
38. 2020-07-25 10:21:47.748226, Thread-12 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=5&salaire=100000' [GET]>
39. 2020-07-25 10:21:47.748226, Thread-12 : [index] mis en pause du thread pendant 1 seconde(s)
40. 2020-07-25 10:21:48.554695, Thread-11 : [index_controller] client [9e14a5d4a3057f69ab95ab2d], données
    fiscales prises en session
41. 2020-07-25 10:21:48.555070, Thread-11 : [index] {'réponse': {'result': {'marié': 'non', 'enfants': 0,
    'salaire': 200000, 'impôt': 64210, 'surcôte': 7498, 'taux': 0.45, 'décôte': 0, 'réduction': 0}}}
42. 2020-07-25 10:21:48.748753, Thread-12 : [index_controller] client [fa3c83b82761c83e13217967], données
    fiscales prises en session
43. 2020-07-25 10:21:48.748753, Thread-12 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 5,
    'salaire': 100000, 'impôt': 4230, 'surcôte': 0, 'taux': 0.14, 'décôte': 0, 'réduction': 0}}}
```

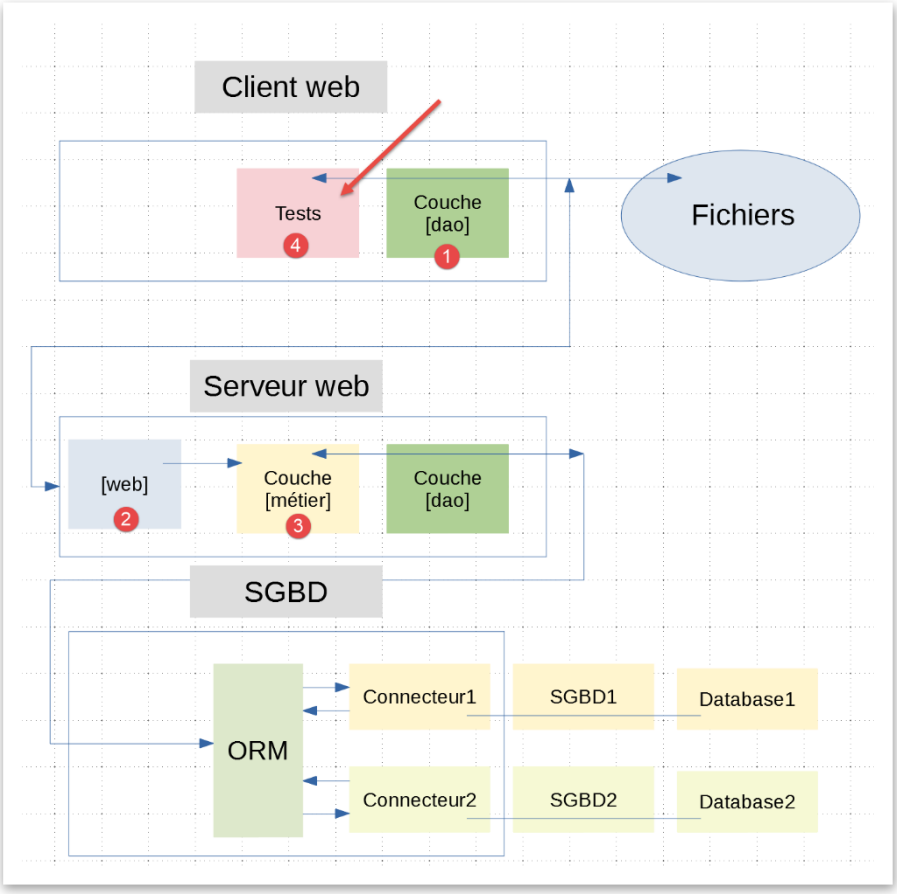
- on trouve le client **[fa3c83b82761c83e13217967]** pour la 1ère fois en ligne 14 : pour calculer l'impôt, le serveur a dû aller chercher en base les données de l'administration fiscale ;
- puis on retrouve le client **[fa3c83b82761c83e13217967]** en ligne 36. Cette fois-ci, le serveur trouve les données de l'administration fiscale en session, ce qui lui évite un accès, possiblement coûteux, à la couche **[dao]** ;
- on retrouve le client **[fa3c83b82761c83e13217967]** une 3ième fois en ligne 42, où là encore le serveur utilise la session du client ;

Cet exemple montre bien l'intérêt de la session pour un client : on y met des données partagées par toutes les requêtes de ce client et qui sont coûteuses à acquérir.

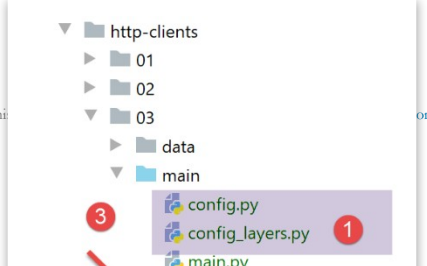
Côté client, les résultats dans le fichier **[data/output/résultats.json]** sont les mêmes que pour les versions précédentes.

25.4 Tests de la couche [dao]

Comme nous l'avons fait dans les |versions précédentes| nous testons la couche **[dao]** du client :



La classe de test sera exécutée dans l'environnement suivant :



- la configuration **[2]** est identique à la configuration **[1]** que nous venons d'étudier ;

La classe de test **[TestHttpClientDao]** est la suivante :

```
1.  import unittest
2.
3.  from Logger import Logger
4.
5.
6.  class TestHttpClientDao(unittest.TestCase):
7.
8.      def test_1(self) -> None:
9.          from TaxPayer import TaxPayer
10.
11.         # {'marié': 'oui', 'enfants': 2, 'salaire': 55555,
12.         # 'impôt': 2814, 'surcôte': 0, 'décôte': 0, 'réduction': 0, 'taux': 0.14}
13.         taxpayer = TaxPayer().fromdict({"marié": "oui", "enfants": 2, "salaire": 55555})
14.         dao.calculate_tax(taxpayer)
15.         # vérification
16.         self.assertAlmostEqual(taxpayer.impôt, 2815, delta=1)
17.         self.assertEqual(taxpayer.décôte, 0)
18.         self.assertEqual(taxpayer.réduction, 0)
19.         self.assertAlmostEqual(taxpayer.taux, 0.14, delta=0.01)
20.         self.assertEqual(taxpayer.surcôte, 0)
21. …
22.
23.
24. if __name__ == '__main__':
25.     # on configure l'application
26.     import config
27.     config = config.configure({})
28.
29.     # logger
30.     logger = Logger(config["logsFilename"])
31.     # on le mémorise dans la config
32.     config["logger"] = logger
33.     # on récupère la factory de la couche [dao]
34.     dao_factory = config["layers"]["dao_factory"]
35.     # on crée une instance de la couche [dao]
36.     dao = dao_factory.new_instance()
37.
38.     # on exécute les méthodes de test
39.     print("tests en cours...")
40.     unittest.main()
```

- on crée une |configuration d'exécution| pour ce test ;
- on lance le serveur web avec tout son environnement ;
- on exécute le test ;

Les résultats sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/http-clients/03/tests/TestHttpClientDao.py
2.  tests en cours...
3.  ..........
4.  ----------------------------------------------------------------------
5.  Ran 11 tests in 3.392s
6.
7.  OK
8.
9.  Process finished with exit code 0
```

26 Du dictionnaire à XML et vice-versa

Nous nous proposons ici de découvrir le module **[xml2dict]** qui permet de transformer :

- une chaîne XML en dictionnaire :
- un dictionnaire en chaîne XML ;

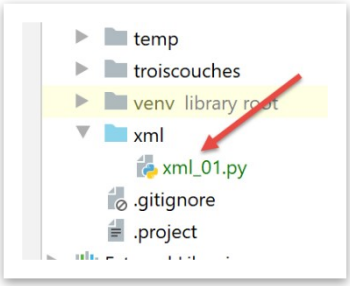
Avant l'avènement du jSON, la réponse des services web était souvent du XML (eXtended Markup Language). Par ailleurs, le protocole de ces services web était souvent SOAP (Simple Object Process Protocol). SOAP est un protocole qui s'appuie sur le protocole HTTP du web. Actuellement (2020), les services web sont plutôt de type REST (Representational State Transfer). Les services web que nous avons étudiés ne sont d'aucun de ces types mais sont définitivement plus proches de REST que de SOAP. Néanmoins je préfère dire qu'ils sont de type 'libre' ou 'inconnu' car ils ne respectent pas toutes les règles du REST.

Nous allons montrer combien il est facile de transformer nos architectures client / serveur jSON en architectures client / serveur XML. Il suffit d'utiliser le module **[xmltodict]**.

Nous commençons par l'installer dans un terminal Python :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\packages>pip install xmltodict
2.  Collecting xmltodict
3.    Using cached xmltodict-0.12.0-py2.py3-none-any.whl (9.2 kB)
4.  Installing collected packages: xmltodict
5.  Successfully installed xmltodict-0.12.0
```

Ceci fait, nous allons étudier sur un exemple ce qu'on peut faire avec ce module :



Le script **[xml_01]** est le suivant :

```
1.  from collections import OrderedDict
2.
3.  import xmltodict
4.
5.
6.  # xmltodict.parse pour passer du XML au dictionnaire. Le dictionnaire doit avoir une racine
7.  # le dictionnaire produit est de type OrderedDict
8.  # xmltodict.unparse pour passer du dictionnaire au XML
9.
10. def ordereddict2dict(ordered_dictionary) -> dict:
11.     …
12.
13.
14. def transform(message: str, dictionary: dict):
15.     # logs
16.     print(f"\n{message}-------")
17.     print(f"dictionnaire={dictionary}")
18.     # dict -> xml
19.     xml1 = xmltodict.unparse(dictionary)
20.     print(f"xml={xml1}")
21.     # xml -> OrderedDict
22.     ordereddict_dictionary1 = xmltodict.parse(xml1)
23.     print(f"ordereddict_dictionary1={ordereddict_dictionary1}")
24.     # OrderedDict -> dict
25.     print(f"dict_dictionary1={ordereddict2dict(ordereddict_dictionary1)}")
26.
27.
28. # test 1
29. transform("test 1", {"nom": "séléné"})
30. # test 2
31. transform("test 2", {"famille": {"père": {"prénom": "andré"}, "mère": {"prénom": "angèle"}, "nom": "séléné"}})
32. # test 3
33. transform("test 3", {"famille": {"nom": "séléné", "père": {"prénom": "andré"}, "mère": {"prénom": "angèle"},
34.                                  "hobbies": ["chant", "footing"]}})
35. # test 4
36. transform("test 4", {'réponse': {
```

```
37.     'erreurs': ['Méthode GET requise avec les seuls paramètres [marié, enfants, salaire]', 'paramètre [marié]
    manquant',
38.                 'paramètre [enfants] manquant', 'paramètre [salaire] manquant']}})
39. # test 5
40. transform("test 5", {'réponse': {
41.     'result': {'id': 0, 'marié': 'oui', 'enfants': 2, 'salaire': 50000, 'impôt': 1384, 'décôte': 384, 'surcôte': 0,
42.                'réduction': 347, 'taux': 0.14}}})
43. # test 6
44. transform("test 6", {"root": {'liste': ["un", "deux", "trois"]}})
45. # test 7
46. transform("test 7", {"root": {'liste': [{"un": [10, 11]}, {"deux": [20, 21]}, {"trois": [30, 31]}]}})
```

- lignes 14-25 : la fonction **[transform]** reçoit un texte à écrire **[message]** et un dictionnaire **[dictionary]** ;
- ligne 16 : affichage du message ;
- ligne 17 : on affiche le dictionnaire reçu ;
- lignes 19-20 : ce dictionnaire est transformé en chaîne XML et celle-ci est affichée. La méthode qui sait faire cela est **[xmltodict.unparse]** ;
- lignes 21-23 : la chaîne XML précédente est transformée en dictionnaire et celui-ci est affiché. La méthode qui sait faire cela est **[xmltodict.parse]**. Cette méthode ne produit pas un dictionnaire de type **[dict]** mais de type **[OrderedDict]** (ligne 1) ;
- lignes 24-25 : on transforme le type **[OrderedDict]** obtenu en type **[dict]** à l'aide de la méthode (non encore écrite) **[ordereddict2dict]**. Cette méthode travaille de façon récursive. Si certaines valeurs du dictionnaire sont de type **[OrderedDict, list]**, les valeurs de ces collections sont examinées pour savoir si elles-aussi sont de type **[OrderedDict]**. Si c'est le cas, elles sont transformées en type **[dict]**. On remarquera que la méthode **[xmltodict.parse]** ne produit aucun dictionnaire de type **[dict]** ;

Avant d'examiner les fonctions manquantes, examinons les résultats pour voir ce qui est cherché :

Le test 1 (lignes 28-29) produit les résultats suivants :

```
1.  test 1-------
2.  dictionnaire={'nom': 'séléné'}
3.  xml=<?xml version="1.0" encoding="utf-8"?>
4.  <nom>séléné</nom>
5.  ordereddict_dictionary1=OrderedDict([('nom', 'séléné')])
6.  dict_dictionary1={'nom': 'séléné'}
```

- ligne 2 : le dictionnaire testé. Il faut noter un point important : la méthode **[xml2dict.unparse]** exisge que le dictionnaire soit de la forme {'clé' : valeur} où **[valeur]** peut être ensuite un dictionnaire, une liste, un type simple ;
- lignes 3-4 : la chaîne XML issue du dictionnaire. Elle est précédée de l'entête **[<?xml version="1.0" encoding="utf-8"?>\n]** qui est normalement la 1ère ligne d'un fichier XML ;
- ligne 5 : le type **[OrderedDict]** obtenu par la méthode **[xml2dict.parse]** recevant pour paramètre la chaîne XML précédente ;
- ligne 6 : le dictionnaire de type **[dict]** obtenu en appliquant la méthode **[ordereddict2dict]** au type précédent. On retrouve le dictionnaire d'origine de la ligne 2 ;

Tous les autres tests sont construits sur le même schéma et devraient vous permettre de comprendre comment passer d'un dictionnaire à une chaîne XML puis de cette chaîne XML au dictionnaire d'origine.

Les autres tests donnent les résultats suivants :

```
1.  test 2-------
2.  dictionnaire={'famille': {'père': {'prénom': 'andré'}, 'mère': {'prénom': 'angèle'}, 'nom': 'séléné'}}
3.  xml=<?xml version="1.0" encoding="utf-8"?>
4.  <famille><père><prénom>andré</prénom></père><mère><prénom>angèle</prénom></mère><nom>séléné</nom></famille>
5.  ordereddict_dictionary1=OrderedDict([('famille', OrderedDict([('père', OrderedDict([('prénom', 'andré')])),
    ('mère', OrderedDict([('prénom', 'angèle')])), ('nom', 'séléné')]))])
6.  dict_dictionary1={'famille': {'père': {'prénom': 'andré'}, 'mère': {'prénom': 'angèle'}, 'nom': 'séléné'}}
7.
8.  test 3-------
9.  dictionnaire={'famille': {'nom': 'séléné', 'père': {'prénom': 'andré'}, 'mère': {'prénom': 'angèle'},
    'hobbies': ['chant', 'footing']}}
10. xml=<?xml version="1.0" encoding="utf-8"?>
11. <famille><nom>séléné</nom><père><prénom>andré</prénom></père><mère><prénom>angèle</prénom></
    mère><hobbies>chant</hobbies><hobbies>footing</hobbies></famille>
12. ordereddict_dictionary1=OrderedDict([('famille', OrderedDict([('nom', 'séléné'), ('père',
    OrderedDict([('prénom', 'andré')])), ('mère', OrderedDict([('prénom', 'angèle')])), ('hobbies', ['chant',
    'footing'])]))])
13. dict_dictionary1={'famille': {'nom': 'séléné', 'père': {'prénom': 'andré'}, 'mère': {'prénom': 'angèle'},
    'hobbies': ['chant', 'footing']}}
14.
15. test 4-------
```

```
16. dictionnaire={'réponse': {'erreurs': ['Méthode GET requise avec les seuls paramètres [marié, enfants,
    salaire]', 'paramètre [marié] manquant', 'paramètre [enfants] manquant', 'paramètre [salaire] manquant']}}
17. xml=<?xml version="1.0" encoding="utf-8"?>
18. <réponse><erreurs>Méthode GET requise avec les seuls paramètres [marié, enfants,
    salaire]</erreurs><erreurs>paramètre [marié] manquant</erreurs><erreurs>paramètre [enfants]
    manquant</erreurs><erreurs>paramètre [salaire] manquant</erreurs></réponse>
19. ordereddict_dictionary1=OrderedDict([('réponse', OrderedDict([('erreurs', ['Méthode GET requise avec les
    seuls paramètres [marié, enfants, salaire]', 'paramètre [marié] manquant', 'paramètre [enfants] manquant',
    'paramètre [salaire] manquant'])]))])
20. dict_dictionary1={'réponse': {'erreurs': ['Méthode GET requise avec les seuls paramètres [marié, enfants,
    salaire]', 'paramètre [marié] manquant', 'paramètre [enfants] manquant', 'paramètre [salaire] manquant']}}
21.
22. test 5-------
23. dictionnaire={'réponse': {'result': {'id': 0, 'marié': 'oui', 'enfants': 2, 'salaire': 50000, 'impôt':
    1384, 'décôte': 384, 'surcôte': 0, 'réduction': 347, 'taux': 0.14}}}
24. xml=<?xml version="1.0" encoding="utf-8"?>
25. <réponse><result><id>0</id><marié>oui</marié><enfants>2</enfants><salaire>50000</salaire><impôt>1384</
    impôt><décôte>384</décôte><surcôte>0</surcôte><réduction>347</réduction><taux>0.14</taux></result></
    réponse>
26. ordereddict_dictionary1=OrderedDict([('réponse', OrderedDict([('result', OrderedDict([('id', '0'),
    ('marié', 'oui'), ('enfants', '2'), ('salaire', '50000'), ('impôt', '1384'), ('décôte', '384'), ('surcôte',
    '0'), ('réduction', '347'), ('taux', '0.14')]))]))])
27. dict_dictionary1={'réponse': {'result': {'id': '0', 'marié': 'oui', 'enfants': '2', 'salaire': '50000',
    'impôt': '1384', 'décôte': '384', 'surcôte': '0', 'réduction': '347', 'taux': '0.14'}}}
28.
29. test 6-------
30. dictionnaire={'root': {'liste': ['un', 'deux', 'trois']}}
31. xml=<?xml version="1.0" encoding="utf-8"?>
32. <root><liste>un</liste><liste>deux</liste><liste>trois</liste></root>
33. ordereddict_dictionary1=OrderedDict([('root', OrderedDict([('liste', ['un', 'deux', 'trois'])]))])
34. dict_dictionary1={'root': {'liste': ['un', 'deux', 'trois']}}
35.
36. test 7-------
37. dictionnaire={'root': {'liste': [{'un': [10, 11]}, {'deux': [20, 21]}, {'trois': [30, 31]}]}}
38. xml=<?xml version="1.0" encoding="utf-8"?>
39. <root><liste><un>10</un><un>11</un></liste><liste><deux>20</deux><deux>21</deux></liste><liste><trois>30</
    trois><trois>31</trois></liste></root>
40. ordereddict_dictionary1=OrderedDict([('root', OrderedDict([('liste', [OrderedDict([('un', ['10', '11'])]),
    OrderedDict([('deux', ['20', '21'])]), OrderedDict([('trois', ['30', '31'])])])]))])
41. dict_dictionary1={'root': {'liste': [{'un': ['10', '11']}, {'deux': ['20', '21']}, {'trois': ['30',
    '31']}]}}
42.
43. Process finished with exit code 0
```

- les lignes 23 et 27 montrent un point important :
  - ligne 23 : les valeurs associées aux clés du dictionnaire **[result]** sont des nombres ;
  - ligne 26 : les valeurs associées aux clés du dictionnaire **[ordereddict_dictionary1]** sont des chaînes de caractères. C'est une faiblesse de la bibliothèque **[xmltodict]**. Sa méthode **[parse]** ne produit que des chaînes de caractères. Cela peut se comprendre aisément :
    - ligne 25 : la chaîne XML à partir de laquelle est produit le dictionnaire. Dans cette chaîne, il n'y a aucune indication du type des données encapsulées dans les balises XML. **[xmltodict.parse]** fait ce qu'il y a de plus logique : elle laisse tout en chaîne de caractères dans le dictionnaire produit. On trouve d'autres bibliothèques similaires à **[xmltodict]** où le type des données encapsulées est indiqué dans les balises. On pourrait trouver par exemple la balise **[<enfants type='int'>2</enfants>]** ;
    - la conséquence de ceci est que lorsqu'on exploite un dictionnaire produit par le module **[xmltodict]** on doit connaître le type des données qu'il encapsule pour pouvoir passer du type 'str' au type réel de la donnée ;

Attardons-nous maintenant sur la méthode **[ordereddict2dict]** qui transforme un type **[OrderedDict]** en type **[dict]** :

```
1.  # xmltodict.parse pour passer du XML au dictionnaire. Le dictionnaire doit avoir une racine
2.  # le dictionnaire produit est de type OrderedDict
3.  # xmltodict.unparse pour passer du dictionnaire au XML
4.
5.  def check(value):
6.      # si la valeur est de type OrderedDict, on la transforme
7.      if isinstance(value, OrderedDict):
8.          value2 = ordereddict2dict(value)
9.      # si la valeur est de type list, on la transforme
10.     elif isinstance(value, list):
11.         value2 = list2list(value)
12.     else:
13.         # on est en présence d'un type simple pas d'une collection
14.         value2 = value
15.     # on rend la nouvelle valeur
```

```
16.     return value2
17.
18.
19. def list2list(liste: list) -> list:
20.     # la nouvelle liste
21.     newliste = []
22.     # on exploite les éléments de la liste paramètre
23.     for value in liste:
24.         # on ajoute value à la nouvelle liste
25.         newliste.append(check(value))
26.     # on rend la nouvelle liste
27.     return newliste
28.
29.
30. def ordereddict2dict(ordered_dictionary: OrderedDict) -> dict:
31.     # OrderedDict -> dict de façon récursive
32.     newdict = {}
33.     for key, value in ordered_dictionary.items():
34.         # on mémorise la valeur dans le nouveau dictionnaire
35.         newdict[key] = check(value)
36.     # on rend le dictionnaire
37.     return newdict
```

- ligne 30 : la fonction **[ordereddict2dict]** reçoit un type **[OrderedDict]** comme paramètre ;
- ligne 32 : le dictionnaire de type **[dict]** qui sera rendu ligne 37 par la fonction ;
- ligne 33 : on explore tous les tuples (clé, valeur) du dictionnaire **[ordered_dictionary]** ;
- ligne 35 : dans le nouveau dictionnaire, la clé **[key]** est gardée mais la valeur associée n'est pas **[value]** mais **[check(value)]**. La fonction **[check(value)]** est chargée de trouver, si **[value]** est une collection, tous les éléments de type **[OrderedDict]** et de les transformer en type **[dict]** ;

La méthode **[check]** est définie aux lignes 5-16 :

- ligne 5 : on ne connaît pas le type de **[value]**, aussi n'a-t-on pu écrire **[value : type]** ;
- lignes 7-8 : si **[value]** est de type **[OrderedDict]** alors on appelle de façon récursive la fonction **[ordereddict2dict]** qu'on vient de commenter ;
- lignes 9-11 : un autre cas possible est que **[value]** soit une liste. Dans ce cas, ligne 11, on appelle la fonction **[list2list]** des 19-27 ;
- lignes 12-14 : le dernier cas est que **[value]** n'est pas une collection mais un type simple. La fonction **[check]**, comme les fonctions **[ordereddict2dict]** et **[list2list]** sont récursives. On sait qu'alors il faut toujours prévoir le cas où la récursion s'arrête. Les lignes 12-14 sont ce cas ;
- ligne 16 : la fonction **[check]** appelée récursivement ou pas produit une valeur **[valeur2]** qui doit remplacer le paramètre **[value]** de la ligne 5 ;

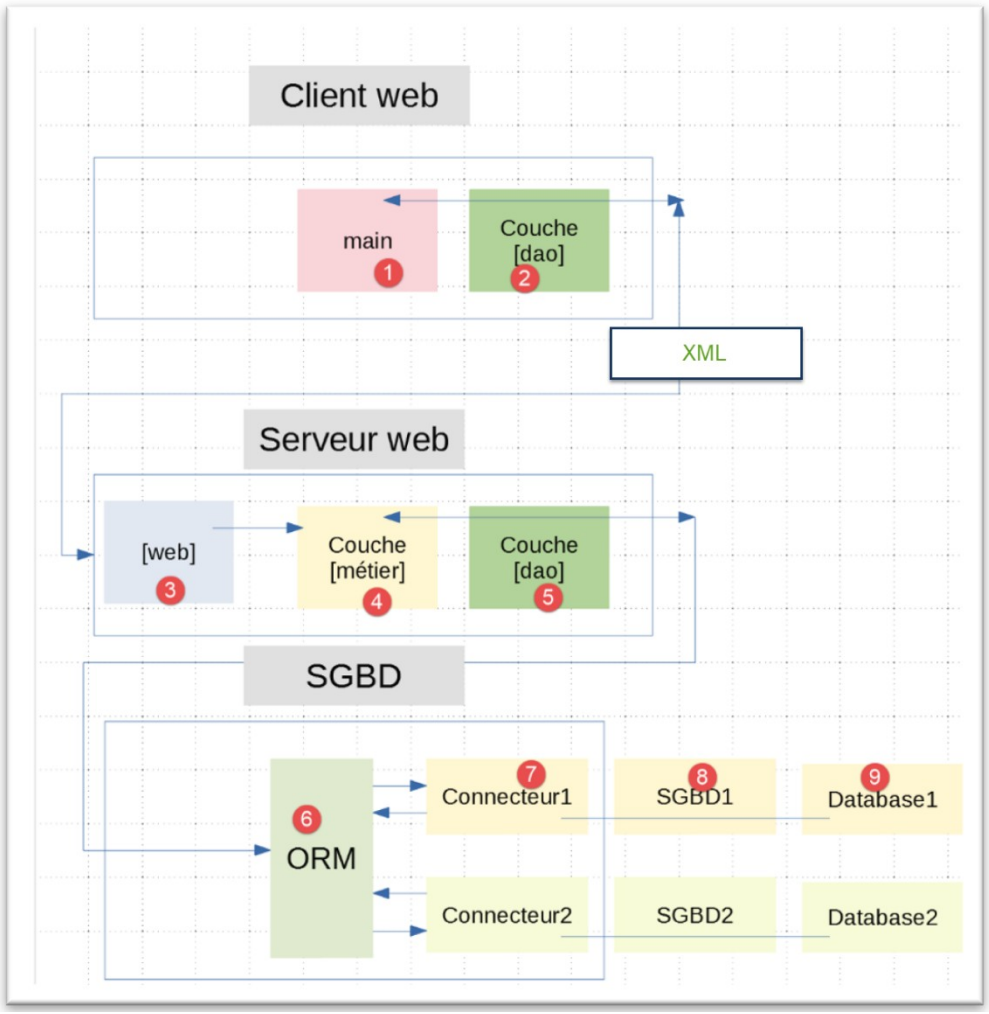
La méthode **[list2list]** définie aux lignes 19-27 exploite une liste passée en paramètre. Elle va l'explorer et remplacer toute valeur de type **[OrderedDict]** trouvée dedans en un type **[dict]**.

- ligne 21 : la nouvelle liste que va créer la fonction ;
- lignes 23-25 : toutes les valeurs **[value]** de la liste sont explorées et remplacées par la valeur **[check(value)]**. Cette valeur **[value]** peut elle-même contenir des éléments de type **[list]** ou **[OrderedDict]**. Ils seront traités correctement par la fonction récursive **[check]** ;

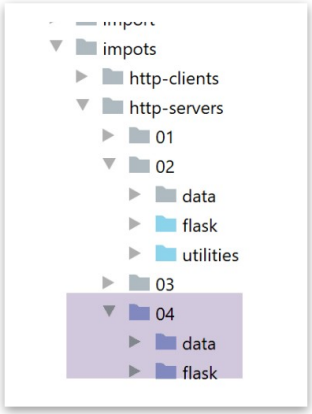
27 Exercice d'application : version 9

Nous revenons à la version 7 de l'exercice d'application et au lieu que le client et le serveur web s'échangent des chaînes jSON ils vont ici s'échanger du XML.

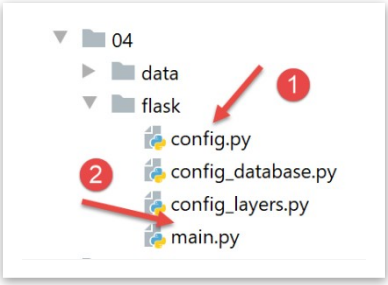
L'architecture reste la même :



27.1 Le serveur web



Le dossier **[http-servers/04]** est obtenue par recopie du dossier **[http-servers/02]** à l'exception du sous-dossier **[utilities]**. Puis on change les éléments suivants :



Le fichier **[config]** est modifié de la façon suivante :

```
1.      # dépendances absolues
2.      absolute_dependencies = [
3.          # dossiers du projet
4.          # BaseEntity, MyException
5.          f"{root_dir}/classes/02/entities",
6.          # InterfaceImpôtsDao, InterfaceImpôtsMétier, InterfaceImpôtsUi
7.          f"{root_dir}/impots/v04/interfaces",
8.          # AbstractImpôtsdao, ImpôtsConsole, ImpôtsMétier
9.          f"{root_dir}/impots/v04/services",
10.         # ImpotsDaoWithAdminDataInDatabase
11.         f"{root_dir}/impots/v05/services",
12.         # AdminData, ImpôtsError, TaxPayer
13.         f"{root_dir}/impots/v04/entities",
14.         # Constantes, tranches
15.         f"{root_dir}/impots/v05/entities",
16.         # index_controller
17.         f"{root_dir}/impots/http-servers/01/controllers",
18.         # scripts [config_database, config_layers]
19.         script_dir,
20.         # Logger, SendAdminMail
21.         f"{root_dir}/impots/http-servers/02/utilities",
22.     ]
```

- ligne 21 : on indique le dossier des utilitaires qui sont restés dans **[http-servers/02]** ;

Le script principal **[main]** évolue de la façon suivante :

```
1.  # Home URL
2.  @app.route('/', methods=['GET'])
3.  @auth.login_required
4.  def index():
5.      logger = None
6.      try:
7.          # logger
8.          logger = Logger(config["logsFilename"])
9.          # on le mémorise dans une config associée au thread
10.         thread_config = {"logger": logger}
11.         thread_name = threading.current_thread().name
12.         config[thread_name] = {"config": thread_config}
13.         # on logue la requête
14.         logger.write(f"[index] requête : {request}\n")
15.         …
16.         # on fait exécuter la requête par un contrôleur
17.         résultat, status_code = index_controller.execute(request, config)
18.         # y-a-t-il eu une erreur fatale ?
19.         …
20.         # on logue la réponse
21.         logger.write(f"[index] {résultat}\n")
22.         # on envoie la réponse
23.         return xml_response(résultat, status_code)
24.     except BaseException as erreur:
25.         # on logue l'erreur si c'est possible
26.         if logger:
27.             logger.write(f"[index] {erreur}")
28.         # on prépare la réponse au client
29.         résultat = {"réponse": {"erreurs": [f"{erreur}"]}}
30.         # on envoie la réponse
31.         return xml_response(résultat, status.HTTP_500_INTERNAL_SERVER_ERROR)
32.     finally:
```

```
33.         # on ferme le fichier de logs s'il a été ouvert
34.         if logger:
35.             logger.close()
```

- l'unique modification est ligne 23 : on envoie désormais une réponse XML ;

La fonction **[xml_response]** est définie dans le module **[myutils]** :

```
1.  import xmltodict
2.  …
3.  def xml_response(résultat: dict, status_code: int) -> tuple:
4.      # résultat : le dictionnaire à transformer en chaîne XML
5.      xmlString = xmltodict.unparse(résultat)
6.      # on rend la réponse HTTP
7.      response = make_response(xmlString)
8.      response.headers['Content-Type'] = 'application/xml; charset=utf-8'
9.      return response, status_code
```

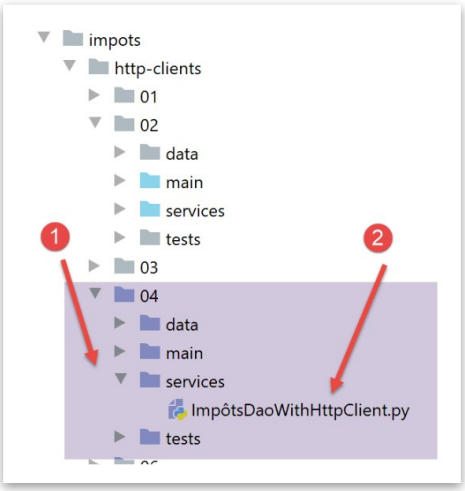
- ligne 3 : la fonction **[xml_response]** reçoit en paramètres :
  - le dictionnaire **[résultat]** à transformer en XML ;
  - le code de statut **[status_code]** à renvoyer au client web ;
- ligne 5 : on utilise la bibliothèque **[xmltodict]** pour produire la chaîne XML ;
- ligne 8 : on utilise l'entête **[Content-Type]** pour dire au client qu'on lui envoie du XML ;

La fonction **[xml_response]** doit être importée dans le script **[__init__.py]** :

```
from .myutils import set_syspath, json_response, decode_flask_session, xml_response
```

Puis le module **[myutils]** doit être incorporé dans les modules de portée machine. Cela se fait dans un terminal PyCharm avec la commande **[pip install .]** (dans le dossier packages).

27.2 Le client web



27.2.1 Le code

Le dossier **[http-clients/04]** est obtenu par recopie du client **[http-clients/02]**. Puis nous modifions la classe **[ImpôtsDaoWithHttpClient]** de la façon suivante :

```
1.  # imports
2.
3.  import requests
4.  import xmltodict
5.  from flask_api import status
6.
7.  from AbstractImpôtsDao import AbstractImpôtsDao
8.  from AdminData import AdminData
9.  from ImpôtsError import ImpôtsError
10. from InterfaceImpôtsMétier import InterfaceImpôtsMétier
11. from TaxPayer import TaxPayer
12.
13.
```

```
14. class ImpôtsDaoWithHttpClient(AbstractImpôtsDao, InterfaceImpôtsMétier):
15.
16.     # constructeur
17.     def __init__(self, config: dict):
18.         …
19.
20.     # méthode inutilisée
21.     def get_admindata(self) -> AdminData:
22.         pass
23.
24.     # calcul de l'impôt
25.     def calculate_tax(self, taxpayer: TaxPayer, admindata: AdminData = None):
26.         ….
27.         # code de statut de la réponse HTTP
28.         status_code = response.status_code
29.         # on met la réponse XML dans un dictionnaire
30.         résultat = xmltodict.parse(response.text[39:])
31.         # erreur si code de statut différent de 200 OK
32.         ….
```

- ligne 30 : la réponse HTTP **[response]** du serveur web est désormais une chaîne XML. Les logs montrent la nature de celle-ci :

```
2020-07-27 15:53:47.886283, Thread-2 : <?xml version="1.0" encoding="utf-8"?>
<réponse><result><marié>non</marié><enfants>2</enfants><salaire>100000</salaire><impôt>19884</
impôt><surcôte>4480</surcôte><taux>0.41</taux><décôte>0</décôte><réduction>0</réduction></result></
réponse>
```

  La chaîne **[<?xml version="1.0" encoding="utf-8"?>]** compte 38 caractères. Par ailleurs, si on regarde le fichier de logs avec un éditeur hexadécimal, on voit que derrière cette chaîne il y a un caractère de saut de ligne \n. Puis vient la réponse <réponse>…</réponse>. La chaîne XML que nous devons convertir commence donc après les 39 premiers caractères de la chaîne XML. Elle commence à partir du caractère n° 39, le 1er caractère étant numéroté 0. Cette chaîne est obtenue par l'expression **[response.text[39:]]**.

Si nous exécutons le client (suivez la procédure des exemples précédents), nous obtenons les mêmes résultats dans le fichier **[résultats.json]** que dans les versions précédentes. Les logs sont eux les suivants :

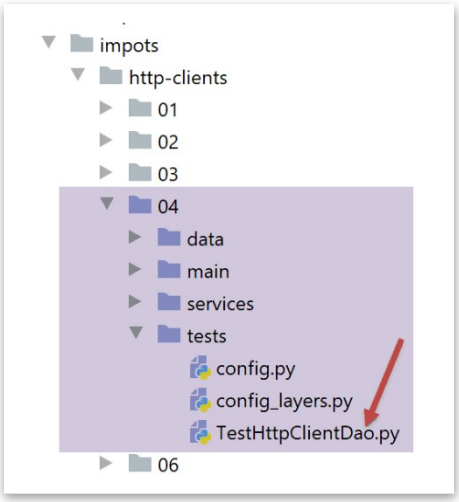
```
1.  2020-07-27 16:21:14.015941, Thread-1 : début du thread [Thread-1] avec 2 contribuable(s)
2.  2020-07-27 16:21:14.016940, Thread-1 : début du calcul de l'impôt de {"id": 1, "marié": "oui", "enfants":
    2, "salaire": 55555}
3.  2020-07-27 16:21:14.016940, Thread-2 : début du thread [Thread-2] avec 3 contribuable(s)
4.  2020-07-27 16:21:14.018939, Thread-2 : début du calcul de l'impôt de {"id": 3, "marié": "oui", "enfants":
    3, "salaire": 50000}
5.  2020-07-27 16:21:14.019979, Thread-3 : début du thread [Thread-3] avec 3 contribuable(s)
6.  2020-07-27 16:21:14.019979, Thread-3 : début du calcul de l'impôt de {"id": 6, "marié": "oui", "enfants":
    3, "salaire": 100000}
7.  2020-07-27 16:21:14.021938, Thread-4 : début du thread [Thread-4] avec 2 contribuable(s)
8.  2020-07-27 16:21:14.021938, Thread-4 : début du calcul de l'impôt de {"id": 9, "marié": "oui", "enfants":
    2, "salaire": 30000}
9.  2020-07-27 16:21:14.021938, Thread-5 : début du thread [Thread-5] avec 1 contribuable(s)
10. 2020-07-27 16:21:14.022939, Thread-5 : début du calcul de l'impôt de {"id": 11, "marié": "oui", "enfants":
    3, "salaire": 200000}
11. 2020-07-27 16:21:14.031942, Thread-1 : <?xml version="1.0" encoding="utf-8"?>
12. <réponse><result><marié>oui</marié><enfants>2</enfants><salaire>55555</salaire><impôt>2814</
    impôt><surcôte>0</surcôte><taux>0.14</taux><décôte>0</décôte><réduction>0</réduction></result></réponse>
13. 2020-07-27 16:21:14.031942, Thread-1 : fin du calcul de l'impôt de {"id": 1, "marié": "oui", "enfants": 2,
    "salaire": 55555, "impôt": 2814, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0}
14. 2020-07-27 16:21:14.031942, Thread-1 : début du calcul de l'impôt de {"id": 2, "marié": "oui", "enfants":
    2, "salaire": 50000}
15. 2020-07-27 16:21:14.034941, Thread-4 : <?xml version="1.0" encoding="utf-8"?>
16. <réponse><result><marié>oui</marié><enfants>2</enfants><salaire>30000</salaire><impôt>0</
    impôt><surcôte>0</surcôte><taux>0.0</taux><décôte>0</décôte><réduction>0</réduction></result></réponse>
17. …
18. 2020-07-27 16:21:17.055931, Thread-3 : fin du thread [Thread-3]
19. 2020-07-27 16:21:17.059930, Thread-2 : <?xml version="1.0" encoding="utf-8"?>
20. <réponse><result><marié>non</marié><enfants>3</enfants><salaire>100000</salaire><impôt>16782</
    impôt><surcôte>7176</surcôte><taux>0.41</taux><décôte>0</décôte><réduction>0</réduction></result></réponse>
21. 2020-07-27 16:21:17.060971, Thread-2 : fin du calcul de l'impôt de {"id": 5, "marié": "non", "enfants": 3,
    "salaire": 100000, "impôt": 16782, "surcôte": 7176, "taux": 0.41, "décôte": 0, "réduction": 0}
22. 2020-07-27 16:21:17.060971, Thread-2 : fin du thread [Thread-2]
```

Côté serveur, les logs sont les suivants :

```
1.  2020-07-27 16:32:04.983020, Thread-46 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=2&salaire=50000' [GET]>
2.  2020-07-27 16:32:04.983020, Thread-46 : [index] mis en pause du thread pendant 1 seconde(s)
3.  2020-07-27 16:32:04.984021, Thread-47 : [index] requête : <Request 'http://127.0.0.1:5000/?
    marié=oui&enfants=2&salaire=55555' [GET]>
4.  2020-07-27 16:32:04.984021, Thread-47 : [index] mis en pause du thread pendant 1 seconde(s)
5.  …
6.  2020-07-27 16:32:07.001271, Thread-56 : [index] mis en pause du thread pendant 1 seconde(s)
7.  2020-07-27 16:32:07.003078, Thread-54 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 5,
    'salaire': 100000, 'impôt': 4230, 'surcôte': 0, 'taux': 0.14, 'décôte': 0, 'réduction': 0}}}
8.  2020-07-27 16:32:07.006078, Thread-55 : [index] {'réponse': {'result': {'marié': 'oui', 'enfants': 3,
    'salaire': 200000, 'impôt': 42842, 'surcôte': 17283, 'taux': 0.41, 'décôte': 0, 'réduction': 0}}}
9.  2020-07-27 16:32:08.002824, Thread-56 : [index] {'réponse': {'result': {'marié': 'non', 'enfants': 2,
    'salaire': 100000, 'impôt': 19884, 'surcôte': 4480, 'taux': 0.41, 'décôte': 0, 'réduction': 0}}}
```

- le logueur continue à écrire le dictionnaire de la réponse et non la chaîne XML qui est envoyée au client. Ce n'est pas une erreur et c'est voulu ;

27.2.2 Test de la couche `[dao]` du client



La classe de test **`[TestHttpClientDao]`** est la même que dans la |version 7| et donne les mêmes résultats.

28 Exercice d'application : version 10

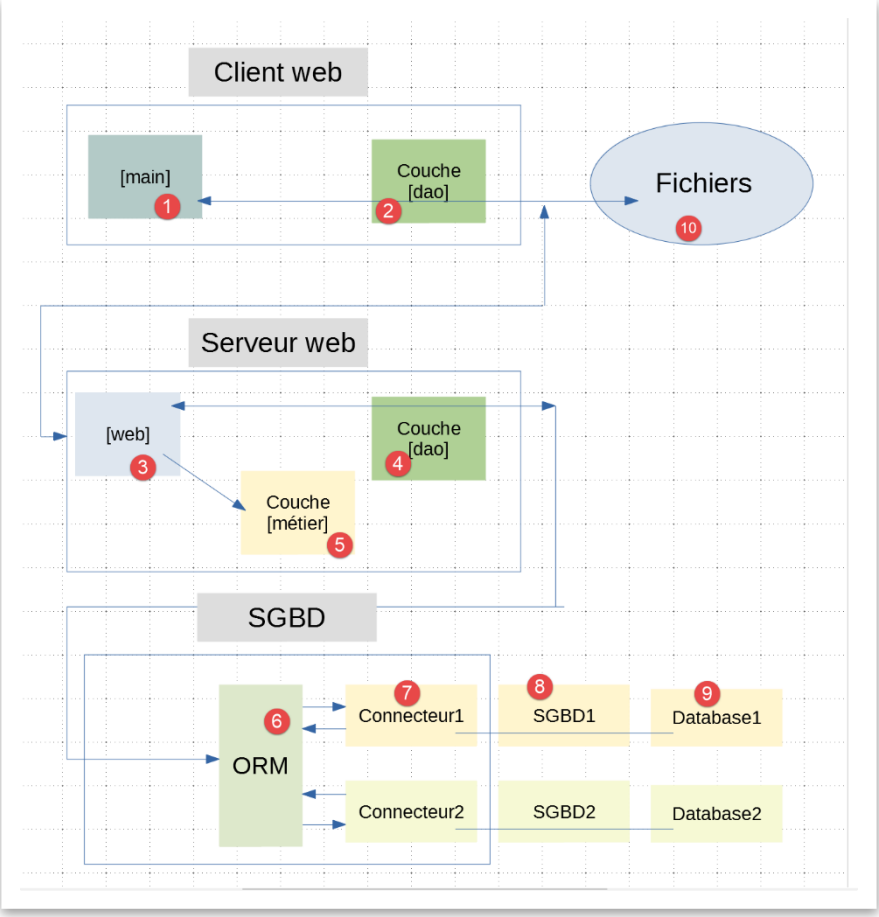
28.1 Introduction

Dans les exemples de clients du serveur de calcul de l'impôt, les threads envoyaient N requêtes séquentiellement si elles devaient traiter N contribuables. L'idée ici est d'envoyer une seule requête encapsulant les N contribuables. Pour chacun d'entre-eux, il faut envoyer les informations **`[marié, enfants, salaire]`**. On peut les envoyer comme paramètres :

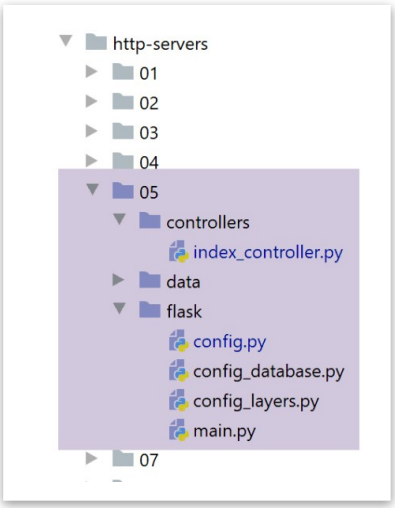
- de l'URL. On aura alors une longue URL peu significative ;
- dans le corps (body) de la requête HTTP. On sait que ce corps est caché à l'utilisateur utilisant un navigateur ;

Dans les deux cas, on peut utiliser une requête **`[GET]`** ou **`[POST]`**. Nous utiliserons une requête POST avec les paramètres encapsulés dans le corps de la requête HTTP.

L'architecture client / serveur n'a pas changé :



28.2 Le serveur web



Le dossier **[http-servers/05]** est obtenu initialement par recopie du dossier **[http-servers/02]**. On revient aux échanges jSON entre le client et le serveur. On a vu que passer du jSON au XML est très simple.

28.2.1 Configuration

La configuration **[config, config_database, config_layers]** reste analogue à celle des versions précédentes. Nous ne revenons pas dessus.

28.2.2 Le script principal [main]

Le script **[main]** est identique à celui du dossier **[http-servers/02]** qu'on a recopié. Une seule chose diffère :

```
1.  # Home URL
2.  @app.route('/', methods=['POST'])
3.  @auth.login_required
4.  def index():
5.      …
```

- ligne 2 : désormais l'URL / s'obtient via un POST ;

28.2.3 Le contrôleur [index_controller]

Le contrôleur **[index_controller]** évolue de la façon suivante :

```
1.  # import des dépendances
2.
3.  import json
4.
5.  from flask_api import status
6.  from werkzeug.local import LocalProxy
7.
8.
9.  def execute(request: LocalProxy, config: dict) -> tuple:
10.     # dépendances
11.     from ImpôtsError import ImpôtsError
12.     from TaxPayer import TaxPayer
13.
14.     # on récupère le corps du post - on attend une liste de dictionnaires
15.     msg_erreur = None
16.     list_dict_taxpayers = None
17.     # le corps jSON du POST
18.     request_text = request.data
19.     try:
20.         # qu'on transforme en une liste de dictionnaires
21.         list_dict_taxpayers = json.loads(request_text)
22.     except BaseException as erreur:
23.         # on note l'erreur
24.         msg_erreur = f"le corps du POST n'est pas une chaîne jSON valide : {erreur}"
25.     # a-t-on une liste non vide ?
26.     if not msg_erreur and (not isinstance(list_dict_taxpayers, list) or len(list_dict_taxpayers) == 0):
27.         # on note l'erreur
```

```
28.         msg_erreur = "le corps du POST n'est pas une liste ou alors cette liste est vide"
29.     # a-t-on une liste de dictionnaires ?
30.     if not msg_erreur:
31.         erreur = False
32.         i = 0
33.         while not erreur and i < len(list_dict_taxpayers):
34.             erreur = not isinstance(list_dict_taxpayers[i], dict)
35.             i += 1
36.         # erreur ?
37.         if erreur:
38.             msg_erreur = "le corps du POST doit être une liste de dictionnaires"
39.     # erreur ?
40.     if msg_erreur:
41.         # on envoie une réponse d'erreur au client
42.         résultats = {"réponse": {"erreurs": [msg_erreur]}}
43.         return résultats, status.HTTP_400_BAD_REQUEST
44.
45.     # on vérifie les TaxPayers un par un
46.     # au départ pas d'erreurs
47.     list_erreurs = []
48.     for dict_taxpayer in list_dict_taxpayers:
49.         # on crée un TaxPayer à partir de dict_taxpayer
50.         msg_erreur = None
51.         try:
52.             # l'opération suivante va éliminer les cas où les paramètres ne sont pas
53.             # des propriétés de la classe TaxPayer ainsi que les cas où leurs valeurs
54.             # sont incorrectes
55.             TaxPayer().fromdict(dict_taxpayer)
56.         except BaseException as erreur:
57.             msg_erreur = f"{erreur}"
58.         # certaines clés doivent être présentes dans le dictionnaire
59.         if not msg_erreur:
60.             # les clés [marié, enfants, salaire] doivent être présentes dans le dictionnaire
61.             keys = dict_taxpayer.keys()
62.             if 'marié' not in keys or 'enfants' not in keys or 'salaire' not in keys:
63.                 msg_erreur = "le dictionnaire doit inclure les clés [marié, enfants, salaire]"
64.         # des erreurs ?
65.         if msg_erreur:
66.             # on note l'erreur dans le TaxPayer lui-même
67.             dict_taxpayer['erreur'] = msg_erreur
68.             # on ajoute le TaxPayer à la liste des erreurs
69.             list_erreurs.append(dict_taxpayer)
70.
71.     # on a traité tous les taxpayers - y-a-t-il des erreurs ?
72.     if list_erreurs:
73.         # on envoie une réponse d'erreur au client
74.         résultats = {"réponse": {"erreurs": list_erreurs}}
75.         return résultats, status.HTTP_400_BAD_REQUEST
76.
77.     # pas d'erreurs, on peut travailler
78.     # récupération des données de l'administration fiscale
79.     admindata = config["admindata"]
80.     métier = config["layers"]["métier"]
81.     try:
82.         # on traite les TaxPayer un à un
83.         list_taxpayers = []
84.         for dict_taxpayer in list_dict_taxpayers:
85.             # calcul de l'impôt
86.             taxpayer = TaxPayer().fromdict(
87.                 {'marié': dict_taxpayer['marié'], 'enfants': dict_taxpayer['enfants'],
88.                  'salaire': dict_taxpayer['salaire']})
89.             métier.calculate_tax(taxpayer, admindata)
90.             # on mémorise le résultat en tant que dictionnaire
91.             list_taxpayers.append(taxpayer.asdict())
92.         # on envoie la réponse au client
93.         return {"réponse": {"results": list_taxpayers}}, status.HTTP_200_OK
94.     except ImpôtsError as erreur:
95.         # on envoie une réponse d'erreur au client
96.         return {"réponse": {"erreurs": f"[{erreur}]"}}, status.HTTP_500_INTERNAL_SERVER_ERROR
```

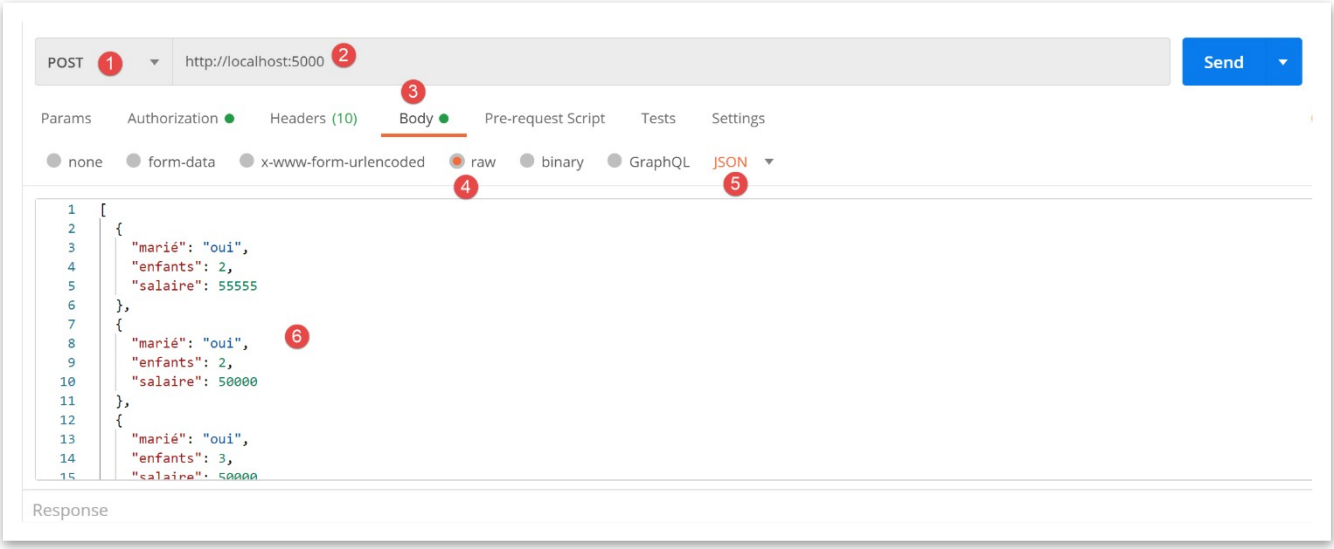
- ligne 9 : le contrôleur reçoit :
  - la requête **[request]** du client ;
  - la configuration **[config]** du serveur ;
- lignes 14-18 : on récupère le corps du POST. Les paramètres encapsulés dans le corps de la requête HTTP peuvent être encodés de différentes façons. Nous en avons déjà rencontré une : **[x-www-form-urlencoded]**. Nous allons ici utiliser un autre encodage : jSON ;

- ligne 18 : **[request.data]** permet de récupérer le corps (body) de la requête HTTP. On récupère ici du texte et nous savons que ce texte est du jSON qui représente une liste de dictionnaire **[marié, enfants, salaire]** ;
- lignes 19-24 : on récupère cette liste de dictionnaires ;
- lignes 22-24 : si la récupération du jSON s'est mal passée, on note l'erreur ;
- lignes 26-28 : si on découvre que l'objet récupéré n'est pas une liste ou que c'est une liste vode, on note l'erreur ;
- lignes 29-38 : si on a bien récupéré une liste, on vérifie que c'est bien une liste de dictionnaires ;
- lignes 40-43 : s'il y a eu erreur, on s'arrête là et on envoie une réponse d'erreur au client ;
- lignes 45-69 : on vérifie maintenant chacun des dictionnaires :
  - ils doivent contenir les clés **[marié, enfants, salaire]** ;
  - ils doivent permettre de construire un objet **[TaxPayer]** valide ;
- lignes 65-69 : si une erreur a été détectée dans un dictionnaire, alors elle est mise dans ce même dictionnaire associée à la clé 'erreur' ;
- lignes 72-75 : les dictionnaires erronés ont été cumulés dans la liste **[list_erreurs]**. Si cette liste n'est pas vide, alors on l'envoie dans une réponse d'erreur faite au client ;
- ligne 77 : arrivé là, on sait qu'on peut créer une liste d'objets de type **[TaxPayer]** à partir du corps de la requête envoyée par le client ;
- lignes 84-91 : on exploite la liste des dictionnaires reçus ;
- ligne 86 : à partir d'un dictionnaire, on crée un objet **[TaxPayer]** ;
- ligne 89 : on calcule l'impôt de ce **[TaxPayer]** ;
- ligne 91 : on sait que **[taxpayer]** a été modifié par le calcul de l'impôt. On le transforme en dictionnaire et on l'ajoute à une liste de résultats ;
- ligne 93 : on envoie cette liste de résultats au client ;

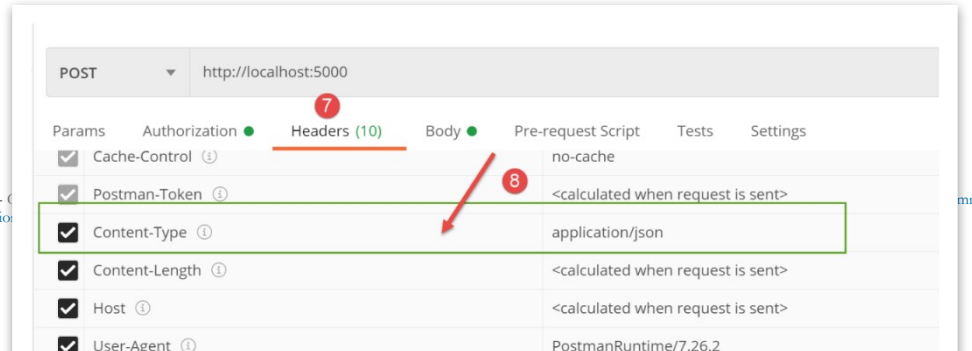
28.2.4 Tests du serveur

Nous allons tester le serveur avec un client Postman :

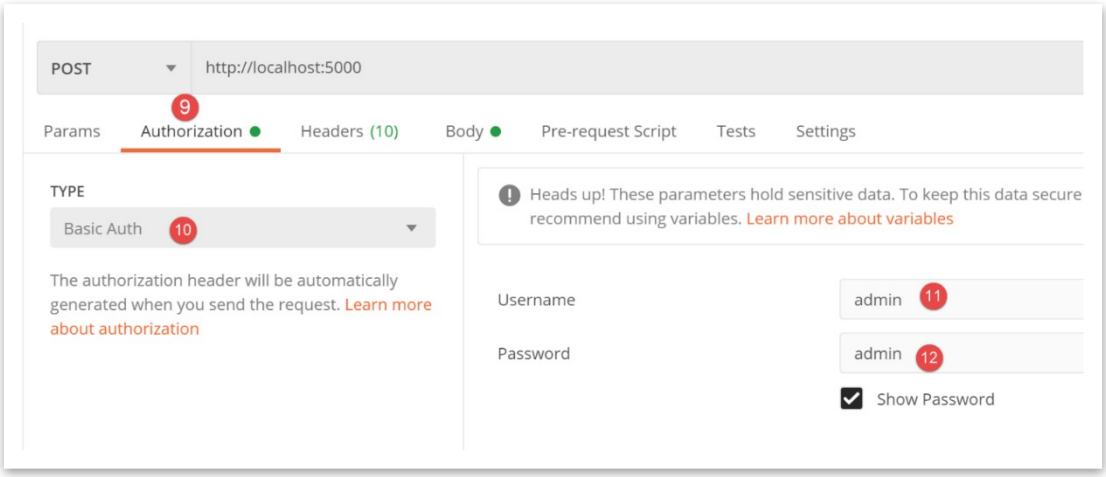
- nous lançons le serveur web, le SGBD, le serveur de mails **[hMailServer]** ;
- nous lançons le client Postman ainsi que sa console (Ctrl-Alt-C) ;



- en **[1]** : on émet une requête **[POST]** ;
- en **[2]** : l'URL du serveur ;
- en **[3]** : le corps de la requête HTTP ;
- en **[5]** : on indique que ce corps devra être envoyé sous la forme d'une chaîne jSON ;
- en **[4]** : on se met en mode **[raw]** pour pouvoir copier / coller une chaîne jSON ;
- en **[6]** : on colle la chaîne jSON prise dans un des fichiers **[résultats.json]** des différentes versions. Puis on ne garde pour chaque contribuable que les propriétés **[marié, salaire, enfants]** ;

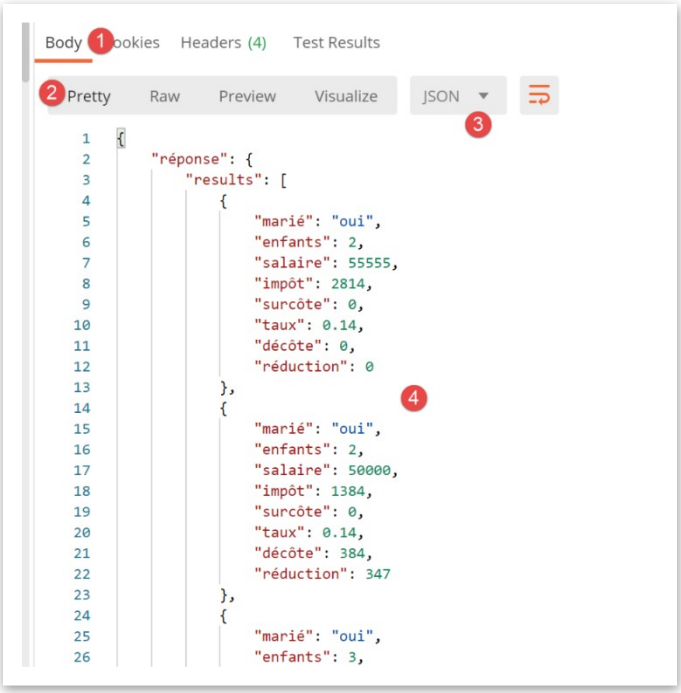


- en **[7]**, on regarde les entêtes HTTP que va envoyer le client Postman au serveur ;
- en **[8]**, on voit qu'il va lui envoyer un entête **[Content-Type]** lui indiquant que la requête contient un corps codé en jSON. Cela vient du choix **[5]** fait précédemment ;



- en **[9-12]** : on met dans la requête les identifiants attendus par le serveur ;

On envoie cette requête. La réponse du serveur est la suivante :



- en **[3]**, on a reçu du jSON ;
- en **[4]**, l'impôt des contribuables ;

Examinons dans la console Postman (Ctrl-Alt-C) le dialogue client / serveur qui a eu lieu :

Le client Postman a envoyé le texte suivant :

```
1.  POST / HTTP/1.1
2.  Authorization: Basic YWRtaW46YWRtaW4=
3.  Content-Type: application/json
4.  User-Agent: PostmanRuntime/7.26.2
5.  Accept: */*
6.  Cache-Control: no-cache
7.  Postman-Token: 03c4aa28-5a5d-4bb5-ac51-7ad51968c71d
8.  Host: localhost:5000
```

```
9.  Accept-Encoding: gzip, deflate, br
10. Connection: keep-alive
11. Content-Length: 824
12.
13. [
14.   {
15.     "marié": "oui",
16.     "enfants": 2,
17.     "salaire": 55555
18.   },
19.   {
20.     "marié": "oui",
21.     "enfants": 2,
22.     "salaire": 50000
23.   },
24.   {
25.     "marié": "oui",
26.     "enfants": 3,
27.     "salaire": 50000
28.   },
29.   {
30.     "marié": "non",
31.     "enfants": 2,
32.     "salaire": 100000
33.   },
34.   {
35.     "marié": "non",
36.     "enfants": 3,
37.     "salaire": 100000
38.   },
39.   {
40.     "marié": "oui",
41.     "enfants": 3,
42.     "salaire": 100000
43.   },
44.   {
45.     "marié": "oui",
46.     "enfants": 5,
47.     "salaire": 100000
48.   },
49.   {
50.     "marié": "non",
51.     "enfants": 0,
52.     "salaire": 100000
53.   },
54.   {
55.     "marié": "oui",
56.     "enfants": 2,
57.     "salaire": 30000
58.   },
59.   {
60.     "marié": "non",
61.     "enfants": 0,
62.     "salaire": 200000
63.   },
64.   {
65.     "marié": "oui",
66.     "enfants": 3,
67.     "salaire": 200000
68.   }
69. ]
```

- ligne 1 : le POST vers le serveur ;
- ligne 2 : l'entête HTTP d'authentification ;
- ligne 3 : le client dit au serveur qu'il lui envoie une chaîne jSON et que cette chaîne fait 824 octets (ligne 11) ;
- lignes 13-69 : le corps jSON de la requête ;

Le serveur lui a répondu le texte suivant :

```
1.  HTTP/1.0 200 OK
2.  Content-Type: application/json; charset=utf-8
3.  Content-Length: 1461
4.  Server: Werkzeug/1.0.1 Python/3.8.1
5.  Date: Tue, 28 Jul 2020 07:16:34 GMT
6.
```

```
7.  {"réponse": {"results": [{"marié": "oui", "enfants": 2, "salaire": 55555, "impôt": 2814, "surcôte": 0,
    "taux": 0.14, "décôte": 0, "réduction": 0}, {"marié": "oui", "enfants": 2, "salaire": 50000, "impôt": 1384,
    "surcôte": 0, "taux": 0.14, "décôte": 384, "réduction": 347}, {"marié": "oui", "enfants": 3, "salaire":
    50000, "impôt": 0, "surcôte": 0, "taux": 0.14, "décôte": 720, "réduction": 0}, {"marié": "non", "enfants":
    2, "salaire": 100000, "impôt": 19884, "surcôte": 4480, "taux": 0.41, "décôte": 0, "réduction": 0},
    {"marié": "non", "enfants": 3, "salaire": 100000, "impôt": 16782, "surcôte": 7176, "taux": 0.41, "décôte":
    0, "réduction": 0}, {"marié": "oui", "enfants": 3, "salaire": 100000, "impôt": 9200, "surcôte": 2180,
    "taux": 0.3, "décôte": 0, "réduction": 0}, {"marié": "oui", "enfants": 5, "salaire": 100000, "impôt": 4230,
    "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0}, {"marié": "non", "enfants": 0, "salaire": 100000,
    "impôt": 22986, "surcôte": 0, "taux": 0.41, "décôte": 0, "réduction": 0}, {"marié": "oui", "enfants": 2,
    "salaire": 30000, "impôt": 0, "surcôte": 0, "taux": 0.0, "décôte": 0, "réduction": 0}, {"marié": "non",
    "enfants": 0, "salaire": 200000, "impôt": 64210, "surcôte": 7498, "taux": 0.45, "décôte": 0, "réduction":
    0}, {"marié": "oui", "enfants": 3, "salaire": 200000, "impôt": 42842, "surcôte": 17283, "taux": 0.41,
    "décôte": 0, "réduction": 0}]}}
```

- ligne 1 : la requête a réussi ;
- ligne 2 : le corps de la réponse du serveur est une chaîne jSON. Celle-ci fait 1461 octets (ligne 3) ;
- ligne 7 : la réponse jSON du serveur ;

Testons maintenant des cas d'erreur.

Cas 1 : on envoie n'importe quoi

```
1.  POST / HTTP/1.1
2.  Authorization: Basic YWRtaW46YWRtaW4=
3.  Content-Type: application/json
4.  User-Agent: PostmanRuntime/7.26.2
5.  Accept: */*
6.  Cache-Control: no-cache
7.  Postman-Token: 47652706-9744-46a0-a682-de010e5406c0
8.  Host: localhost:5000
9.  Accept-Encoding: gzip, deflate, br
10. Connection: keep-alive
11. Content-Length: 3
12.
13. abc
14.
15. HTTP/1.0 400 BAD REQUEST
16. Content-Type: application/json; charset=utf-8
17. Content-Length: 125
18. Server: Werkzeug/1.0.1 Python/3.8.1
19. Date: Tue, 28 Jul 2020 07:43:27 GMT
20.
21. {"réponse": {"erreurs": ["le corps du POST n'est pas une chaîne jSON valide : Expecting value: line 1
    column 1 (char 0)"]}}
```

- ligne 13 : on a envoyé la chaîne **[abc]** qui n'est pas une chaîne jSON valide (ligne 3) ;
- ligne 15 : le serveur répond par un code d'erreur 400 ;
- ligne 21 : la réponse jSON du serveur ;

Cas 2 : envoyons une chaîne jSON valide qui ne soit pas une liste

```
1.  POST / HTTP/1.1
2.  Authorization: Basic YWRtaW46YWRtaW4=
3.  Content-Type: application/json
4.  User-Agent: PostmanRuntime/7.26.2
5.  Accept: */*
6.  Cache-Control: no-cache
7.  Postman-Token: 03b64735-9239-47b3-b92d-be7c9ebc7559
8.  Host: localhost:5000
9.  Accept-Encoding: gzip, deflate, br
10. Connection: keep-alive
11. Content-Length: 17
12.
13. {"att1":"value1"}
14.
15. HTTP/1.0 400 BAD REQUEST
16. Content-Type: application/json; charset=utf-8
17. Content-Length: 97
18. Server: Werkzeug/1.0.1 Python/3.8.1
19. Date: Tue, 28 Jul 2020 07:50:11 GMT
20.
21. {"réponse": {"erreurs": ["le corps du POST n'est pas une liste ou alors cette liste est vide"]}}
```

Cas 3 : envoyons une chaîne jSON qui soit une liste dont les éléments ne sont pas tous des dictionnaires

```
POST / HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json
User-Agent: PostmanRuntime/7.26.2
Accept: */*
Cache-Control: no-cache
Postman-Token: a1528a5f-777c-413f-b3be-7d4e9955b12a
Host: localhost:5000
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 7

[0,1,2]

HTTP/1.0 400 BAD REQUEST
Content-Type: application/json; charset=utf-8
Content-Length: 85
Server: Werkzeug/1.0.1 Python/3.8.1
Date: Tue, 28 Jul 2020 07:52:10 GMT

{"réponse": {"erreurs": ["le corps du POST doit être une liste de dictionnaires"]}}
```

Cas 4 : envoyons une liste de dictionnaires avec un dictionnaire n'ayant pas les bonnes clés

```
POST / HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json
User-Agent: PostmanRuntime/7.26.2
Accept: */*
Cache-Control: no-cache
Postman-Token: ba964d81-c9d9-46ff-a521-b4c4e5639484
Host: localhost:5000
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 19

[{"att1":"value1"}]

HTTP/1.0 400 BAD REQUEST
Content-Type: application/json; charset=utf-8
Content-Length: 112
Server: Werkzeug/1.0.1 Python/3.8.1
Date: Tue, 28 Jul 2020 07:54:33 GMT

{"réponse": {"erreurs": [{"att1": "value1", "erreur": "MyException[2, la clé [att1] n'est pas
autorisée]"}]}}
```

Cas 5 : envoyons une liste de dictionnaires avec un dictionnaire avec des clés manquantes :

```
POST / HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json
User-Agent: PostmanRuntime/7.26.2
Accept: */*
Cache-Control: no-cache
Postman-Token: 98aec51d-f37d-4c14-81cd-c7ffcbbcdc65
Host: localhost:5000
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 18

[{"marié":"oui"}]

HTTP/1.0 400 BAD REQUEST
Content-Type: application/json; charset=utf-8
Content-Length: 125
Server: Werkzeug/1.0.1 Python/3.8.1
Date: Tue, 28 Jul 2020 07:56:40 GMT

{"réponse": {"erreurs": [{"marié": "oui", "erreur": "le dictionnaire doit inclure les clés [marié, enfants,
salaire]"}]}}
```

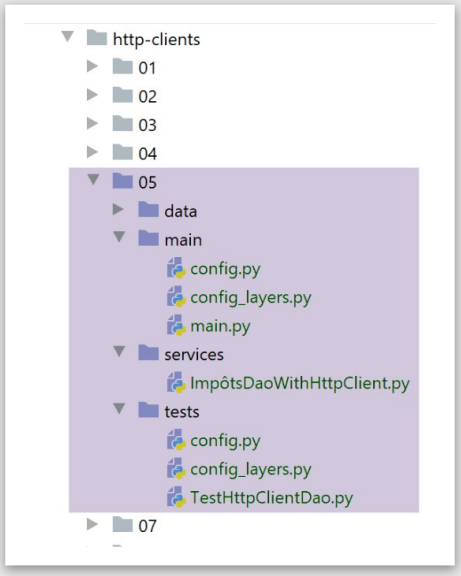
Cas 6 : envoyons une liste de dictionnaires avec un dictionnaire ayant les bonnes clés mais certaines ayant des valeurs erronées :

```
POST / HTTP/1.1
Authorization: Basic YWRtaW46YWRtaW4=
Content-Type: application/json
User-Agent: PostmanRuntime/7.26.2
Accept: */*
Cache-Control: no-cache
Postman-Token: 3083e601-dee4-4e15-9ea4-fc0328d0fcf0
Host: localhost:5000
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 46

[{"marié":"x", "enfants":"x", "salaire":"x"}]

HTTP/1.0 400 BAD REQUEST
Content-Type: application/json; charset=utf-8
Content-Length: 167
Server: Werkzeug/1.0.1 Python/3.8.1
Date: Tue, 28 Jul 2020 07:59:32 GMT

{"réponse": {"erreurs": [{"marié": "x", "enfants": "x", "salaire": "x", "erreur": "MyException[31,
l'attribut marié [x] doit avoir l'une des valeurs oui / non]"}]}}
```



28.3 Le client web

Le dossier **[http-clients/05]** (version 10) est obtenu initialement par recopie du dossier **[http-clients/02]** (version 7). Il est ensuite modifié.

28.3.1 La couche [dao]

La couche **[dao]** est implémentée par la classe **[ImpôtsDaoWithHttpClient]** suivante :

```
# imports

import requests
from flask_api import status

from AbstractImpôtsDao import AbstractImpôtsDao
from AdminData import AdminData
from ImpôtsError import ImpôtsError
from InterfaceImpôtsMétier import InterfaceImpôtsMétier
from TaxPayer import TaxPayer


class ImpôtsDaoWithHttpClient(AbstractImpôtsDao, InterfaceImpôtsMétier):

    # constructeur
    def __init__(self, config: dict):
```

```
17.         …
18.
19.     # méthode inutilisée
20.     def get_admindata(self) -> AdminData:
21.         pass
22.
23.     # calcul de l'impôt
24.     def calculate_tax(self, taxpayer: TaxPayer, admindata: AdminData = None):
25.         …
26.
27.     # calcul de l'impôt en mode bulk
28.     def calculate_tax_in_bulk_mode(self, taxpayers: list) -> list:
29.         # on laisse remonter les exceptions
30.
31.         # on transforme les taxpayers en liste de dictionnaires
32.         # on ne garde que les propriétés [marié, enfants, salaire]
33.         list_dict_taxpayers = list(
34.             map(lambda taxpayer:
35.                 taxpayer.asdict(included_keys=[
36.                     '_TaxPayer__marié',
37.                     '_TaxPayer__enfants',
38.                     '_TaxPayer__salaire']),
39.                 taxpayers))
40.
41.         # connexion au serveur
42.         config_server = self.__config_server
43.         if config_server['authBasic']:
44.             response = requests.post(config_server['urlServer'], json=list_dict_taxpayers,
45.                                      auth=(config_server["user"]["login"],
46.                                            config_server["user"]["password"]))
47.         else:
48.             response = requests.post(config_server['urlServer'], json=list_dict_taxpayers)
49.         # mode debug ?
50.         if self.__debug:
51.             # logueur
52.             if not self.__logger:
53.                 self.__logger = self.__config['logger']
54.             # on logue
55.             self.__logger.write(f"{response.text}\n")
56.         # code de statut de la réponse HTTP
57.         status_code = response.status_code
58.         # on met la réponse jSON dans un dictionnaire
59.         résultat = response.json()
60.         # erreur si code de statut différent de 200 OK
61.         if status_code != status.HTTP_200_OK:
62.             # on sait que les erreurs ont été associées à la clé [erreurs] de la réponse
63.             raise ImpôtsError(93, résultat['réponse']['erreurs'])
64.         # on sait que le résultat a été associé à la clé [results] de la réponse
65.         list_dict_taxpayers2 = résultat['réponse']['results']
66.         # on met à jour la liste initiale des taxpayers avec les résultats reçus
67.         for i in range(len(taxpayers)):
68.             # mise à jour de taxpayers[i]
69.             taxpayers[i].fromdict(list_dict_taxpayers2[i])
70.         # ici le paramètre [taxpayers] a été mis à jour avec les résultats du serveur
```

- lignes 1-26 : le code reste ce qu'il était dans la version 7 et dans d'autres versions ;
- lignes 27-70 : on introduit une nouvelle méthode **[calculate_tax_in_bulk_mode]** dont le rôle est de calculer l'impôt d'une liste de contribuables ;
- ligne 28 : **[taxpayers]** est cette liste de contribuables ;
- lignes 31-39 : on passe d'une liste d'objets de type **[TaxPayer]** à une liste de dictionnaires grâce à une fonction **[map]** ;
- lignes 34-38 : la fonction lambda utilisée transforme un objet de type **[TaxPayer]** en un dictionnaire de type **[dict]** ayant les seules clés **[marié, enfants, salaire]**. On utilise pour cela le paramètre nommé **[included_keys]** de la méthode **[BaseEntity.asdict]**. On rappelle que pour connaître les noms exacts des propriétés à mettre dans les paramètres **[excluded_keys, included_keys]**, il faut utiliser le dictionnaire prédéfini **[taxpayer.__dict__]** ;
- lignes 41-48 : connexion au serveur puis obtention de sa réponse HTTP ;
- lignes 44, 48 :
  - on utilise la méthode statique **[requests.post]** pour faire un POST vers le serveur ;
  - on utilise le paramètre nommé **[json]** pour indiquer que le corps du POST est une chaîne jSON. Cela va avoir deux conséquences :
    - l'objet affecté au paramètre nommé **[json]**, ici une liste de dictionnaires, va être transformé en chaîne jSON ;
    - l'entête

```
Content-Type: application/json
```

sera inclus dans les entêtes HTTP du POST ;
- ligne 59 : la réponse jSON du serveur est désérialisée dans le dictionnaire **[résultat]** ;
- lignes 61-63 : on gère l'éventuelle erreur envoyée par le serveur ;
- ligne 65 : les résultats du calcul de l'impôt sont dans une liste de dictionnaires ;
- lignes 67-69 : ces résultats sont exploités pour mettre à jour la liste initiale des contribuables **[taxpayers]** initialement reçus par la méthode, ligne 28 ;
- ligne 70 : ici la liste la liste initiale des contribuables a été mise à jour avec les résultats du calcul de l'impôt ;

28.3.2 Le script principal [main]

Le script principal **[main]** évolue de la façon suivante : seule la fonction **[thread_function]** exécutée par les threads créés par le client est modifiée. Le reste du code reste à l'identique.

```
1.  # exécution de la couche [dao] dans un thread
2.  # taxpayers est une liste de contribuables
3.  def thread_function(dao: ImpôtsDaoWithHttpClient, logger: Logger, taxpayers: list):
4.      # log début du thread
5.      thread_name = threading.current_thread().name
6.      nb_taxpayers = len(taxpayers)
7.      # log
8.      logger.write(f"début du calcul de l'impôt des {nb_taxpayers} contribuables\n")
9.      # on calcule l'impôt des contribuables
10.     dao.calculate_tax_in_bulk_mode(taxpayers)
11.     # log
12.     logger.write(f"fin du calcul de l'impôt des {nb_taxpayers} contribuables\n")
```

- lignes 9-10 : alors que précédemment on avait une boucle qui passait successivement chacun des contribuables à la méthode **[dao.calculate_tax]**, ici on ne fait qu'un unique appel à la méthode **[dao.calculate_tax_in_bulk_mode]** à laquelle on passe tous les contribuables ;

28.3.3 Exécution du client

Nous allons comparer les temps d'exécution des versions :

- 7, où chaque contribuable fait l'objet d'une requête HTTP ;
- 10 (celle-ci) où on rassemble des contribuables dans une unique requête HTTP ;

Tout d'abord la version 6. Pour comparer les deux versions, on met la propriété **[sleep_time]** du serveur à zéro pour qu'il n'y ait pas d'attente forcée des threads. Les logs du client sont les suivants :

```
1.  2020-07-28 14:20:45.811347, Thread-1 : début du thread [Thread-1] avec 4 contribuable(s)
2.  2020-07-28 14:20:45.811347, Thread-1 : début du calcul de l'impôt de {"id": 1, "marié": "oui", "enfants":
    2, "salaire": 55555}
3.  …
4.  2020-07-28 14:20:45.913065, Thread-3 : fin du calcul de l'impôt de {"id": 11, "marié": "oui", "enfants": 3,
    "salaire": 200000, "impôt": 42842, "surcôte": 17283, "taux": 0.41, "décôte": 0, "réduction": 0}
5.  2020-07-28 14:20:45.913065, Thread-3 : fin du thread [Thread-3]
```

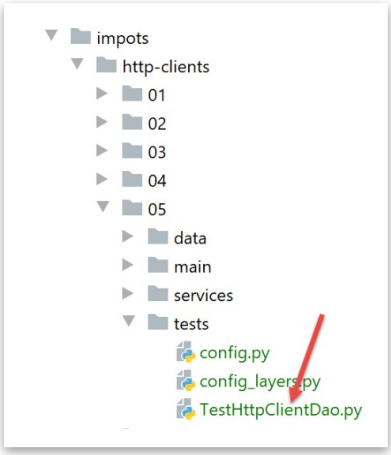
La durée de l'exécution du client pour calculer l'impôt de 11 contribuables est donc **[913065-811347= 101718]**, ç-à-d environ 102 millisecondes.

Faisons la même chose avec la version 10 (sleep_time du serveur à zéro). Les logs du client sont alors les suivants :

```
1.  2020-07-28 14:25:31.871428, Thread-1 : début du calcul de l'impôt des 4 contribuables
2.  2020-07-28 14:25:31.873594, Thread-2 : début du calcul de l'impôt des 3 contribuables
3.  2020-07-28 14:25:31.877429, Thread-3 : début du calcul de l'impôt des 3 contribuables
4.  2020-07-28 14:25:31.882855, Thread-4 : début du calcul de l'impôt des 1 contribuables
5.  2020-07-28 14:25:31.930723, Thread-2 : {"réponse": {"results": [{"marié": "non", "enfants": 3, "salaire":
    100000, "impôt": 16782, "surcôte": 7176, "taux": 0.41, "décôte": 0, "réduction": 0}, {"marié": "oui",
    "enfants": 3, "salaire": 100000, "impôt": 9200, "surcôte": 2180, "taux": 0.3, "décôte": 0, "réduction": 0},
    {"marié": "oui", "enfants": 5, "salaire": 100000, "impôt": 4230, "surcôte": 0, "taux": 0.14, "décôte": 0,
    "réduction": 0}]}}
6.  ….
7.  2020-07-28 14:25:31.935958, Thread-4 : fin du calcul de l'impôt des 1 contribuables
8.  2020-07-28 14:25:31.935958, Thread-1 : fin du calcul de l'impôt des 4 contribuables
```

La durée de l'exécution du client pour calculer l'impôt de 11 contribuables est donc **[935958-871428= 64530 ns]** (ligne 8 – ligne 1), ç-à-d environ 65 millisecondes. Cette nouvelle version 10 amène ainsi un gain de 57 % environ sur la version 7.

28.3.4 Tests de la couche [dao] du client



Le test **[TestHttpClientDao]** du client de la version 10 est très proche de celui de la version 7 :

```
1.  import unittest
2.
3.  from Logger import Logger
4.
5.
6.  class TestHttpClientDao(unittest.TestCase):
7.
8.      def test_1(self) -> None:
9.          from TaxPayer import TaxPayer
10.
11.         # {'marié': 'oui', 'enfants': 2, 'salaire': 55555,
12.         # 'impôt': 2814, 'surcôte': 0, 'décôte': 0, 'réduction': 0, 'taux': 0.14}
13.         taxpayer = TaxPayer().fromdict({"marié": "oui", "enfants": 2, "salaire": 55555})
14.         dao.calculate_tax_in_bulk_mode([taxpayer])
15.         # vérification
16.         self.assertAlmostEqual(taxpayer.impôt, 2815, delta=1)
17.         self.assertEqual(taxpayer.décôte, 0)
18.         self.assertEqual(taxpayer.réduction, 0)
19.         self.assertAlmostEqual(taxpayer.taux, 0.14, delta=0.01)
20.         self.assertEqual(taxpayer.surcôte, 0)
21.
22.     …
23.
24. if __name__ == '__main__':
25.     # on configure l'application
26.     import config
27.     config = config.configure({})
28.
29.     # logger
30.     logger = Logger(config["logsFilename"])
31.     # on le mémorise dans la config
32.     config["logger"] = logger
33.     # on récupère la couche [dao]
34.     dao = config["layers"]["dao"]
35.
36.     # on exécute les méthodes de test
37.     print("tests en cours...")
38.     unittest.main()
```

- ligne 14 : au lieu d'appeler la méthode **[dao.calculate_tax]**, on appelle la méthode **[dao.calculate_tax_in_bulk_mode]** à laquelle on passe une liste (présence des crochets) d'un contribuable ;

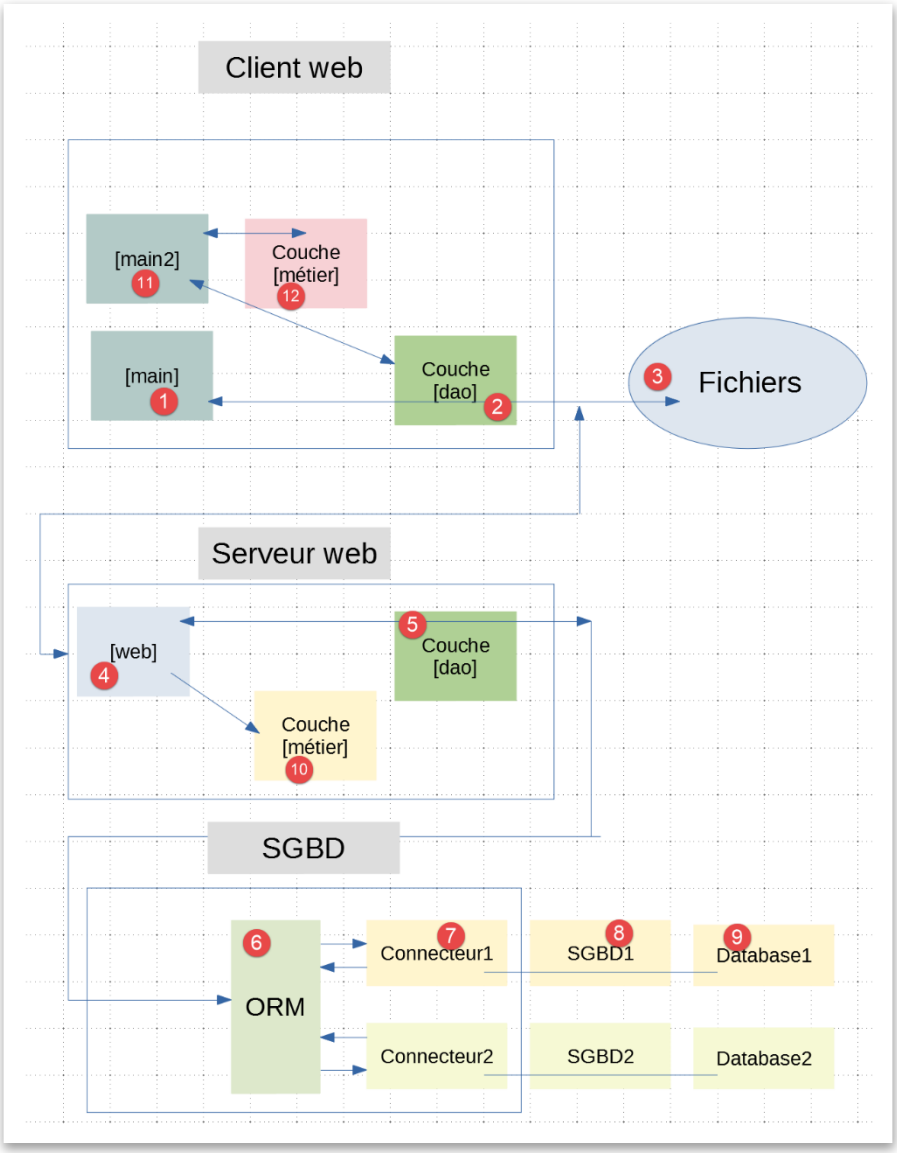
Tous les tests passent.

29 Exercice d'application : version 11

29.1 Introduction

Dans les versions précédentes de l'application client / serveur de calcul de l'impôt, la couche **[métier]** qui implémente les règles métier de ce calcul était côté serveur. On se propose maintenant de la déplacer côté client. Quel est l'intérêt ? Du travail que faisait le serveur va être déplacé côté client. Si on pense à la situation d'un serveur interrogé par N clients, N calculs métier de l'impôt seront faits par les clients. Dans les versions précédentes, le serveur faisait ces N calculs métier. Parce qu'il ne fait plus le calcul métier, le serveur va répondre plus rapidement à ses clients et va alors pouvoir en servir davantage simultanément.

L'architecture client / serveur devient la suivante :



- la couche **[métier] [10]** a été dupliquée **[12]** sur le client ;
- un nouveau script **[main2] [11]** a été ajouté au client ;

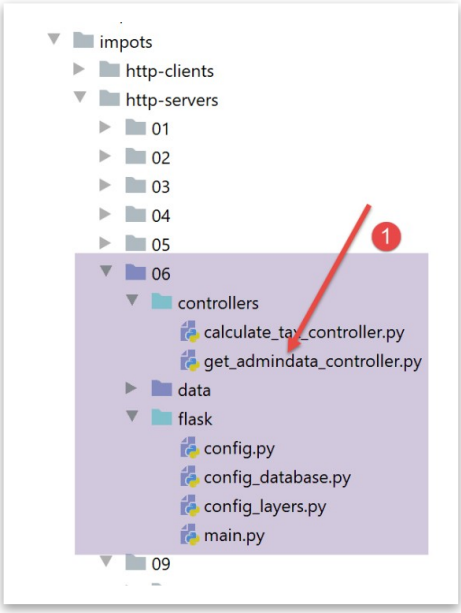
Le client web aura deux façon de calculer l'imppôt de la liste de contribuables trouvée en **[3]** :

- utiliser la méthode de la version précédente. Il utilise la couche **[métier] [10]** du serveur. Le script **[main]** utilisera cette méthode ;
- se contenter de demander au serveur les données de l'administration fiscale **[2-4]** puis utiliser la couche **[métier] [12]** locale au client ;

Nous comparerons les performances des deux méthodes.

29.2 Le serveur web

L'arborescence du serveur web sera la suivante :



- le dossier **[http-servers/06]** est initialement obtenu par recopie du dossier **[http-servers/05]**. On va en effet conserver l'acquis de la version 10 précédente. On va simplement lui ajouter une nouvelle fonctionnalité. Celle-ci est matérialisée par la présence d'un nouveau contrôleur **[get_admindata_controller] [1]**. L'autre contrôleur **[calculate_tax_controller]** n'est autre que l'ancien contrôleur **[index_controller]** qui a été renommé ;

29.3 Configuration

Le serveur offrira deux URL de service :

- **[/calculate-tax]** pour calculer l'impôt d'une liste de contribuables passée dans le corps d'un POST. Elle correspond donc à l'URL **[/]** de la version 10 précédente ;
- **[/get-admindata]** délivre la chaîne jSON des données de l'administration fiscale ;

La configuration **[config]** associe chacune de ces URL au contrôleur qui la traite :

```
1.      # dictionnaire des contrôleurs de l'application web
2.      import calculate_tax_controller, get_admindata_controller
3.      controllers = {"calculate-tax": calculate_tax_controller, "get-admindata": get_admindata_controller}
4.      config['controllers'] = controllers
```

29.4 Le script principal [main]

Le script principal **[main]** restructure le script **[main]** de la version précédente :

```
1.  # on attend un paramètre mysql ou pgres
2.  import sys
3.  syntaxe = f"{sys.argv[0]} mysql / pgres"
4.  erreur = len(sys.argv) != 2
5.  if not erreur:
6.      sgbd = sys.argv[1].lower()
7.      erreur = sgbd != "mysql" and sgbd != "pgres"
8.  if erreur:
9.      print(f"syntaxe : {syntaxe}")
10.     sys.exit()
11.
12. # on configure l'application
13. import config
14. config = config.configure({'sgbd': sgbd})
15.
16. # dépendances
17. …
```

```
18.
19. # récupération des données de l'administration fiscale
20. erreur = False
21. try:
22.     # admindata sera une donnée de portée application en lecture seule
23.     config["admindata"] = config["layers"]["dao"].get_admindata()
24.     # log de réussite
25.     logger.write("[serveur] connexion à la base de données réussie\n")
26. except ImpôtsError as ex:
27.     # on note l'erreur
28.     …
29. …
30. # s'il y a eu erreur on s'arrête
31. if erreur:
32.     sys.exit(2)
33.
34. # l'application Flask peut démarrer
35. app = Flask(__name__)
36.
37. # contrôleur principal
38. def main_controller() -> tuple:
39.     # on récupère l'action demandée
40.     dummy, action=request.path.split('/')
41.     logger = None
42.     try:
43.         # logger
44.         logger = Logger(config["logsFilename"])
45.         # on le mémorise dans une config associée au thread
46.         thread_config = {"logger": logger}
47.         thread_name = threading.current_thread().name
48.         config[thread_name] = {"config": thread_config}
49.         # on logue la requête
50.         logger.write(f"[index] requête : {request}\n")
51.         # on interrompt le thread si cela a été demandé
52.         sleep_time = config["sleep_time"]
53.         if sleep_time != 0:
54.             # la pause est aléatoire pour que certains threads soient interrompus et d'autres pas
55.             aléa = randint(0, 1)
56.             if aléa == 1:
57.                 # log avant pause
58.                 logger.write(f"[index] mis en pause du thread pendant {sleep_time} seconde(s)\n")
59.                 # pause
60.                 time.sleep(sleep_time)
61.         # on fait exécuter la requête par le contrôleur de la requête
62.         controller = config['controllers'][action]
63.         résultat, status_code = controller.execute(request, config)
64.         # y-a-t-il eu une erreur fatale ?
65.         if status_code == status.HTTP_500_INTERNAL_SERVER_ERROR:
66.             # on envoie un mail à l'administrateur de l'application
67.             config_mail = config["adminMail"]
68.             config_mail["logger"] = logger
69.             SendAdminMail.send(config_mail, json.dumps(résultat, ensure_ascii=False))
70.         # on logue la réponse
71.         logger.write(f"[index] {résultat}\n")
72.         # on envoie la réponse
73.         print(résultat)
74.         return json_response(résultat, status_code)
75.     except BaseException as erreur:
76.         # on logue l'erreur si c'est possible
77.         if logger:
78.             logger.write(f"[index] {erreur}")
79.         # on prépare la réponse au client
80.         résultat = {"réponse": {"erreurs": [f"{erreur}"]}}
81.         # on envoie la réponse
82.         return json_response(résultat, status.HTTP_500_INTERNAL_SERVER_ERROR)
83.     finally:
84.         # on ferme le fichier de logs s'il a été ouvert
85.         if logger:
86.             logger.close()
87.
88. # calcul de l'impôt
89. @app.route('/calculate-tax', methods=['POST'])
90. @auth.login_required
91. def calculate_tax():
92.     # on passe la main au contrôleur principal
93.     return main_controller()
94.
```

```
95. # obtenir les données de l'administration fiscale
96. @app.route('/get-admindata', methods=['GET'])
97. @auth.login_required
98. def get_admindata():
99.     # on passe la main au contrôleur principal
100.    return main_controller()
101.
102.# main uniquement
103.if __name__ == '__main__':
104.    # on lance le serveur
105.    app.config.update(ENV="development", DEBUG=True)
106.    app.run(threaded=True)
```

- lignes 88-93 : la fonction **[calculate_tax]** traite l'URL **[/calculate-tax]** ;
- lignes 95-100 : la fonction **[get_admindata]** traite l'URL **[/get-admindata]** ;
- ces deux fonctions ne font rien par elles-mêmes. Elles passent immédiatement la main au contrôleur principal **[main_controller]** des lignes 37-86 ;
- lignes 37-86 : le contrôleur principal **[main_controller]** n'est rien d'autre que la fonction **[index]** de la version précédente à un détail près : là où la fonction **[index]** ne traitait qu'une unique URL, ici **[main_controller]** traite deux URL. Il lui faut donc faire traiter celles-ci par l'un des deux contrôleurs **[calculate_tax_controller, get_admin,data_controller]** ;
- lignes 39-40 : on récupère l'action demandée **[calculate_tax]** ou **[get_admindata]**. Cette information est dans le chemin de l'URL **[request.path]**. Selon les cas, **[request.path]** vaut **[/get-admindata]** ou **[/calculate_tax]**. Le split de la ligne 40 va donner deux éléments :
  - la chaîne vide pour la partie qui précède le / ;
  - le nom de l'action demandée pour la partie qui suit le / ;
- lignes 62-63 : une fois l'action de l'URL récupérée, on sait quel contrôleur utiliser pour traiter l'URL. Cette informations sont dans la configuration **[config]** ;

29.5 Les contrôleurs

Le contrôleur **[calculate_tax_controller]** n'est autre que le contrôleur **[index_controller]** de la version précédente.

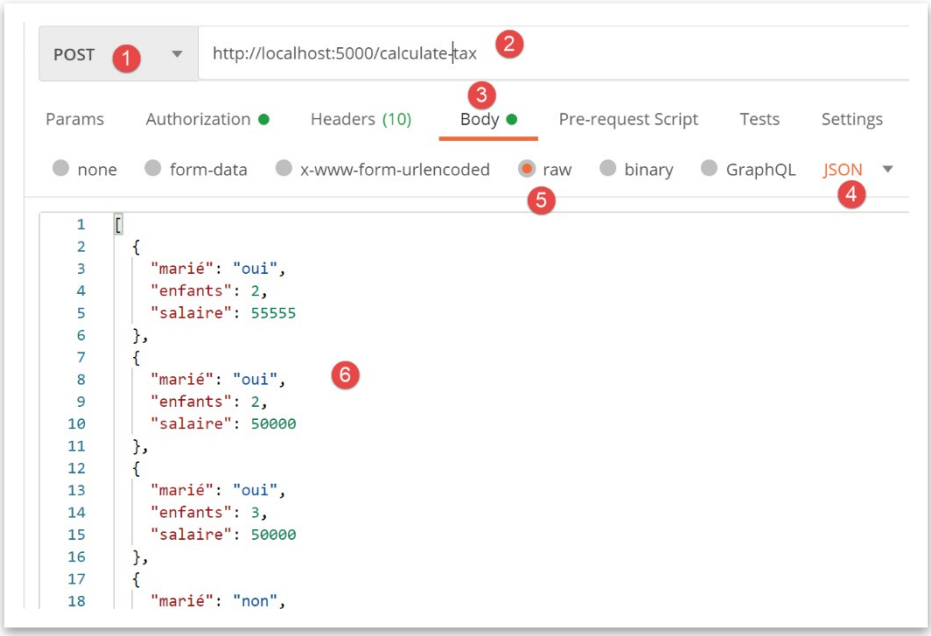
Le contrôleur **[get_admindata_controller]** est lui le suivant :

```
1.  from flask_api import status
2.  from werkzeug.local import LocalProxy
3.
4.  def execute(request: LocalProxy, config: dict) -> tuple:
5.      # on rend la réponse
6.      return {"réponse": {"result": config["admindata"].asdict()}}, status.HTTP_200_OK
```

- l'URL **[/get-admindata]** doit rendre la chaîne jSON des données de l'administration fiscale ;
- ligne 6 : celles-ci ont été récupérées par le script principal **[main]** et mises dans le dictionnaire **[config]** sous la forme d'un objet **[AdminData]**. On rend le dictionnaire de cet objet ;

29.6 Tests Postman

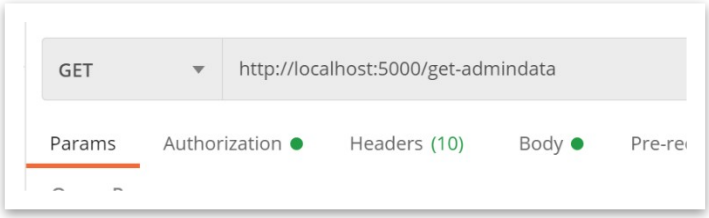
On lance le serveur web, le SGBD et le serveur de mails **[hMailServer]**. Puis avec un client Postman, on fait le calcul de l'impôt de plusieurs contribuables :



Dans la console Postman, le dialogue client / serveur est le suivant :

```
1.  POST /calculate-tax HTTP/1.1
2.  Authorization: Basic YWRtaW46YWRtaW4=
3.  Content-Type: application/json
4.  User-Agent: PostmanRuntime/7.26.2
5.  Accept: */*
6.  Cache-Control: no-cache
7.  Postman-Token: 5e71461a-fec8-4315-85e8-41721de939e5
8.  Host: localhost:5000
9.  Accept-Encoding: gzip, deflate, br
10. Connection: keep-alive
11. Content-Length: 824
12.
13. [
14. {
15. "marié": "oui",
16. "enfants": 2,
17. "salaire": 55555
18. },
19. …
20. {
21. "marié": "oui",
22. "enfants": 3,
23. "salaire": 200000
24. }
25. ]
26.
27. HTTP/1.0 200 OK
28. Content-Type: application/json; charset=utf-8
29. Content-Length: 1461
30. Server: Werkzeug/1.0.1 Python/3.8.1
31. Date: Wed, 29 Jul 2020 07:02:07 GMT
32.
33. {"réponse": {"results": [{"marié": "oui", "enfants": 2, "salaire": 55555, "impôt": 2814, "surcôte": 0,
    "taux": 0.14, "décôte": 0, "réduction": 0}, {"marié": "oui", "enfants": 2, "salaire": 50000, "impôt": 1384,
    "surcôte": 0, "taux": 0.14, "décôte": 384, "réduction": 347…]}}
```

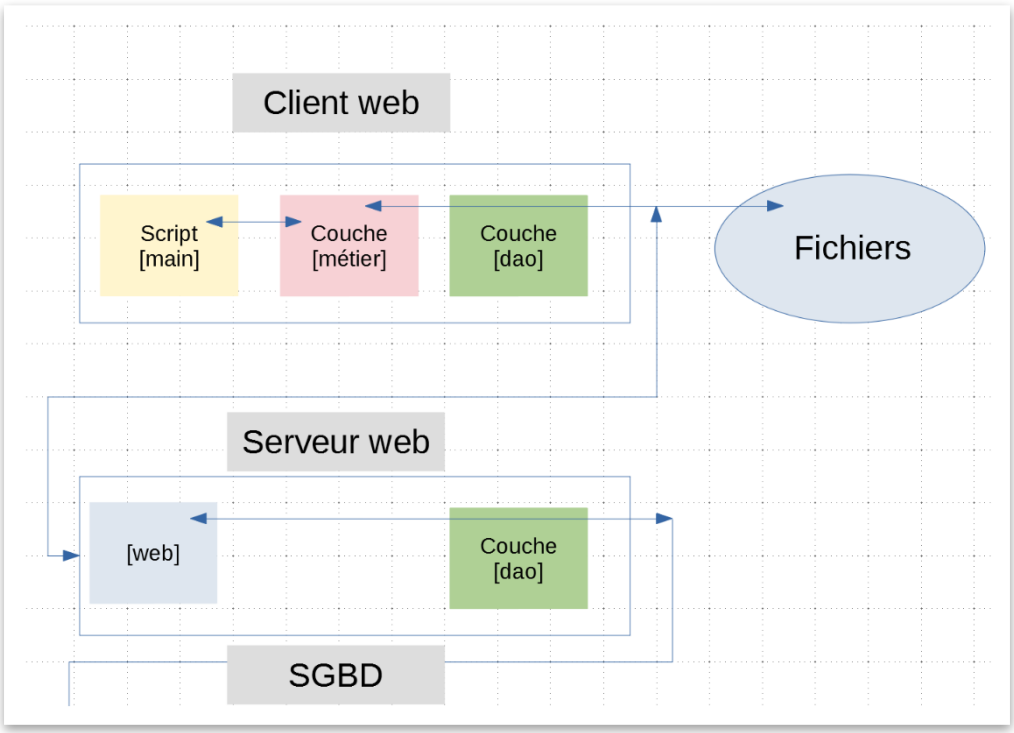
Maintenant demandons l'URL **[/get-admindata]** avec un GET :

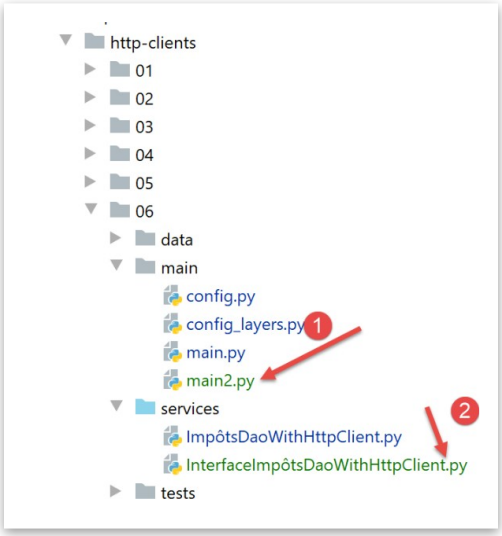


Le dialogue client / serveur dans la console Postman est le suivant :

```
1.  GET /get-admindata HTTP/1.1
2.  Authorization: Basic YWRtaW46YWRtaW4=
3.  User-Agent: PostmanRuntime/7.26.2
4.  Accept: */*
5.  Cache-Control: no-cache
6.  Postman-Token: 4af342c4-7ecb-4ab2-9e12-d653f81da424
7.  Host: localhost:5000
8.  Accept-Encoding: gzip, deflate, br
9.  Connection: keep-alive
10.
11. HTTP/1.0 200 OK
12. Content-Type: application/json; charset=utf-8
13. Content-Length: 596
14. Server: Werkzeug/1.0.1 Python/3.8.1
15. Date: Wed, 29 Jul 2020 07:07:24 GMT
16.
17. {"réponse": {"result": {"limites": [9964.0, 27519.0, 73779.0, 156244.0, 93749.0], "coeffr": [0.0, 0.14,
    0.3, 0.41, 0.45], "coeffn": [0.0, 1394.96, 5798.0, 13913.7, 20163.4], "plafond_decote_couple": 1970.0,
    "valeur_reduc_demi_part": 3797.0, "plafond_revenus_celibataire_pour_reduction": 21037.0,
    "plafond_qf_demi_part": 1551.0, "abattement_dixpourcent_max": 12502.0,
    "plafond_impot_celibataire_pour_decote": 1595.0, "plafond_decote_celibataire": 1196.0,
    "plafond_revenus_couple_pour_reduction": 42074.0, "id": 1, "abattement_dixpourcent_min": 437.0,
    "plafond_impot_couple_pour_decote": 2627.0}}}
```

29.7 Le client web





Le dossier **[http-clients/06]** est initialement obtenu par recopie du dossier **[http-clients/05]**. Le travail de modification consiste essentiellement à :

- modifier la configuration **[config_layers]** pour qu'elle intégre désormais une couche **[métier]**. Auparavant elle n'avait qu'une couche **[dao]** ;
- ajouter une nouvelle méthode à la couche **[dao]** ;
- écrire un script **[main2]** qui s'appuiera sur la couche **[métier]** du client pour calculer l'impôt des contribuables ;

29.7.1 Configuration des couches du client

La configuration des couches intervient en deux points :

- dans la configuration **[config]** qui doit inclure dans les dépendances du client le dossier contenant l'implémentation de la couche **[métier]**. Ce dossier était déjà inclus dans les dépendances :

```
absolute_dependencies = [
        # dossiers du projet
        # BaseEntity, MyException
        f"{root_dir}/classes/02/entities",
        # InterfaceImpôtsDao, InterfaceImpôtsMétier, InterfaceImpôtsUi
        f"{root_dir}/impots/v04/interfaces",
        # AbstractImpôtsdao, ImpôtsConsole, ImpôtsMétier
        f"{root_dir}/impots/v04/services",
        # ImpotsDaoWithAdminDataInDatabase
        f"{root_dir}/impots/v05/services",
        # AdminData, ImpôtsError, TaxPayer
        f"{root_dir}/impots/v04/entities",
        # Constantes, tranches
        f"{root_dir}/impots/v05/entities",
        # ImpôtsDaoWithHttpClient
        f"{script_dir}/../services",
        # scripts de configuration
        script_dir,
        # Logger
        f"{root_dir}/impots/http-servers/02/utilities",
    ]
```

Puis le fichier **[config_layers]** doit être modifié :

```
def configure(config: dict) -> dict:
    # instanciation des couches de l'applicatuon

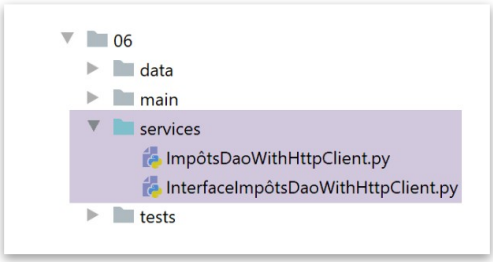
    # couche [métier]
    from ImpôtsMétier import ImpôtsMétier
    métier = ImpôtsMétier()

    # couche dao
    from ImpôtsDaoWithHttpClient import ImpôtsDaoWithHttpClient
```

```
10.     dao = ImpôtsDaoWithHttpClient(config)
11.
12.     # on rend la configuration des couches
13.     return {
14.         "dao": dao,
15.         "métier": métier
16.     }
```

- lignes 4-6 : instanciation de la couche **[métier]** ;
- lignes 13-16 : la couche **[métier]** est rendue dans le dictionnaire des couches ;

29.7.2 Implémentation de la couche [dao]



La couche **[dao]** présentera l'interface **[InterfaceImpôtsDaoWithHttpClient]** suivante :

```
1.  from abc import abstractmethod
2.
3.  from AbstractImpôtsDao import AbstractImpôtsDao
4.
5.  class InterfaceImpôtsDaoWithHttpClient(AbstractImpôtsDao):
6.
7.      # calcul de l'impôt
8.      @abstractmethod
9.      def calculate_tax_in_bulk_mode(self, taxpayers: list):
10.         pass
```

- ligne 5 : l'interface **[InterfaceImpôtsDaoWithHttpClient]** hérite de la classe abstraite **[AbstractImpôtsDao]** qui gère l'accès au système de fichiers du client. On rappelle qu'elle a une méthode abstraite **[get_admindata]** ;
- lignes 7-10 : la méthode **[calculate_tax_in_bulk_mode]** que nous avons définie dans la version précédente permet le calcul de l'impôt d'une liste de contribuables ;

Cette interface est implémentée par la classe **[ImpôtsDaoWithHttpClient]** suivante :

```
1.  # imports
2.
3.  import json
4.
5.  import requests
6.  from flask_api import status
7.
8.  from AbstractImpôtsDao import AbstractImpôtsDao
9.  from AdminData import AdminData
10. from ImpôtsError import ImpôtsError
11. from InterfaceImpôtsDaoWithHttpClient import InterfaceImpôtsDaoWithHttpClient
12.
13. class ImpôtsDaoWithHttpClient(InterfaceImpôtsDaoWithHttpClient):
14.
15.     # constructeur
16.     def __init__(self, config: dict):
17.         # initialisation parent
18.         AbstractImpôtsDao.__init__(self, config)
19.         # mémorisation éléments de la configuration
20.         # config générale
21.         self.__config = config
22.         # serveur
23.         self.__config_server = config["server"]
24.         # mode debug
25.         self.__debug = config["debug"]
26.         # logger
27.         self.__logger = None
28.
```

```
29.     def get_admindata(self) -> AdminData:
30.         # on laisse remonter les exceptions
31.
32.         # url de service
33.         config_server = self.__config_server
34.         config_services = config_server['url_services']
35.         url_service = f"{config_server['urlServer']}{config_services['get-admindata']}"
36.
37.         # connexion
38.         if config_server['authBasic']:
39.             response = requests.get(url_service,
40.                                     auth=(
41.                                         config_server["user"]["login"],
42.                                         config_server["user"]["password"]))
43.         else:
44.             response = requests.get(url_service)
45.
46.         # mode debug ?
47.         if self.__debug:
48.             # logueur
49.             if not self.__logger:
50.                 self.__logger = self.__config['logger']
51.             # on logue
52.             self.__logger.write(f"{response.text}\n")
53.
54.         # code de statut
55.         status_code = response.status_code
56.         # résultat sous forme d'un dictionnaire
57.         résultat = json.loads(response.text)
58.         # si code de statut différent de 200 OK
59.         if status_code != status.HTTP_200_OK:
60.             raise ImpôtsError(58, résultat['réponse']['erreurs'])
61.         # on rend le résultat (un dictionnaire)
62.         return AdminData().fromdict(résultat["réponse"]["result"])
63.
64.     # calcul de l'impôt en mode bulk
65.     def calculate_tax_in_bulk_mode(self, taxpayers: list):
66.         # on laisse remonter les exceptions
67.
68.         …
```

- ligne 13 : la classe **[ImpôtsDaoWithHttpClient]** implémente l'interface **[InterfaceImpôtsDaoWithHttpClient]**. Elle dérive donc de la classe **[AbstractImpôtsDao]** ;
- lignes 65-66 : la méthode **[calculate_tax_in_bulk_mode]** étudiée dans la version précédente ;
- lignes 29-62 : la méthode **[get_admindata]** que la classe parent **[AbstractImpôtsDao]** a déclarée abstraite. Elle est donc implémentée dans la classe fille ;
- lignes 33-35 : on détermine l'URL du service web que la méthode **[get-admindata]** doit interroger. Ces URL de service sont définies dans la configuration **[config]** du client :

```
1.         # le serveur de calcul de l'impôt
2.          "server": {
3.              "urlServer": "http://127.0.0.1:5000",
4.              "authBasic": True,
5.              "user": {
6.                  "login": "admin",
7.                  "password": "admin"
8.              },
9.              "url_services": {
10.                 "calculate-tax": "/calculate-tax",
11.                 "get-admindata": "/get-admindata"
12.             }
13.         },
```

  - lignes 9-12 : les deux URL du serveur web ;
- lignes 37-44 : l'URL de service est interrogée de façon synchrone ;
- lignes 46-42 : si la configuration le demande, la réponse du serevur est loguée ;
- ligne 57 : on sait que le serveur a envoyé une chaîne jSON d'un dictionnaire ;
- lignes 58-60 : si le statut HTTP de la réponse n'est pas 200, alors on lance une exception ;
- lignes 61-62 : on rend l'objet **[AdminData]** encapsulant les données de l'administration fiscale envoyées par le serveur ;

29.8 Les scripts [main, main2]

Le script **[main]** est celui de la version précédente. Il utilise la méthode **[calculate_tax_in_bulk_mode]** de la couche **[dao]** et utilise donc la couche **[métier]** du serveur ;

Le script **[main2]** fait la même chose que le script **[main]** mais en utilisant la couche **[métier]** du client :

```
1.  # on configure l'application
2.
3.  import config
4.  config = config.configure({})
5.
6.  # dépendances
7.  from ImpôtsError import ImpôtsError
8.  from Logger import Logger
9.
10. logger = None
11. # code
12. try:
13.     # logger
14.     logger = Logger(config["logsFilename"])
15.     # on le mémorise dans la config
16.     config["logger"] = logger
17.     # log de début
18.     logger.write("début du calcul de l'impôt des contribuables\n")
19.     # on récupère la couche [dao]
20.     dao = config["layers"]["dao"]
21.     # on récupère les contribuables
22.     taxpayers = dao.get_taxpayers_data()["taxpayers"]
23.     # des contribuables ?
24.     if not taxpayers:
25.         raise ImpôtsError(36, f"Pas de contribuables valides dans le fichier {config['taxpayersFilename']}")
26.     # on récupère les données de l'administration fiscale
27.     admindata = dao.get_admindata()
28.     # calcul de l'impôt des contribuables par la couche [métier]
29.     métier = config['layers']['métier']
30.     for taxpayer in taxpayers:
31.         métier.calculate_tax(taxpayer, admindata)
32.     # on enregistre les résultats dans le fichier jSON
33.     dao.write_taxpayers_results(taxpayers)
34. # except BaseException as erreur:
35. #     # affichage de l'erreur
36. #     print(f"L'erreur suivante s'est produite : {erreur}")
37. finally:
38.     # on ferme le logueur
39.     if logger:
40.         # log de fin
41.         logger.write("fin du calcul de l'impôt des contribuables\n")
42.         # fermeture du logueur
43.         logger.close()
44.     # on a fini
45.     print("Travail terminé...")
```

- lignes 26-27 : on récupère les données de l'administration fiscale auprès du serveur ;
- lignes 28-31 : ensuite le calcul de l'impôt des contribuables est fait localement ;

29.9 Tests du client

Dans chacun des scripts **[main, main2]** on logue le début et la fin du script. On pourra ainsi calculer la durée d'exécution du script. Faisons quelques pronostics :

- le script **[main]** de la version précédente :
  - crée N threads qui s'exécutent simultanément ;
  - chaque thread traite un lot de contribuables dont il fait calculer l'impôt via une unique requête au serveur ;
  - parce que les N threads s'exécutent simultanément, la requête N+1 est lancée avant que la requête N ait reçu sa réponse. Ainsi les N requêtes coûtent plus cher qu'une unique requête mais probablement pas beaucoup plus. Il y a par ailleurs 11 (le nombre de contribuables) calculs métier sur le serveur ;
- le script **[main2]** de cette version :
  - fait une unique requête au serveur ;
  - fait 11 calculs métier localement sur le client ;

Les calculs métier auront la même durée que ce soit sur le serveur ou le client. La différence va alors se faire sur les requêtes. On peut s'attendre alors que la durée d'exécution de **[main]** soit légèrement supérieure à celle de **[main2]**.

On lance le serveur de la version 11, le SGBD et le serveur de mails **[hMailServer]**. Côté serveur, on met le paramètre **[sleep_time]** à zéro pour que les deux tests soient exécutés dans les mêmes conditions.

Exécution 1 [main]

L'exécution de **[main]** donne les logs suivants :

```
1.  2020-07-29 14:35:50.016079, MainThread : début du calcul de l'impôt des contribuables
2.  2020-07-29 14:35:50.016079, Thread-1 : début du calcul de l'impôt des 1 contribuables
3.  2020-07-29 14:35:50.016079, Thread-2 : début du calcul de l'impôt des 4 contribuables
4.  2020-07-29 14:35:50.016079, Thread-3 : début du calcul de l'impôt des 2 contribuables
5.  2020-07-29 14:35:50.016079, Thread-4 : début du calcul de l'impôt des 2 contribuables
6.  2020-07-29 14:35:50.024426, Thread-5 : début du calcul de l'impôt des 2 contribuables
7.  2020-07-29 14:35:50.050473, Thread-1 : {"réponse": {"results": [{"marié": "oui", "enfants": 2, "salaire":
    55555, "impôt": 2814, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0}]}}
8.  2020-07-29 14:35:50.050473, Thread-1 : fin du calcul de l'impôt des 1 contribuables
9.  2020-07-29 14:35:50.050473, Thread-3 : {"réponse": {"results": [{"marié": "oui", "enfants": 3, "salaire":
    100000, "impôt": 9200, "surcôte": 2180, "taux": 0.3, "décôte": 0, "réduction": 0}, {"marié": "oui",
    "enfants": 5, "salaire": 100000, "impôt": 4230, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0}]}}
10. 2020-07-29 14:35:50.051214, Thread-3 : fin du calcul de l'impôt des 2 contribuables
11. 2020-07-29 14:35:50.051214, Thread-5 : {"réponse": {"results": [{"marié": "non", "enfants": 0, "salaire":
    200000, "impôt": 64210, "surcôte": 7498, "taux": 0.45, "décôte": 0, "réduction": 0}, {"marié": "oui",
    "enfants": 3, "salaire": 200000, "impôt": 42842, "surcôte": 17283, "taux": 0.41, "décôte": 0, "réduction":
    0}]}}
12. 2020-07-29 14:35:50.051214, Thread-5 : fin du calcul de l'impôt des 2 contribuables
13. 2020-07-29 14:35:50.051214, Thread-2 : {"réponse": {"results": [{"marié": "oui", "enfants": 2, "salaire":
    50000, "impôt": 1384, "surcôte": 0, "taux": 0.14, "décôte": 384, "réduction": 347}, {"marié": "oui",
    "enfants": 3, "salaire": 50000, "impôt": 0, "surcôte": 0, "taux": 0.14, "décôte": 720, "réduction": 0},
    {"marié": "non", "enfants": 2, "salaire": 100000, "impôt": 19884, "surcôte": 4480, "taux": 0.41, "décôte":
    0, "réduction": 0}, {"marié": "non", "enfants": 3, "salaire": 100000, "impôt": 16782, "surcôte": 7176,
    "taux": 0.41, "décôte": 0, "réduction": 0}]}}
14. 2020-07-29 14:35:50.051214, Thread-2 : fin du calcul de l'impôt des 4 contribuables
15. 2020-07-29 14:35:50.051214, Thread-4 : {"réponse": {"results": [{"marié": "non", "enfants": 0, "salaire":
    100000, "impôt": 22986, "surcôte": 0, "taux": 0.41, "décôte": 0, "réduction": 0}, {"marié": "oui",
    "enfants": 2, "salaire": 30000, "impôt": 0, "surcôte": 0, "taux": 0.0, "décôte": 0, "réduction": 0}]}}
16. 2020-07-29 14:35:50.051214, Thread-4 : fin du calcul de l'impôt des 2 contribuables
17. 2020-07-29 14:35:50.051214, MainThread : fin du calcul de l'impôt des contribuables
```

La durée de l'exécution a été de **[051214-016079]** nanosecondes (ligne 17 – ligne 1), ç-à-d 35 millisecondes et 135 nanosecondes.

On voit qu'entre la 1ère demande faite au serveur et la dernière réponse reçue par le client, il y a la même durée **[051214-016079]** (ligne 15 – ligne 1), 35 millisecondes et 135 nanosecondes.

Exécution 2 [main2]

L'exécution de **[main2]** donne les logs suivants :

```
1.  2020-07-29 14:41:03.303520, MainThread : début du calcul de l'impôt des contribuables
2.  2020-07-29 14:41:03.345084, MainThread : {"réponse": {"result": {"limites": [9964.0, 27519.0, 73779.0,
    156244.0, 13500.0], "coeffr": [0.0, 0.14, 0.3, 0.41, 0.45], "coeffn": [0.0, 1394.96, 5798.0, 13913.7,
    20163.4], "plafond_decote_couple": 1970.0, "valeur_reduc_demi_part": 3797.0,
    "plafond_revenus_celibataire_pour_reduction": 21037.0, "plafond_qf_demi_part": 1551.0,
    "abattement_dixpourcent_max": 12502.0, "plafond_impot_celibataire_pour_decote": 1595.0,
    "plafond_decote_celibataire": 1196.0, "plafond_revenus_couple_pour_reduction": 42074.0, "id": 1,
    "abattement_dixpourcent_min": 437.0, "plafond_impot_couple_pour_decote": 2627.0}}}
3.  2020-07-29 14:41:03.349975, MainThread : fin du calcul de l'impôt des contribuables
```

La durée de l'exécution a été de **[349975-303520]** nanosecondes (ligne 3 - ligne 1), ç-à-d 46 millisecondes et 455 nanosecondes. De façon tout à fait inattendue **[main]** est plus rapide que **[main2]**.

On voit que l'unique requête de **[main2]** a duré **[345084-303520]** (ligne 2 – ligne 1), ç-à-d 41 millisecondes et 564 nanosecondes. Le calcul de l'impôt a ensuite duré **[349975-345084]** (ligne 3 – ligne 2) ç-à-d 4 millisecondes et 91 nanosecondes. C'est la requête HTTP qui fait la durée d'exécution. De façon surprenante, on voit ici que l'unique requête de **[main2]** a duré plus longtemps **[41 millisecondes]** que les quatre requêtes simultanées de **[main]** **[35 millisecondes]**.

Côté serveur, les logs sont les suivants :

```
1.  2020-07-29 14:35:27.047721, MainThread : [serveur] démarrage du serveur
2.  2020-07-29 14:35:27.140927, MainThread : [serveur] connexion à la base de données réussie
3.  2020-07-29 14:35:28.790716, MainThread : [serveur] démarrage du serveur
4.  2020-07-29 14:35:28.847518, MainThread : [serveur] connexion à la base de données réussie
```

```
5.  2020-07-29 14:35:50.039178, Thread-2 : [index] requête : <Request 'http://127.0.0.1:5000/calculate-tax'
    [POST]>
6.  2020-07-29 14:35:50.039178, Thread-3 : [index] requête : <Request 'http://127.0.0.1:5000/calculate-tax'
    [POST]>
7.  2020-07-29 14:35:50.043220, Thread-4 : [index] requête : <Request 'http://127.0.0.1:5000/calculate-tax'
    [POST]>
8.  2020-07-29 14:35:50.044307, Thread-5 : [index] requête : <Request 'http://127.0.0.1:5000/calculate-tax'
    [POST]>
9.  2020-07-29 14:35:50.045796, Thread-2 : [index] {'réponse': {'results': [{'marié': 'oui', 'enfants': 3,
    'salaire': 100000, 'impôt': 9200, 'surcôte': 2180, 'taux': 0.3, 'décôte': 0, 'réduction': 0}, {'marié':
    'oui', 'enfants': 5, 'salaire': 100000, 'impôt': 4230, 'surcôte': 0, 'taux': 0.14, 'décôte': 0,
    'réduction': 0}]}}
10. 2020-07-29 14:35:50.045796, Thread-3 : [index] {'réponse': {'results': [{'marié': 'oui', 'enfants': 2,
    'salaire': 55555, 'impôt': 2814, 'surcôte': 0, 'taux': 0.14, 'décôte': 0, 'réduction': 0}]}}
11. 2020-07-29 14:35:50.046825, Thread-6 : [index] requête : <Request 'http://127.0.0.1:5000/calculate-tax'
    [POST]>
12. 2020-07-29 14:35:50.046825, Thread-6 : [index] {'réponse': {'results': [{'marié': 'oui', 'enfants': 2,
    'salaire': 50000, 'impôt': 1384, 'surcôte': 0, 'taux': 0.14, 'décôte': 384, 'réduction': 347}, {'marié':
    'oui', 'enfants': 3, 'salaire': 50000, 'impôt': 0, 'surcôte': 0, 'taux': 0.14, 'décôte': 720, 'réduction':
    0}, {'marié': 'non', 'enfants': 2, 'salaire': 100000, 'impôt': 19884, 'surcôte': 4480, 'taux': 0.41,
    'décôte': 0, 'réduction': 0}, {'marié': 'non', 'enfants': 3, 'salaire': 100000, 'impôt': 16782, 'surcôte':
    7176, 'taux': 0.41, 'décôte': 0, 'réduction': 0}]}}
13. 2020-07-29 14:35:50.046825, Thread-4 : [index] {'réponse': {'results': [{'marié': 'non', 'enfants': 0,
    'salaire': 200000, 'impôt': 64210, 'surcôte': 7498, 'taux': 0.45, 'décôte': 0, 'réduction': 0}, {'marié':
    'oui', 'enfants': 3, 'salaire': 200000, 'impôt': 42842, 'surcôte': 17283, 'taux': 0.41, 'décôte': 0,
    'réduction': 0}]}}
14. 2020-07-29 14:35:50.046825, Thread-5 : [index] {'réponse': {'results': [{'marié': 'non', 'enfants': 0,
    'salaire': 100000, 'impôt': 22986, 'surcôte': 0, 'taux': 0.41, 'décôte': 0, 'réduction': 0}, {'marié':
    'oui', 'enfants': 2, 'salaire': 30000, 'impôt': 0, 'surcôte': 0, 'taux': 0.0, 'décôte': 0, 'réduction':
    0}]}}
15. 2020-07-29 14:41:03.341582, Thread-7 : [index] requête : <Request 'http://127.0.0.1:5000/get-admindata'
    [GET]>
16. 2020-07-29 14:41:03.341582, Thread-7 : [index] {'réponse': {'result': {'limites': [9964.0, 27519.0,
    73779.0, 156244.0, 13500.0], 'coeffr': [0.0, 0.14, 0.3, 0.41, 0.45], 'coeffn': [0.0, 1394.96, 5798.0,
    13913.7, 20163.4], 'plafond_decote_couple': 1970.0, 'valeur_reduc_demi_part': 3797.0,
    'plafond_revenus_celibataire_pour_reduction': 21037.0, 'plafond_qf_demi_part': 1551.0,
    'abattement_dixpourcent_max': 12502.0, 'plafond_impot_celibataire_pour_decote': 1595.0,
    'plafond_decote_celibataire': 1196.0, 'plafond_revenus_couple_pour_reduction': 42074.0, 'id': 1,
    'abattement_dixpourcent_min': 437.0, 'plafond_impot_couple_pour_decote': 2627.0}}}
```

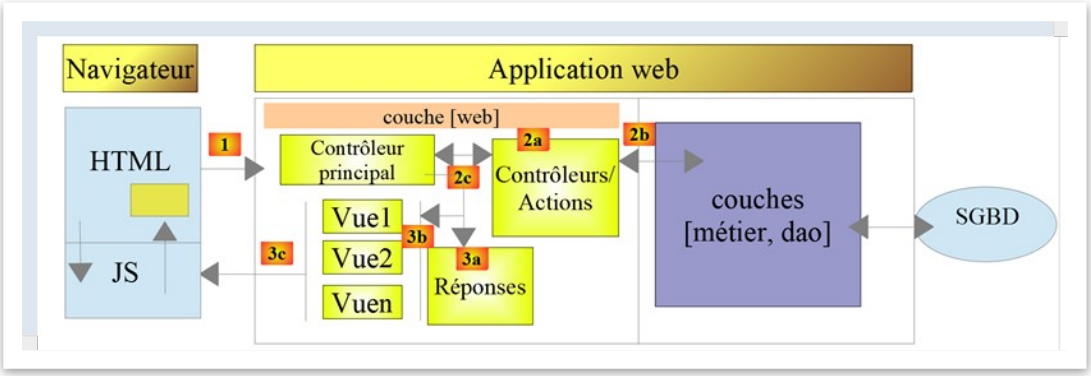
- ligne 5 : la 1ère requête du client **[main]** ;
- ligne 14 : la dernière réponse au client **[main]**. Il y a 6 millisecondes et 647 nanosecondes entre les deux ;
- lignes 15-16 : l'unique requête du client **[main2]**. La réponse est instantanée ;

30 Exercice d'application : version 12

Nous allons dans ce chapitre écrire une application web respectant l'architecture MVC (Modèle-Vue-Contrôleur). L'application pourra délivrer ses réponses dans trois formats : jSON, XML, HTML. Il y a un saut de complexité entre ce que nous allons faire maintenant et ce qui a été fait précédemment. Nous allons réutiliser la plupart des concepts vus jusqu'à maintenant et allons détailler toutes les étapes menant à l'application finale.

30.1 Architecture MVC

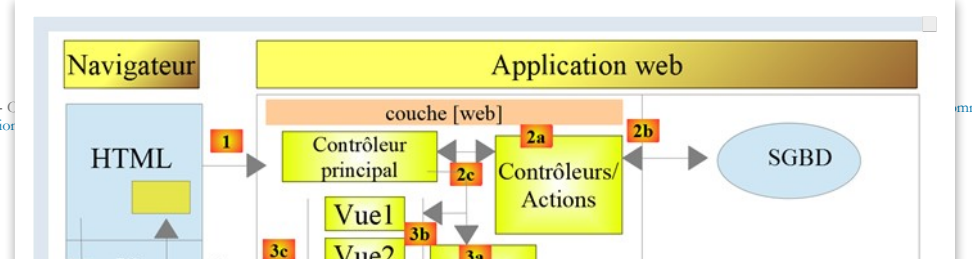
Nous allons implémenter le modèle d'architecture dit MVC (Modèle – Vue – Contrôleur) de la façon suivante :



Le traitement d'une demande d'un client se déroulera de la façon suivante :

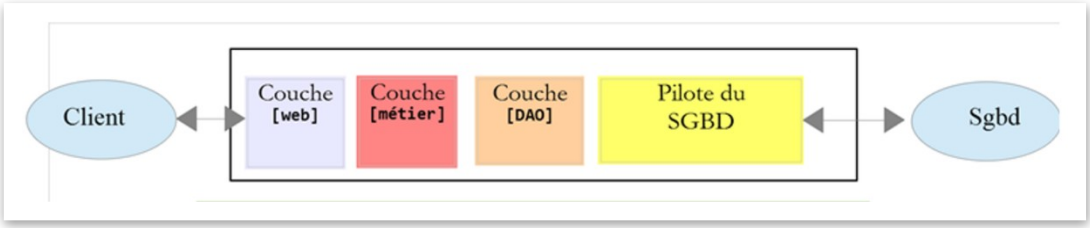
- 1 - demande
  Les URL demandées seront de la forme *http://machine:port/action/param1/param2/…* Le **`[Contrôleur principal]`** utilisera un fichier de configuration pour " router " la demande vers le bon contrôleur. Pour cela, il utilisera le champ **`[action]`** de l'URL. Le reste de l'URL **`[param1/param2/…]`** est formé de paramètres facultatifs qui seront transmis à l'action. Le **C** de MVC est ici la chaîne **`[Contrôleur principal, Contrôleur / Action]`**. Si aucun contrôleur ne peut traiter l'action demandée, le serveur web répondra que l'URL demandée n'a pas été trouvée.
- 2 - traitement
  - l'action choisie **`[2a]`** peut exploiter les paramètres *parami* que le **`[Contrôleur principal]`** lui a transmis. Ceux-ci pourront provenir de deux sources :
    - du chemin **`[/param1/param2/…]`** de l'URL,
    - de paramètres postés dans le corps de la requête du client ;
  - dans le traitement de la demande de l'utilisateur, l'action peut avoir besoin de la couche **`[métier] [2b]`**. Une fois la demande du client traitée, celle-ci peut appeler diverses réponses. Un exemple classique est :
    - une réponse d'erreur si la demande n'a pu être traitée correctement ;
    - une réponse de confirmation sinon ;
  - le **`[Contrôleur / Action]`** rendra sa réponse **`[2c]`** au contrôleur principal ainsi qu'un code d'état. Ces codes d'état représenteront de façon unique l'état dans lequel se trouve l'application. Ce sera soit un code de réussite, soit un code d'erreur ;
- 3 - réponse
  - selon que le client a demandé une réponse jSON, XML ou HTML, le **`[Contrôleur principal]`** instanciera **`[3a]`** le type de réponse appropriée et demandera à celle-ci d'envoyer la réponse au client. Le **`[Contrôleur principal]`** lui transmettra et la réponse et le code d'état fournis par le **`[Contrôleur / Action]`** qui a été exécuté ;
  - si la réponse souhaitée est de type jSON ou XML, la réponse sélectionnée mettra en forme la réponse du **`[Contrôleur / Action]`** qu'on lui a donnée et l'enverra **`[3c]`**. Le client capable d'exploiter cette réponse peut être un script console Python ou un script Javascript logé dans une page HTML ;
  - si la réponse souhaitée est de type HTML, la réponse sélectionnée sélectionnera **`[3b]`** une des **vues** HTML **`[Vuei]`** à l'aide du code d'état qu'on lui a donné. C'est le **V** de MVC. A un code d'état correspond une unique vue. Cette vue **V** va afficher la réponse du **`[Contrôleur / Action]`** qui a été exécuté. Elle habille avec du HTML, CSS, Javascript les données de cette réponse. On appelle ces données le **modèle de la vue**. C'est le **M** de MVC. Le client est alors le plus souvent un navigateur ;

Maintenant, précisons le lien entre architecture web MVC et architecture en couches. Selon la définition qu'on donne au **modèle**, ces deux concepts sont liés ou non. Prenons une application web MVC à une couche :



Ci-dessus, les **[Contrôleur / Action]** intègrent chacun une partie des couches **[métier]** et **[dao]**. Dans la couche **[web]** on a bien une architecture MVC mais l'ensemble de l'application n'a pas une architecture en couches. Ici il n'y a qu'une couche, la couche web, qui fait tout.

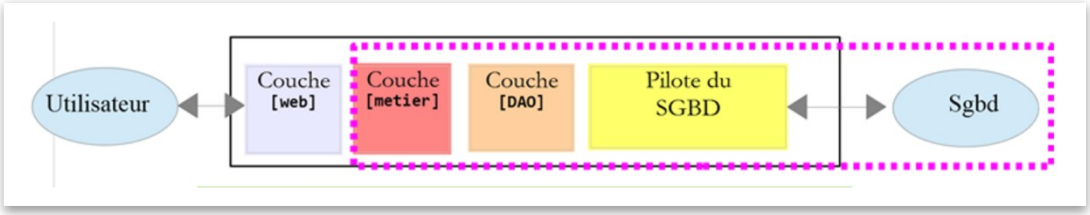
Maintenant, considérons une architecture web multicouche :



La couche **[web]** peut être implémentée sans suivre le modèle MVC. On a bien alors une architecture multicouche mais la couche web n'implémente pas le modèle MVC.

Par exemple, dans le monde .NET la couche **[web]** ci-dessus peut être implémentée avec ASP.NET MVC et on a alors une architecture en couches avec une couche **[web]** de type MVC. Ceci fait, on peut remplacer cette couche ASP.NET MVC par une couche ASP.NET classique (WebForms) tout en gardant le reste (métier, DAO, Pilote) **à l'identique**. On a alors une architecture en couches avec une couche **[web]** qui n'est plus de type MVC.

Dans MVC, nous avons dit que le modèle M était celui de la vue V, c.a.d. l'ensemble des données affichées par la vue V. Une autre définition du modèle M de MVC est donnée :



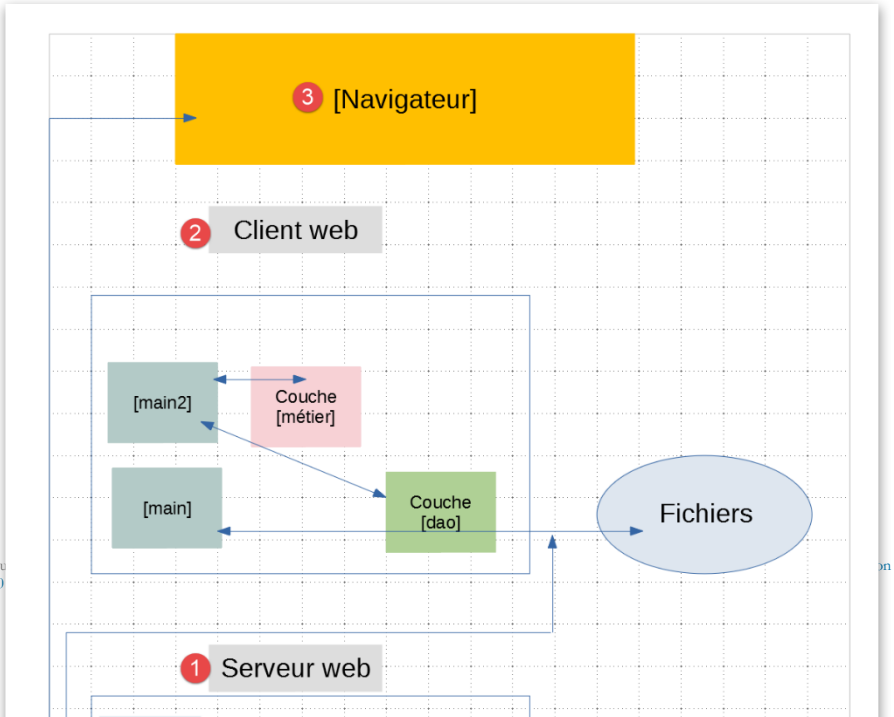
Beaucoup d'auteurs considèrent que ce qui est à droite de la couche **[web]** forme le modèle M du MVC. Pour éviter les ambigüités on peut parler :

- du **modèle du domaine** lorsqu'on désigne tout ce qui est à droite de la couche **[web]** ;
- du **modèle de la vue** lorsqu'on désigne les données affichées par une vue V ;

Dans ce qui suit, lorsque nous parlerons de modèle, il s'agita toujours du **modèle de la vue**.

30.2 Architecture de l'application client / serveur

L'application web aura l'architecture suivante :

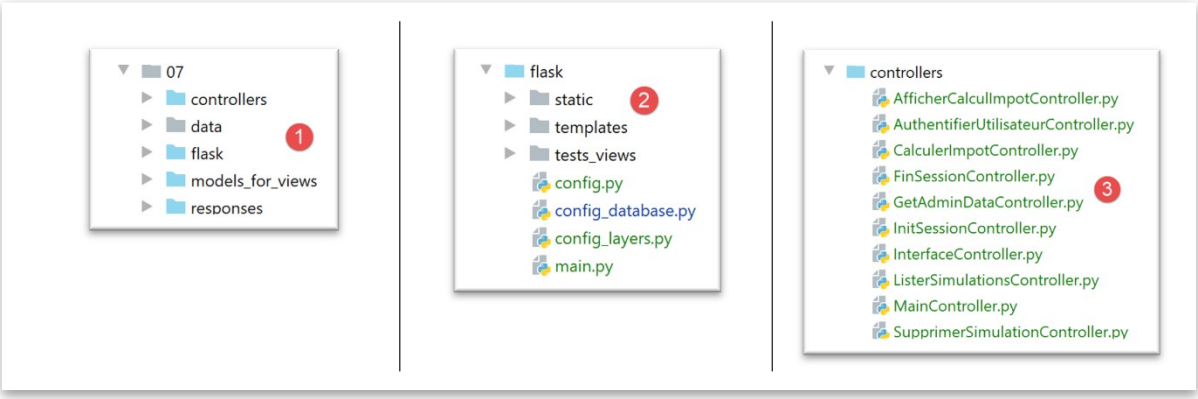


- en **[1]** le serveur web aura deux types de clients :
  - en **[2]**, un client console qui échangera du jSON et du XML avec le serveur ;
  - en **[3]**, un navigateur qui recevra du HTML du serveur et l'affichera ;
- le serveur web **[1]** conserve les couches **[métier]** et **[dao]** des versions précédentes ;
- le client web **[2]** évoluera pour tenir compte des nouvelles URL de service de l'application web ;
- l'application HTML afichée par le navigateur est à écrire complètement ;

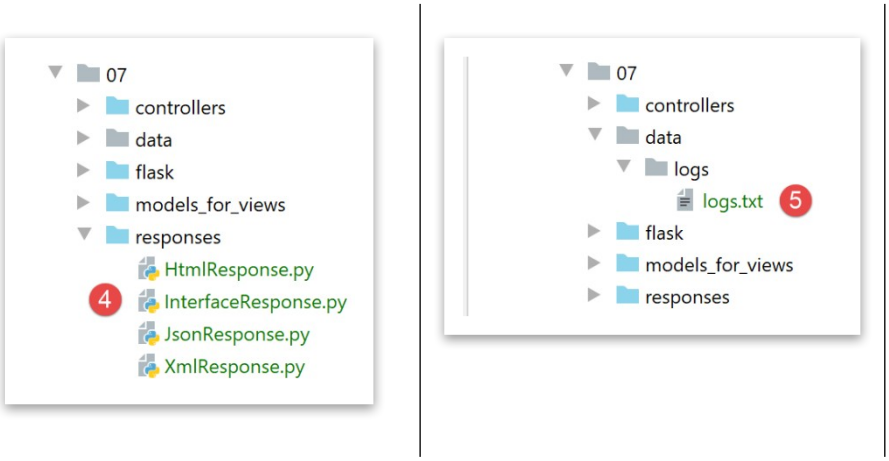
Nous allons développer l'application en plusieurs temps :

- nous allons développer la version jSON du serveur. Nous testerons les URL de service du serveur les unes après les autres avec un client Postman. Cette méthode nous permet de construire l'ossature du seveur web sans nous préoccuper des vues (=HTML) de l'application ;
- après avoir testé le serveur jSON avec Postman, nous le testerons avec un client console ;
- puis nous passerons à la version XML du serveur. Nous avons vu que le passage du jSON au XML était trivial ;
- enfin nous passerons à la version HTML du serveur. Nous construirons une architecture MVC et définirons les vues à afficher. L'application HTML sera testée à la fois avec le client Postman et un navigateur classique ;

30.3 L'arborescence du code du serveur

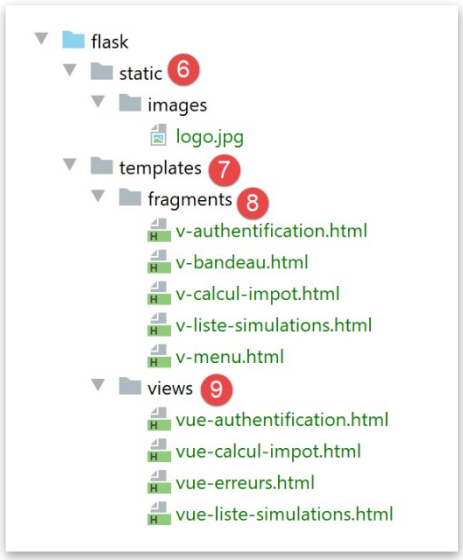


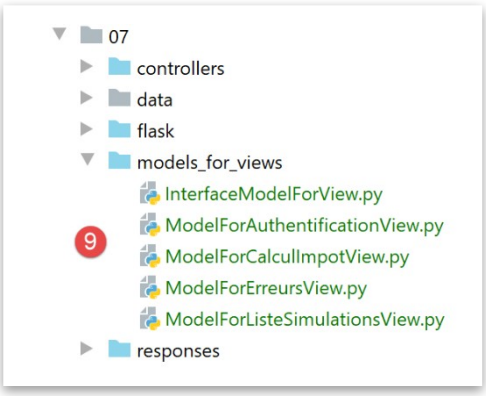
- en **[1 : le serveur web dans sa globalité ;**
- **en [2]** : nous ignorerons pour l'instant les dossiers **[static, templates, tests_views]** qui concernent la version HTML du serveur. En-dehors de ce dossier nous trouverons le script principal **[main]** et sa configuration ;
- en **[3]**, les contrôleurs du serveur web. Ce seront des instances de classes ;



- en **[4]**, la réponse HTTP du serveur sera gérée par des classes ;
- en **[5]**, nous conservons le fichier de logs des serveurs précédents ;

Lorsque nous construirons la version HTML du serveur, d'autres dossiers interviendront :





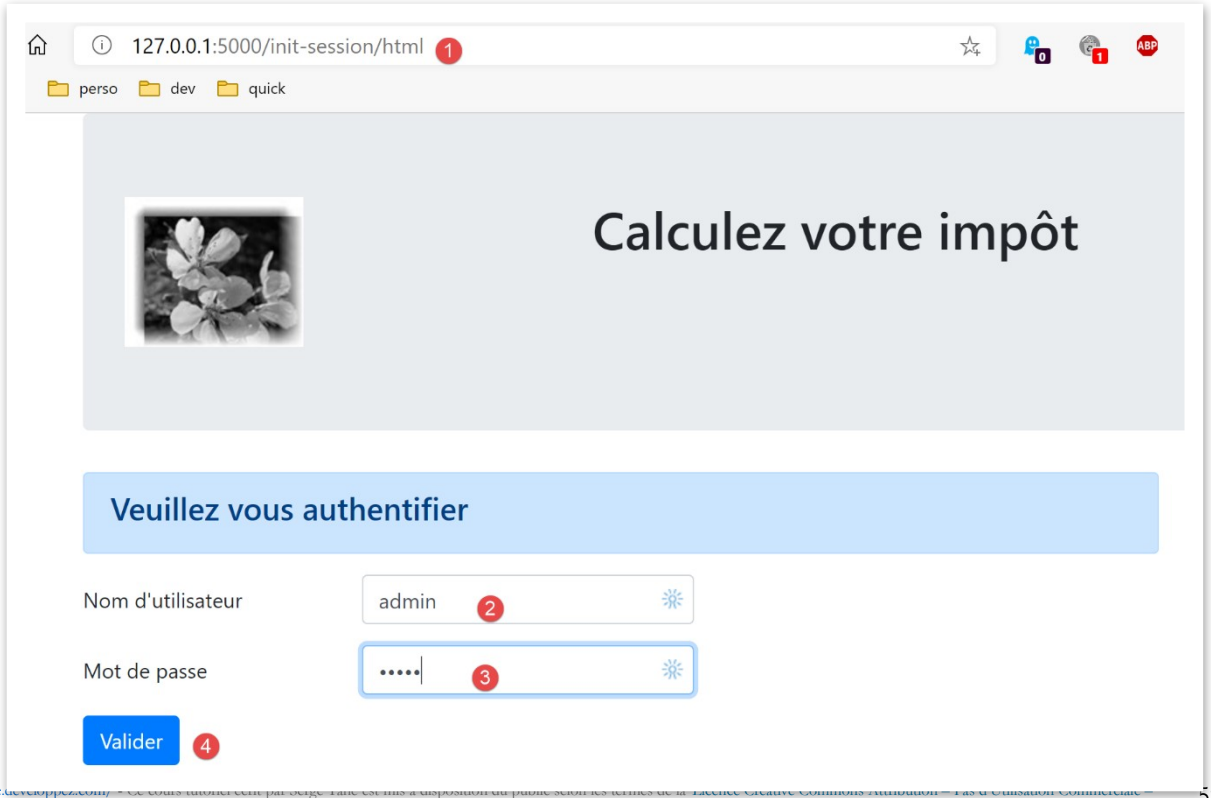
- en **[6]**, les éléments statiques de l'application HTML ;
- en **[7]**, les templates de l'application HTML décomposés en vues **[9]** et en fragments de vue **[8]** ;
- en **[9]**, les classes implémentant les modèles des vues ;

30.4 Les URL de service de l'application

Pour construire le serveur web, nous allons procéder de la façon suivante :

- à partir des vues de l'application HTML, nous allons définir les actions que doit implémenter l'application web. Nous allons ici utiliser les vues réelles mais ce pourrait être simplement des vues sur papier ;
- à partir de ces actions, nous allons définir les URL de service de l'application HTML ;
- nous allons implémenter ces URL de service avec un serveur délivrant du jSON. Cela permet de définir l'ossature du serveur web sans se préoccuper des pages HTML à délivrer. Nous testerons ces URL de service avec Postman ;
- nous testerons ensuite notre serveur jSON avec un client console ;
- une fois que le serveur jSON aura été validé, nous passerons à l'ériture de l'application HTML ;

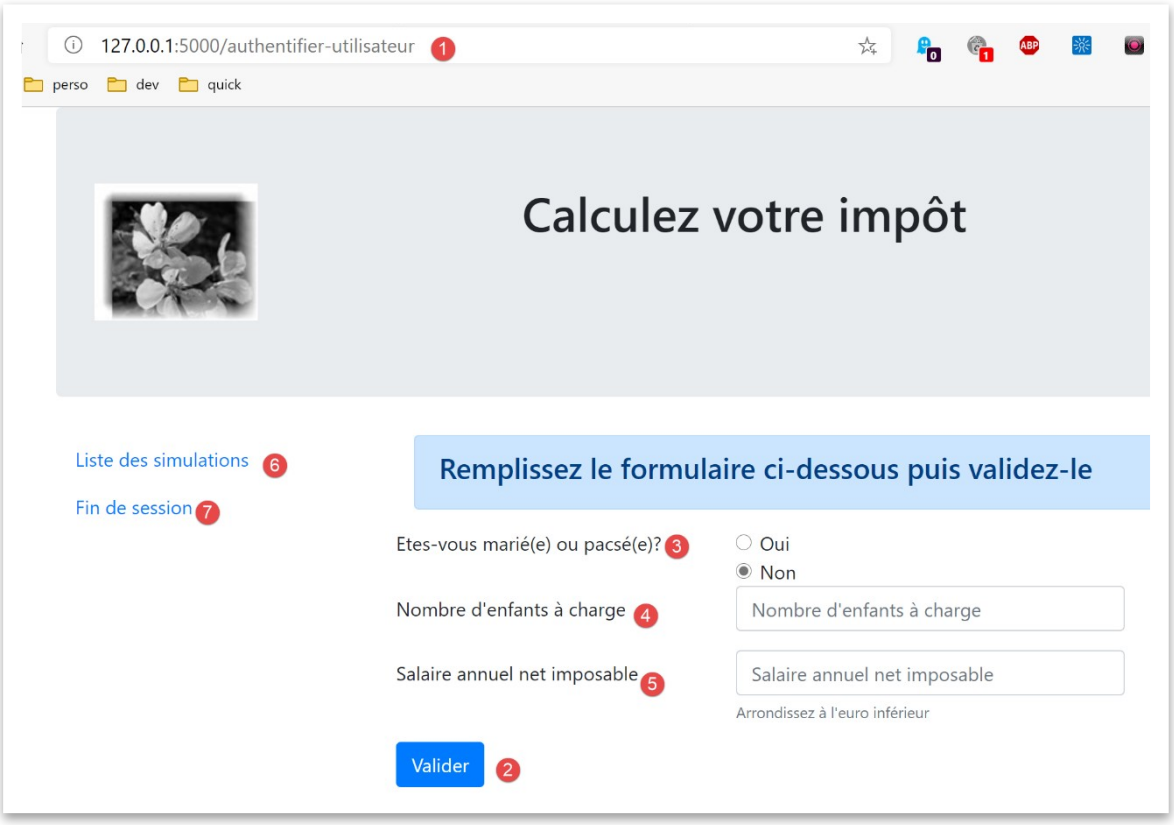
La 1ère vue sera la vue d'authentification :



https://tahe.developpez.com/ Ce cours tutoriel écrit par Serge Tahé est mis à disposition du public selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Partage dans les Mêmes Conditions 3.0 non transposé.

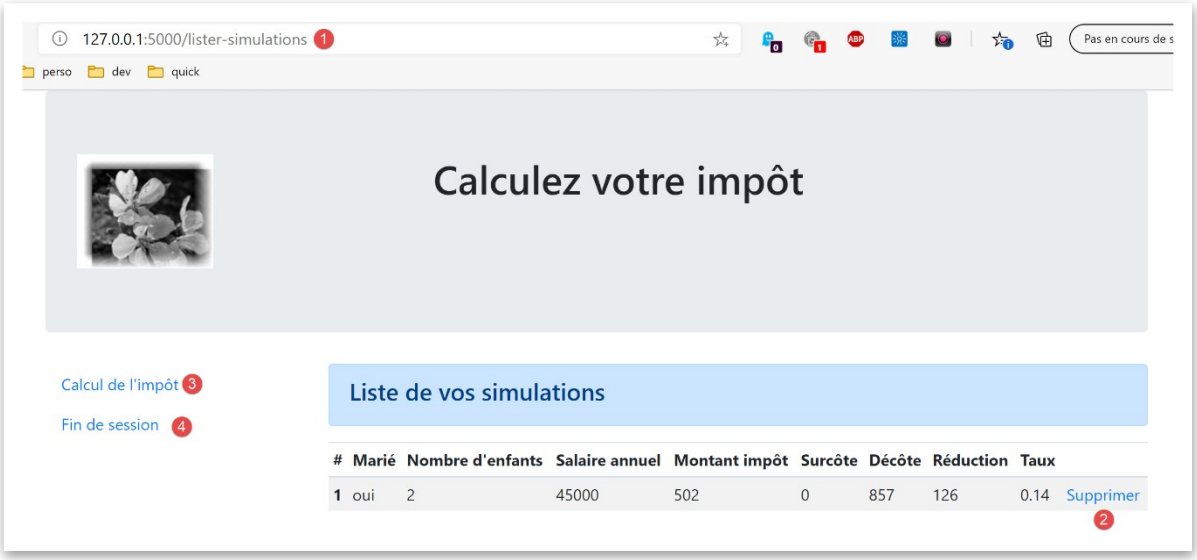
- le clic sur le bouton **[Valider]** va déclencher l'action **[authentifier-utilisateur]** avec deux paramètres postés **[2-3]** ;

La vue du calcul de l'impôt :



- en **[1]**, l'action **[authentifier-utilisateur]** qui a amené à cette vue ;
- en **[2]**, le clic sur le bouton **[Valider]** déclenche l'exécution de l'action **[calculer-impot]** avec trois paramètres postés **[2-5]** ;
- le clic sur le lien **[6]** déclenche l'action **[lister-simulations]** sans paramètres ;
- le clic sur le lien **[7]** déclenche l'action **[fin-session]** sans paramètres ;

La 3ième vue est celle des simulations faites par l'utilisateur authentifié :



- en **[3]**, l'action **[lister-simulations]** qui a amené à cette vue ;
- en **[2]**, un clic sur le lien **[Supprimer]** déclenche l'action **[supprimer-simulation]** avec un paramètre, le n° de la simulation à supprimer dans la liste ;
- un clic sur le lien **[3]** déclenche l'action **[afficher-calcul-impot]** sans paramètres qui réaffiche la vue du calcul de l'impôt ;

- un clic sur le lien **[4]** déclenche l'action **[fin-session]** sans paramètres ;

Avec ces premières informations, nous pouvons définir les différentes URL de service du serveur :

Action	Rôle	Contexte d'exécution
/init-session	Sert à fixer le type (json, xml, html) des réponses souhaitées	Requête **GET** Peut être émise à tout moment
/authentifier-utilisateur	Autorise ou non un utilisateur à se connecter	Requête **POST.** La requête doit avoir deux paramètres postés **[user, password]** Ne peut être émise que si le type de la session (json, xml, html) est connu
/calculer-impot	Fait une simulation de calcul d'impôt	Requête **POST.** La requête doit avoir trois paramètres postés **[marié, enfants, salaire]** Ne peut être émise que si le type de la session (json, xml, html) est connu et l'utilisateur authentifié
/lister-simulations	Demande à voir la liste des simulations opérées depuis le début de la session	Requête **GET.** Ne peut être émise que si le type de la session (json, xml, html) est connu et l'utilisateur authentifié
/supprimer-simulation/numéro	Supprime une simulation de la liste des simulations	Requête **GET.** Ne peut être émise que si le type de la session (json, xml, html) est connu et l'utilisateur authentifié
/afficher-calcul-impot	Affiche la page HTML du calul de l'impôt	Requête **GET**. Ne peut être émise que si le type de la session (json, xml, html) est connu et l'utilisateur authentifié
/fin-session	Termine la session de simulations.	Techniquement l'ancienne session web est supprimée et une nouvelle session est créée Ne peut être émise que si le type de la session (json, xml, html) est connu et l'utilisateur authentifié

Ces différentes URL de service seront utilisées aussi bien pour le serveur HTML que pour les serveurs jSON ou XML. Deux URL ne seront utilisées que pour ces deux derniers serveurs : ce sont les URL de la version précédente du client / serveur web que nous reprenons ici :

Action	Rôle	Contexte d'exécution
/get-admindata	Rend les données fiscales permettant le calcul de l'impôt	Requête GET. N'est utilisée que si le type de la session est json ou xml. L'utilisateur doit être authentifié
/calculer-impots	Fait le calcul de l'impôt d'une liste de contribuables postés en jSON	Requête GET. N'est utilisée que si le type de la session est json ou xml. L'utilisateur doit être authentifié

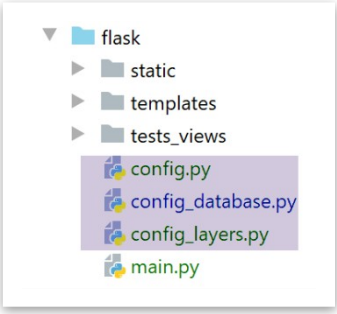
Tous les contrôleurs associés à ces actions procèderont de la même façon :

- ils vérifieront leurs paramètres. Ceux-ci sont trouvés dans l'objet :
  - **[request.path]** pour les paramètres présents dans l'URL sous la forme **[/action/param1/param2/…]** ;
  - dans l'objet **[request.form]** pour ceux qui sont transmis en **[x-www-form-urlencoded]** dans le corps de la requête ;
  - dans l'objet **[request.data]** pour ceux qui sont transmis en jSON dans le corps de la requête ;
- un contrôleur s'apparente à une fonction ou méthode qui vérifie la validité de ses paramètres. Pour le contrôleur c'est cependant un peu plus compliqué :
  - les paramètres attendus peuvent être absents ;
  - les paramètres récupérés par le contrôleur sont des chaînes de caractères. Si le paramètre attendu est un nombre, alors le contrôleur doit vérifier que la chaîne de caractères du paramètre est bien celle d'un nombre ;
  - une fois vérifié, que les paramètres attendus sont présents et syntaxiquement corrects, il faut vérifier qu'ils sont valides dans le contexte d'exécution du moment. Ce contexte est présent dans la session. L'exemple de l'authentification est un exemple de contexte d'exécution. Certaines actions ne doivent être traitées qu'une fois le client authentifié. Généralement, une clé dans la session indique si cette authentification a eu lieu ou pas ;
  - une fois, les vérifications précédentes faites, le contrôleur secondaire peut travailler. Ce travail de vérification des paramètres est très important. On ne peut pas accepter qu'un client nous envoie n'importe quoi à n'importe quel moment de la vie de l'application. On doit contrôler totalement la vie de celle-ci ;
  - une fois son travail fait, le contrôleur secondaire rend un dictionnaire avec les clés **[action, état, réponse]** au contrôleur principal qui l'a appelé :
    - **[action]** est l'action qui vient d'être exécutée ;
    - **[état]** est un nombre à trois chiffres qui indique le résultat du traitement de l'action :
      - **[x00]** signalera une réussite du traitement ;

- **[x01]** signalera un échec du traitement ;
- **[réponse]** est le dictionnaire des résultats sous la forme **{'réponse':objet}**. L'objet aura des structures différentes selon l'action traitée ;

Nous allons maintenant passer en revue les différents contrôleurs ou ce qui revient au même les différentes actions que ces contrôleurs traitent et qui rythment la vie de l'application web.

30.5 Configuration du serveur



La configuration de la base de données **[config_database]** ainsi que celle des couches du serveur **[config_layers]** sont identiques à celles des versions précédentes. Le fichier **[config]** voit apparaître de nouvelles informations :

```
1.  def configure(config: dict) -> dict:
2.      import os
3.
4.      # étape 1 ------
5.
6.      # dossier de ce fichier
7.      script_dir = os.path.dirname(os.path.abspath(__file__))
8.
9.      # chemin racine
10.     root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
11.
12.     # dépendances
13.     absolute_dependencies = [
14.         # dossiers du projet
15.         # BaseEntity, MyException
16.         f"{root_dir}/classes/02/entities",
17.         # InterfaceImpôtsDao, InterfaceImpôtsMétier, InterfaceImpôtsUi
18.         f"{root_dir}/impots/v04/interfaces",
19.         # AbstractImpôtsdao, ImpôtsConsole, ImpôtsMétier
20.         f"{root_dir}/impots/v04/services",
21.         # ImpotsDaoWithAdminDataInDatabase
22.         f"{root_dir}/impots/v05/services",
23.         # AdminData, ImpôtsError, TaxPayer
24.         f"{root_dir}/impots/v04/entities",
25.         # Constantes, tranches
26.         f"{root_dir}/impots/v05/entities",
27.         # Logger, SendAdminMail
28.         f"{root_dir}/impots/http-servers/02/utilities",
29.         # scripts [config_database, config_layers]
30.         script_dir,
31.         # contrôleurs
32.         f"{script_dir}/../controllers",
33.         # réponses HTTP
34.         f"{script_dir}/../responses",
35.         # modèles des vues
36.         f"{script_dir}/../models_for_views",
37.     ]
38.
39.     # on fixe le syspath
40.     from myutils import set_syspath
41.     set_syspath(absolute_dependencies)
42.
43.     # dépendances du serveur web
44.
45.     # les contrôleurs
46.     from AfficherCalculImpotController import AfficherCalculImpotController
47.     from AuthentifierUtilisateurController import AuthentifierUtilisateurController
```

```
48.     from CalculerImpotController import CalculerImpotController
49.     from CalculerImpotsController import CalculerImpotsController
50.     from FinSessionController import FinSessionController
51.     from GetAdminDataController import GetAdminDataController
52.     from InitSessionController import InitSessionController
53.     from ListerSimulationsController import ListerSimulationsController
54.     from MainController import MainController
55.     from SupprimerSimulationController import SupprimerSimulationController
56.
57.     # les réponses HTTP
58.     from HtmlResponse import HtmlResponse
59.     from JsonResponse import JsonResponse
60.     from XmlResponse import XmlResponse
61.
62.     # les modèles des vues
63.     from ModelForAuthentificationView import ModelForAuthentificationView
64.     from ModelForCalculImpotView import ModelForCalculImpotView
65.     from ModelForErreursView import ModelForErreursView
66.     from ModelForListeSimulationsView import ModelForListeSimulationsView
67.
68.     # étape 2 ------
69.     # configuration de l'application
70.     config.update({
71.         # utilisateurs autorisés à utiliser l'application
72.         "users": [
73.             {
74.                 "login": "admin",
75.                 "password": "admin"
76.             }
77.         ],
78.
79.         # fichier de logs
80.         "logsFilename": f"{script_dir}/../data/logs/logs.txt",
81.
82.         # config serveur SMTP
83.         "adminMail": {
84.             # serveur SMTP
85.             "smtp-server": "localhost",
86.             # port du serveur SMTP
87.             "smtp-port": "25",
88.             # administrateur
89.             "from": "guest@localhost.com",
90.             "to": "guest@localhost.com",
91.             # sujet du mail
92.             "subject": "plantage du serveur de calcul d'impôts",
93.             # tls à True si le serveur SMTP requiert une autorisation, à False sinon
94.             "tls": False
95.         },
96.
97.         # durée pause thread en secondes
98.         "sleep_time": 0,
99.
100.        # actions autorisées et leurs contrôleurs
101.        "controllers": {
102.            # initialisation d'une session de calcul
103.            "init-session": InitSessionController(),
104.            # authentification d'un utilisateur
105.            "authentifier-utilisateur": AuthentifierUtilisateurController(),
106.            # calcul de l'impôt en mode individuel
107.            "calculer-impot": CalculerImpotController(),
108.            # calcul de l'impôt en mode lots
109.            "calculer-impots": CalculerImpotsController(),
110.            # liste des simulations
111.            "lister-simulations": ListerSimulationsController(),
112.            # suppression d'une simulation
113.            "supprimer-simulation": SupprimerSimulationController(),
114.            # fin de la session de calcul
115.            "fin-session": FinSessionController(),
116.            # affichage de la vue de calcul de l'impôt
117.            "afficher-calcul-impot": AfficherCalculImpotController(),
118.            # obtention des données de l'administration fiscale
119.            "get-admindata": GetAdminDataController(),
120.            # main controller
121.            "main-controller": MainController()
122.        },
123.
124.        # les différents types de réponse (json, xml, html)
```

```
        "responses": {
            "json": JsonResponse(),
            "html": HtmlResponse(),
            "xml": XmlResponse()
        },

        # les vues HTML et leurs modèles dépendent de l'état rendu par le contrôleur
        "views": [
            {
                # vue d'authentification
                "états": [
                    # /init-session réussite
                    700,
                    # /authentifier-utilisateur échec
                    201
                ],
                "view_name": "views/vue-authentification.html",
                "model_for_view": ModelForAuthentificationView()
            },
            {
                # vue du calcul de l'impôt
                "états": [
                    # /authentifier-utilisateur réussite
                    200,
                    # /calculer-impot réussite
                    300,
                    # /calculer-impot échec
                    301,
                    # /afficher-calcul-impot
                    800
                ],
                "view_name": "views/vue-calcul-impot.html",
                "model_for_view": ModelForCalculImpotView()
            },
            {
                # vue de la liste des simulations
                "états": [
                    # /lister-simulations
                    500,
                    # /supprimer-simulation
                    600
                ],
                "view_name": "views/vue-liste-simulations.html",
                "model_for_view": ModelForListeSimulationsView()
            }
        ],

        # vue des erreurs inattendues
        "view-erreurs": {
            "view_name": "views/vue-erreurs.html",
            "model_for_view": ModelForErreursView()
        },

        # redirections
        "redirections": [
            {
                "états": [
                    400,  # /fin-session réussi
                ],
                # redirection vers
                "to": "/init-session/html",
            }
        ],
    }
    )

    # étape 3 ------
    # configuration de la base de données
    import config_database
    config["database"] = config_database.configure(config)

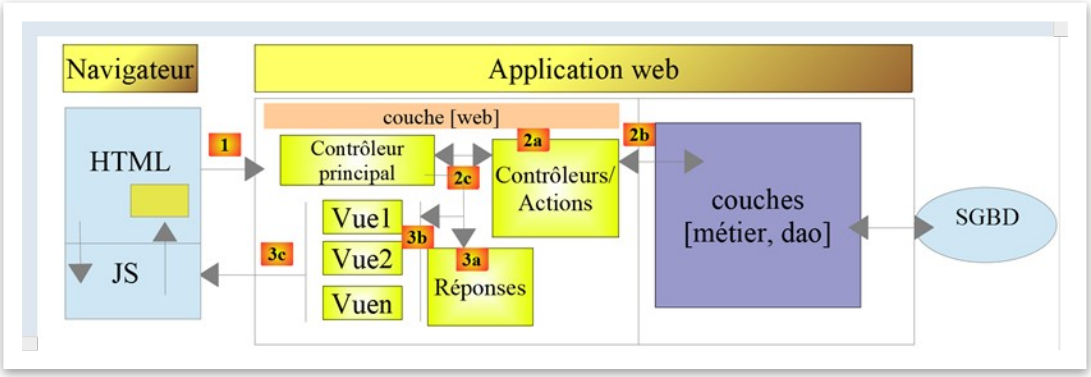
    # étape 4 ------
    # instanciation des couches de l'application
    import config_layers
    config['layers'] = config_layers.configure(config)

    # on rend la configuration
```

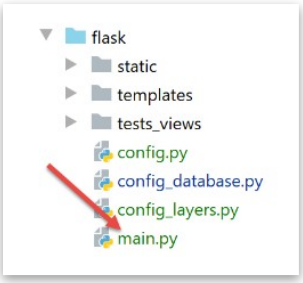
```
202.     return config
```

- jusqu'à la ligne 41, on retrouve des choses classiques ;
- lignes 43-66 : arrivé à la ligne 43, le Python Path du serveur est défini. On peut alors importer les dépendances du projet :
  - lignes 45-55 : la liste des contrôleurs ;
  - lignes 57-60 : la liste des réponses HTTP ;
  - lignes 62-66 : la liste des modèles de vues ;
- lignes 68-189 : la configuration de l'application avec une série de constantes ;
  - lignes 71-98 : nous connaissons déà ces lignes rencontrées dans les versions précédentes ;
  - lignes 101-122 : le dictionnaire des contrôleurs :
    - les clés sont les noms des actions ;
    - les valeurs sont une instance du contrôleur qui doit gérer cette action. Chaque contrôleur n'est instancié qu'en un unique exemplaire (singleton). La même instance sera exécutée par différents threads du serveur. Il faudra donc veiller aux données partagées que chaque contrôleur pourrait vouloir modifier ;
  - lignes 125-129 : le dictionnaire des trois réponses HTTP possibles :
    - les clés sont le type de réponse souhaité par le client (jSON, xml, html) ;
    - les valeurs sont une instance de la réponse HTTP. Chaque générateur de réponse n'est instancié qu'en un unique exemplaire (singleton). Le même générateur sera exécuté par différents threads du serveur. Il faudra donc veiller aux données partagées que chaque générateur pourrait vouloir modifier ;
  - lignbes 132-186 : configuration des vues HTML. Pour l'instant, on ignore ces lignes ;
- lignes 191-202 : nous avons déjà rencontré ces lignes dans les versions précédentes ;

30.6 Cheminement d'une requête client au sein du serveur



Nous allons suivre le cheminement d'une requête client arrivant sur le serveur jusqu'à la réponse HTTP envoyée en retour. Elle suit le cheminement du serveur MVC.



30.6.1 Le script `[main]`

Le script **`[main]`** est identique en de nombreux points à celui des versions précédentes. Nous le donnons néanmoins en entier pour repartir sur de bonnes bases :

```
1.  # on attend un paramètre mysql ou pgres
2.  import sys
3.
4.  syntaxe = f"{sys.argv[0]} mysql / pgres"
5.  erreur = len(sys.argv) != 2
6.  if not erreur:
7.      sgbd = sys.argv[1].lower()
8.      erreur = sgbd != "mysql" and sgbd != "pgres"
9.  if erreur:
```

```
10.     print(f"syntaxe : {syntaxe}")
11.     sys.exit()
12. 
13. # on configure l'application
14. import config
15. config = config.configure({'sgbd': sgbd})
16. 
17. # dépendances
18. from flask import request, Flask, session, url_for, redirect
19. from flask_api import status
20. from SendAdminMail import SendAdminMail
21. from myutils import json_response
22. from Logger import Logger
23. import threading
24. import time
25. from random import randint
26. from ImpôtsError import ImpôtsError
27. import os
28. 
29. # envoi d'un mail à l'administrateur
30. def send_adminmail(config: dict, message: str):
31.     # on envoie un mail à l'administrateur de l'application
32.     config_mail = config["adminMail"]
33.     config_mail["logger"] = config['logger']
34.     SendAdminMail.send(config_mail, message)
35. 
36. # vérification du fichier de logs
37. logger = None
38. erreur = False
39. message_erreur = None
40. try:
41.     # logueur
42.     logger = Logger(config["logsFilename"])
43. except BaseException as exception:
44.     # log console
45.     print(f"L'erreur suivante s'est produite : {exception}")
46.     # on note l'erreur
47.     erreur = True
48.     message_erreur = f"{exception}"
49. # on mémorise le logueur dans la config
50. config['logger'] = logger
51. # gestion de l'erreur
52. if erreur:
53.     # mail à l'administrateur
54.     send_adminmail(config, message_erreur)
55.     # fin de l'application
56.     sys.exit(1)
57. 
58. # log de démarrage
59. log = "[serveur] démarrage du serveur"
60. logger.write(f"{log}\n")
61. print(log)
62. 
63. # récupération des données de l'administration fiscale
64. erreur = False
65. try:
66.     # admindata sera une donnée de portée application en lecture seule
67.     config["admindata"] = config["layers"]["dao"].get_admindata().asdict()
68.     # log de réussite
69.     logger.write("[serveur] connexion à la base de données réussie\n")
70. except ImpôtsError as ex:
71.     # on note l'erreur
72.     erreur = True
73.     # log d'erreur
74.     log = f"L'erreur suivante s'est produite : {ex}"
75.     # console
76.     print(log)
77.     # fichier de logs
78.     logger.write(f"{log}\n")
79.     # mail à l'administrateur
80.     send_adminmail(config, log)
81. 
82. # le thread principal n'a plus besoin du logger
83. logger.close()
84. 
85. # s'il y a eu erreur on s'arrête
86. if erreur:
```

```
87.     sys.exit(2)
88.
89. # application Flask
90. app = Flask(__name__, template_folder="templates", static_folder="static")
91. # clé secrète de la session
92. app.secret_key = os.urandom(12).hex()
93.
94. # le front controller
95. def front_controller() -> tuple:
96.     # on traite la requête
97.     logger = None
98.     …
99.
100.@app.route('/', methods=['GET'])
101.def index() -> tuple:
102.    # redirection vers /init-session/html
103.    return redirect(url_for("init_session", type_response="html"), status.HTTP_302_FOUND)
104.
105.# init-session
106.@app.route('/init-session/<string:type_response>', methods=['GET'])
107.def init_session(type_response: str) -> tuple:
108.    # on exécute le contrôleur associé à l'action
109.    return front_controller()
110.
111.# authentifier-utilisateur
112.@app.route('/authentifier-utilisateur', methods=['POST'])
113.def authentifier_utilisateur() -> tuple:
114.    # on exécute le contrôleur associé à l'action
115.    return front_controller()
116.
117.# calculer-impot
118.@app.route('/calculer-impot', methods=['POST'])
119.def calculer_impot() -> tuple:
120.    # on exécute le contrôleur associé à l'action
121.    return front_controller()
122.
123.# lister-simulations
124.@app.route('/lister-simulations', methods=['GET'])
125.def lister_simulations() -> tuple:
126.    # on exécute le contrôleur associé à l'action
127.    return front_controller()
128.
129.# supprimer-simulation
130.@app.route('/supprimer-simulation/<int:numero>', methods=['GET'])
131.def supprimer_simulation(numero: int) -> tuple:
132.    # on exécute le contrôleur associé à l'action
133.    return front_controller()
134.
135.# fin-session
136.@app.route('/fin-session', methods=['GET'])
137.def fin_session() -> tuple:
138.    # on exécute le contrôleur associé à l'action
139.    return front_controller()
140.
141.# afficher-calcul-impot
142.@app.route('/afficher-calcul-impot', methods=['GET'])
143.def afficher_calcul_impot() -> tuple:
144.    # on exécute le contrôleur associé à l'action
145.    return front_controller()
146.
147.# get-admindata
148.@app.route('/get-admindata/<int:numero>', methods=['GET'])
149.def get_admindata() -> tuple:
150.    # on exécute le contrôleur associé à l'action
151.    return front_controller()
152.
153.# main uniquement
154.if __name__ == '__main__':
155.    # on lance le serveur
156.    app.config.update(ENV="development", DEBUG=True)
157.    app.run(threaded=True)
```

- lignes 1-92 : toutes ces lignes ont été déjà rencontrées et expliquées ;
- ligne 92 : le serveur va gérer une session. Il nous faut donc une clé secrète. Nous mettrons pour chaque utilisateur deux informations dans la session :
    - si l'utilisateur s'est correctement authentifié ;

    - à chaque fois qu'il fera un calcul d'impôt, les résultats de ce calcul seront placés dans une liste qu'on appellera la liste des simulations de l'utilisateur. Cette liste sera placée en session ;
- lignes 100-151 : la liste des URL de service du serveur. Les fonctions associées servent de filtre : toutes les URL non présentes dans cette liste seront rejetées par le serveur Flask avec l'erreur **[404 NOT FOUND]**. Une fois passée ce filtrage, la requête est systématiquement transmise à un 'Front Controller' implémenté par la fonction **[front_controller]** des lignes 94-98 que nous allons bientôt présenter ;
- lignes 100-103 : gestion de la route **[/]**. Le point d'entrée de l'application web sera l'URL de la ligne 107. Aussi ligne 103, nous redirigeons le client vers cette URL :
    - la fonction **[url_for]** est importée ligne 18. Elle a ici deux paramètres :
        - le 1er paramètre est le nom d'une des fonctions de routage, ici celle de la ligne 107. On voit que cette fonction attend un paramètre **[type_response]** qui est le type (json, xml, html) de réponse souhaité par le client ;
        - le 2ième paramètre reprend le nom du paramètre de la ligne 107, **[type_response]** et lui donne une valeur. S'il y avait d'autres paramètres, on répéterait l'opération pour chacun d'eux ;
        - elle rend l'URL associée à la fonction désignée par les deux paramètres qui lui ont été donnés. Ici cela donnera l'URL de la ligne 106 où le paramètre est remplacé par sa valeur **[/init-session/html]** ;
    - la fonction **[redirect]** a été importée ligne 18. Elle a pour rôle d'envoyer un entête HTTP de redirection au client :
        - le 1er paramètre est l'URL vers laquelle le client doit être redirigé ;
        - le 2ième paramètre est le code de statut de la réponse HTTP faite au client. Le code **[status.HTTP_302_FOUND]** correspond à une redirection HTTP ;

La fonction **[front_controller]** des lignes 94-98 fait les premiers traitements de la requête du client :

```
1.  # le front controller
2.  def front_controller() -> tuple:
3.      # on traite la requête
4.      logger = None
5.      try:
6.          # logger
7.          logger = Logger(config["logsFilename"])
8.          # on le mémorise dans une config associée au thread
9.          thread_config = {"logger": logger}
10.         thread_name = threading.current_thread().name
11.         config[thread_name] = {"config": thread_config}
12.         # on logue la requête
13.         logger.write(f"[ front_controller] requête : {request}\n")
14.         # on interrompt le thread si cela a été demandé
15.         sleep_time = config["sleep_time"]
16.         if sleep_time != 0:
17.             # la pause est aléatoire pour que certains threads soient interrompus et d'autres pas
18.             aléa = randint(0, 1)
19.             if aléa == 1:
20.                 # log avant pause
21.                 logger.write(f"[ front_controller] mis en pause du thread pendant {sleep_time} seconde(s)\n")
22.                 # pause
23.                 time.sleep(sleep_time)
24.         # on fait suivre la requête au contrôleur principal
25.         main_controller = config['controllers']["main-controller"]
26.         résultat, status_code = main_controller.execute(request, session, config)
27.         # on logue le résultat envoyé au client
28.         log = f"[front_controller] {résultat}\n"
29.         logger.write(log)
30.         # y-a-t-il eu une erreur fatale ?
31.         if status_code == status.HTTP_500_INTERNAL_SERVER_ERROR:
32.             # on envoie un mail à l'administrateur de l'application
33.             send_adminmail(config, log)
34.         # on détermine le type souhaité pour la réponse
35.         if session.get('typeResponse') is None:
36.             # le type de session n'a pas encore été établi - ce sera du jSON
37.             type_response = 'json'
38.         else:
39.             type_response = session['typeResponse']
40.         # on construit la réponse à envoyer
41.         response_builder = config["responses"][type_response]
42.         response, status_code = response_builder \
43.             .build_http_response(request, session, config, status_code, résultat)
44.         # on envoie la réponse
45.         return response, status_code
46.     except BaseException as erreur:
47.         # c'est une erreur inattendue - on logue l'erreur si c'est possible
48.         if logger:
49.             logger.write(f"[ front_controller] {erreur}")
50.         # on prépare la réponse au client
51.         résultat = {"réponse": {"erreurs": [f"{erreur}"]}}
52.         # on envoie lune réponse en jSON
53.         return json_response(résultat, status.HTTP_500_INTERNAL_SERVER_ERROR)
54.     finally:
55.         # on ferme le fichier de logs s'il a été ouvert
56.         if logger:
```

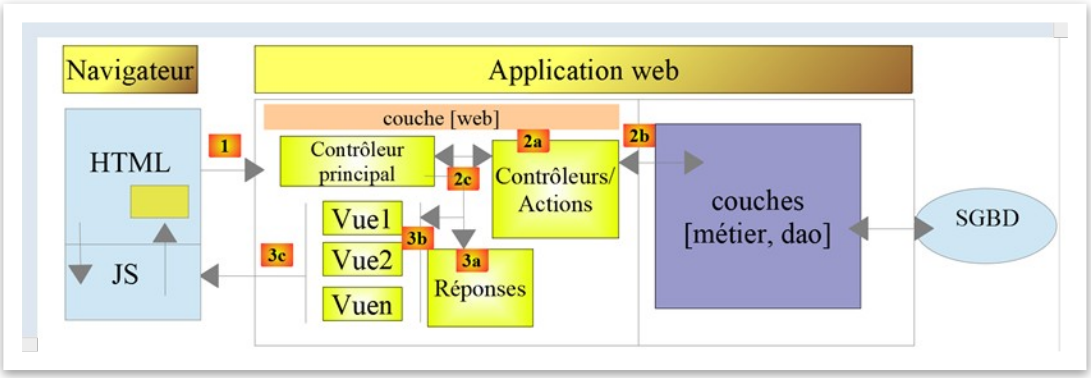
```
57.             logger.close()
```

- lignes 1-57 : nous connaissons ce code. C'était par exemple le code de la fonction appelée **[main]** dans le script **[main]** de la version précédente. Une seule chose est à noter, le contrôleur utilisé aux lignes 25-26 :
- ligne 25 : on récupère dans la configuration l'instance de contrôleur associée au nom **[main-controller]**. Il s'agit des lignes suivantes :

```
1.      # dépendances du serveur web
2.      # les contôleurs
3.      …
4.      from MainController import MainController
5.
6.      # actions autorisées et leurs contrôleurs
7.          "controllers": {
8.              …,
9.              # main controller
10.             "main-controller": MainController()
11.         },
```

  - ligne 10 ci-dessus, on remarquera qu'on récupère une instance de classe ;
- ligne 26 : on demande on contrôleur **[MainController]** de traiter la requête ;
- lignes 30-45 : la réponse rendue par le contrôleur **[MainController]** est envoyée au client. Nous reviendrons sur ces lignes un peu plus tard ;

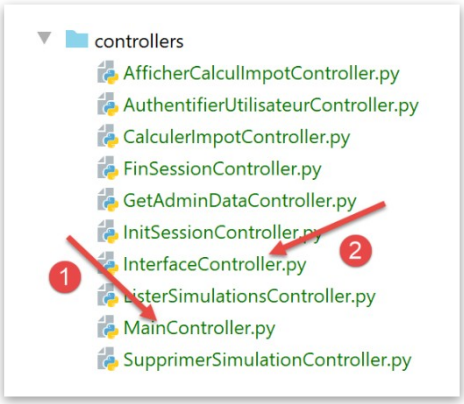
Le travail de la fonction **[front_controller]** puis de la classe **[MainController]** est de faire le travail commun à toutes les requêtes :



Dans le schéma ci-dessus, on en est toujours à la phase 1 du traitement de la requête. Le contrôleur principal **[MainController]** va poursuivre l'étape 1.

30.6.2 Le contrôleur principal [MainController]

Le contrôleur principal **[MainController]** continue le travail commencé par la fonction **[front_controller]** :



Tous les contrôleurs implémentent l'interface **[InterfaceController] [2]** suivante :

```
1.  from abc import ABC, abstractmethod
2.
3.  from werkzeug.local import LocalProxy
4.
```

```
5.  class InterfaceController(ABC):
6.
7.      @abstractmethod
8.      def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
9.          pass
```

- l'interface **[InterfaceController]** ne définit que l'unique méthode **[execute]** de la ligne 8. Cette méthode reçoit trois paramètres :
  - **[request]** : la requête du client ;
  - **[session]** : la session du client ;
  - **[config]** : la configuration de l'application ;

  La méthode **[execute]** rend un tuple de deux éléments :
  - le 1er est le dictionnaire des résultats sous la forme {'action': action, 'état':état, 'réponse':résultats} ;
  - le 2ième est le code de statut HTTP à rendre au client ;

Le contrôleur principal **[MainController] [1]** implémente l'interface **[InterfaceController]** de la façon suivante :

```
1.  # import des dépendances
2.
3.  from flask_api import status
4.  from werkzeug.local import LocalProxy
5.
6.  # contrôleurs de l'application web
7.  from InterfaceController import InterfaceController
8.
9.  class MainController(InterfaceController):
10.     def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
11.         # on récupères les éléments du path
12.         params = request.path.split('/')
13.         action = params[1]
14.
15.         # erreurs
16.         erreur = False
17.         # le type de session doit être connu avant certaines actions
18.         type_response = session.get('typeResponse')
19.         if type_response is None and action != "init-session":
20.             # on note l'erreur
21.             résultat = {"action": action, "état": 101,
22.                         "réponse": ["pas de session en cours. Commencer par action [init-session]"]}
23.             erreur = True
24.         # pour certaines actions on doit être authentifié
25.         user = session.get('user')
26.         if not erreur and user is None and action not in ["init-session", "authentifier-utilisateur"]:
27.             # on note l'erreur
28.             résultat = {"action": action, "état": 101,
29.                         "réponse": [f"action [{action}] demandée par utilisateur non authentifié"]}
30.             erreur = True
31.         # y-a-t-il des erreurs ?
32.         if erreur:
33.             # on renvoie un msg d'erreur
34.             return résultat, status.HTTP_400_BAD_REQUEST
35.         else:
36.             # on exécute le contrôleur associé à l'action
37.             controller = config["controllers"][action]
38.             résultat, status_code = controller.execute(request, session, config)
39.             return résultat, status_code
```

Le contrôleur **[MainController]** fait les premières vérifications de la validité de la requête.

- lignes 11-13 : le contrôleur commence par récupérer l'action demandée par le client. On rappelle que les URL de service sont de la forme **[/action/param1/param2/…]** et que cette URL est dans **[request.path]** ;
- lignes 17-23 : l'action **[init-session]** sert à initialiser le type de réponse (json, xml, html) désiré par le client. Cette information est mise en session associée à la clé **[typeRéponse]**. Si donc l'action n'est pas **[init-session]** alors la session doit contenir la clé **[typeRéponse]**, sinon la requête est erronée ;
- lignes 21-22 : la structure du résultat rendu par chaque contrôleur, ici un résultat d'erreur :
  - **[action]** : est le nom de l'action en cours. Cela va permettre d'avoir son nom lorsqu'on va loguer le résultat de la requête ;
  - **[état]** : est un code d'état à trois chiffres :
    - **[x00]** pour une réussite ;
    - **[x01]** pour un échec ;

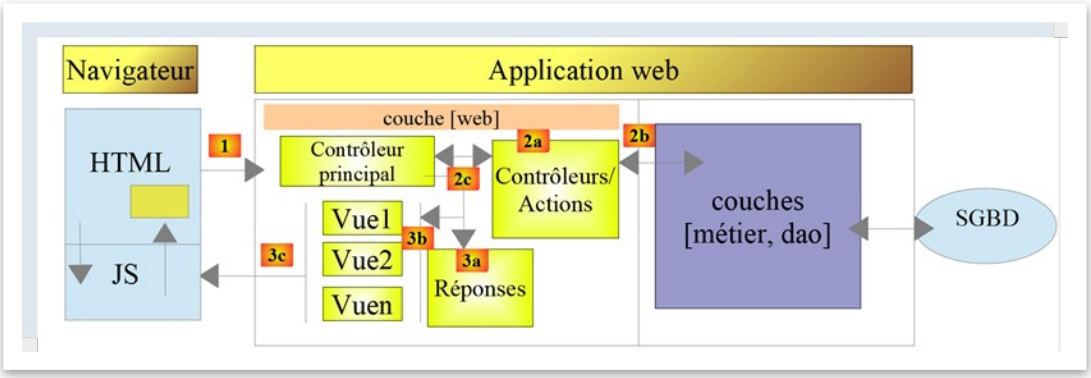
    - **[réponse]** : est la réponse à la requête. Sa nature est propre à chaque requête ;
- lignes 24-30 : l'action **[authentifier-utilisateur]** sert à authentifier l'utilisateur. Si elle réussit, une clé **[user=True]** est mise dans la session de l'utilisateur. Certaines URL de service ne sont accessibles que par un utilisateur authentifié. C'est ce qu'on vérifie ici ;
- ligne 26 : seules les actions **[init-session]** et **[authentifier-utilisateur]** peuvent être faites par un utilisateur non encore authentifié ;
- lignes 28-29 : le résultat à envoyer en cas d'erreur ;
- lignes 32-34 : si l'une des deux erreurs précédentes s'est produite, alors on envoie la réponse d'erreur au client avec le statut HTTP 400 BAD REQUEST ;
- lignes 35-39 : s'il n'y a pas eu d'erreur alors on passe la main au contrôleur chargé de traiter l'action en cours. Son instance est trouvée dans la configuration de l'application ;

La classe **[MainController]** continue le travail de la fonction **[front_controller]** : à elles deux, elles rassemblent tout ce qui peut être factorisé dans le traitement des requêtes, attendant le dernier moment pour passer la requête à un contrôleur spécifique. La répartition du code entre la fonction **[front_controller]** et la classe **[MainController]** est tout à fait subjective. Ici j'ai voulu conserver l'acquis de la version précédente : la fonction **[front_controller]** existait déjà sous le nom **[main]**. Dans la pratique, on pourrait :

- tout mettre dans la fonction **[front_controller]** et éliminer la classe **[MainController]** ;
- tout mettre dans la classe **[MainController]** et éliminer la fonction **[front_controller]**. C'est plutôt cette solution que je choisirais car elle a le mérite d'alléger le code du script principal **[main]** ;

30.7 Traitement spécifique à une action

Revenons à l'architecture MVC de l'application :



Nous en sommes toujours à l'étape 1 ci-dessus. S'il n'y a pas eu d'erreur, l'étape 2 va commencer. La requête a été transmise au contrôleur spécifique à l'action demandée par la requête. Supposons que cette action soit **[/init-session]** définie par la route :

```
1.  # init-session
2.  @app.route('/init-session/<string:type_response>', methods=['GET'])
3.  def init_session(type_response: str) -> tuple:
4.      # on exécute le contrôleur associé à l'action
5.      return front_controller()
```

Cette action est reliée à un contrôleur dans la configuration **[config]** :

```
1.          # actions autorisées et leurs contrôleurs
2.          "controllers": {
3.              # initialisation d'une session de calcul
4.              "init-session": InitSessionController(),
5.              …
6.          },
```

Le contrôleur **[InitSessionController]** (ligne 4) prend donc la main. Son code est le suivant :

```
1.  from flask_api import status
2.  from werkzeug.local import LocalProxy
3.
4.  from InterfaceController import InterfaceController
5.
6.  class InitSessionController(InterfaceController):
7.
```

```
8.      def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
9.          # on récupère les éléments du path
10.         dummy, action, type_response = request.path.split('/')
11.
12.         # au départ pas d'erreur
13.         erreur = False
14.         # vérification du type de réponse
15.         if type_response not in config['responses'].keys():
16.             erreur = True
17.             résultat = {"action": action, "état": 701,
18.                         "réponse": [f"paramètre [type={type_response}] invalide"]}
19.         # si pas d'erreur
20.         if not erreur:
21.             # on met le type de la session dans la session flask
22.             session['typeResponse'] = type_response
23.             résultat = {"action": action, "état": 700,
24.                         "réponse": [f"session démarrée avec le type de réponse {type_response}"]}
25.             return résultat, status.HTTP_200_OK
26.         else:
27.             return résultat, status.HTTP_400_BAD_REQUEST
```

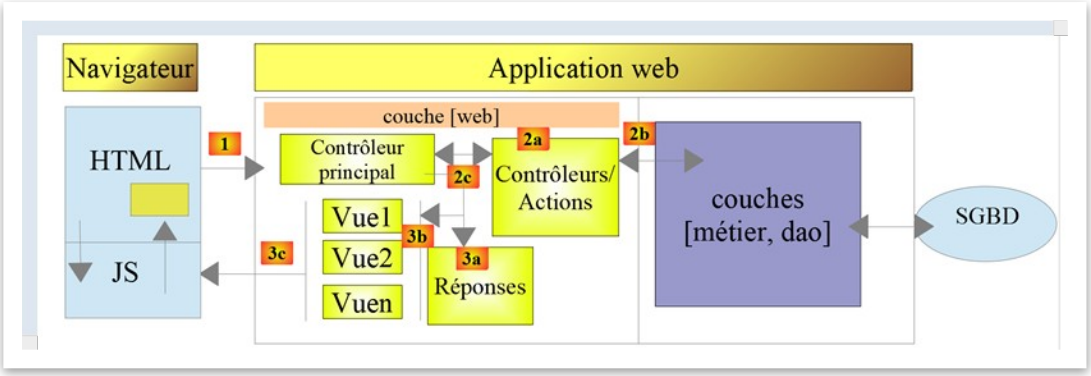
- ligne 6 : comme les autres contrôleurs, le contrôleur **[InitSessionController]** implémente l'interface **[InterfaceController]** ;
- ligne 10 : l'URL est de type **[/init-session/type_response]**. On récupère l'action **[init-session]** et le type de réponse souhaité ;
- ligne 15 : le type de réponse souhaité ne peut être que l'un de ceux présents dans la configuration des réponses :

```
        # les différents types de réponse (json, xml, html)
        "responses": {
            "json": JsonResponse(),
            "html": HtmlResponse(),
            "xml": XmlResponse()
        },
```

- si ce n'est pas le cas on prépare une réponse d'erreur 701 (ligne 17) ;
- lignes 20-25 : cas où le type de réponse souhaité est valide ;
  - ligne 22 : le type de réponse souhaité est mis en session. En effet, il va falloir s'en souvenir pour les requêtes qui vont suivre ;
  - lignes 23-24 : on prépare une réponse de réussite 700 ;
  - ligne 25 : la réponse de réussite est rendue au code appelant ;
- ligne 27 : s'il y a eu erreur, la réponse d'erreur est rendue au code appelant ;

30.8 Elaboration de la réponse HTTP du serveur

Revenons à l'architecture MVC de l'application :



Nous venons de voir les étapes 1 et 2. Nous avons rencontré trois codes d'état :

- 700 : /init-session a réussi ;
- 701 : /init-session a échoué ;
- 101 : requête invalide soit parce que la session n'a pas été initialisée soit parce que l'utilisateur n'est pas authentifié ;

Examinons comment la réponse du serveur va être envoyée au client lors de l'étape 3 ci-dessus. Cela se passe dans la fonction **[front_controller]** du script **[main]** :

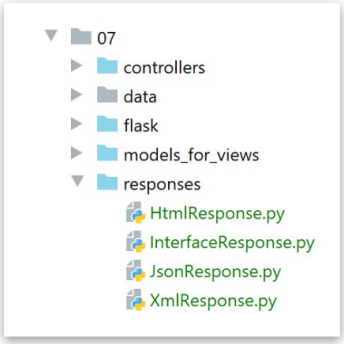
```
1.  # le front controller
2.  def front_controller() -> tuple:
3.      # on traite la requête
4.      logger = None
5.      try:
6.          # logger
7.          logger = Logger(config["logsFilename"])
8.          # on le mémorise dans une config associée au thread
9.          thread_config = {"logger": logger}
10.         thread_name = threading.current_thread().name
11.         config[thread_name] = {"config": thread_config}
12.         # on logue la requête
13.         logger.write(f"[ front_controller] requête : {request}\n")
14.         # on interrompt le thread si cela a été demandé
15.         sleep_time = config["sleep_time"]
16.         if sleep_time != 0:
17.             # la pause est aléatoire pour que certains threads soient interrompus et d'autres pas
18.             aléa = randint(0, 1)
19.             if aléa == 1:
20.                 # log avant pause
21.                 logger.write(f"[ front_controller] mis en pause du thread pendant {sleep_time} seconde(s)\n")
22.                 # pause
23.                 time.sleep(sleep_time)
24.         # on fait suivre la requête au contrôleur principal
25.         main_controller = config['controllers']["main-controller"]
26.         résultat, status_code = main_controller.execute(request, session, config)
27.         # on logue le résultat envoyé au client
28.         log = f"[front_controller] {résultat}\n"
29.         logger.write(log)
30.         # y-a-t-il eu une erreur fatale ?
31.         if status_code == status.HTTP_500_INTERNAL_SERVER_ERROR:
32.             # on envoie un mail à l'administrateur de l'application
33.             send_adminmail(config, log)
34.         # on détermine le type souhaité pour la réponse
35.         if session.get('typeResponse') is None:
36.             # le type de session n'a pas encore été établi - ce sera du jSON
37.             type_response = 'json'
38.         else:
39.             type_response = session['typeResponse']
40.         # on construit la réponse à envoyer
41.         response_builder = config["responses"][type_response]
42.         response, status_code = response_builder \
43.             .build_http_response(request, session, config, status_code, résultat)
44.         # on envoie la réponse
45.         return response, status_code
46.     except BaseException as erreur:
47.         # c'est une erreur inattendue - on logue l'erreur si c'est possible
48.         if logger:
49.             logger.write(f"[ front_controller] {erreur}")
50.         # on prépare la réponse au client
51.         résultat = {"réponse": {"erreurs": [f"{erreur}"]}}
52.         # on envoie lune réponse en jSON
53.         return json_response(résultat, status.HTTP_500_INTERNAL_SERVER_ERROR)
54.     finally:
55.         # on ferme le fichier de logs s'il a été ouvert
56.         if logger:
57.             logger.close()
```

- nous en sommes à la ligne 26 : le contrôleur principal a rendu sa réponse d'erreur ;
- lignes 27-29 : quelque soit la réponse du contrôleur principal (réussite ou échec) cette réponse est loguée dans le fichier de logs ;
- lignes 30-33 : comme dans les versions précédentes, si le statut HTTP est **[500 INTERNAL SERVER ERROR]**, on envoie un mail à l'administrateur de l'application avec le log de l'erreur ;
- lignes 34-39 : on va envoyer la réponse HTTP et le résultat rendu par le contrôleur va être mis dans le corps de cette réponse. Il nous faut savoir sous quelle forme (json, xml, html) le client veut cette réponse. On cherche le type de réponse souhaitée dans la session. S'il n'y est pas, alors on fixe arbitrairement ce type à du jSON ;
- lignes 40-43 : la réponse HTTP est construite ;

Dans le fichier de configuration, chaque type de réponse (json, xml, html) a été associé à une instance de classe :

```
1.          # les différents types de réponse (json, xml, html)
2.          "responses": {
3.              "json": JsonResponse(),
4.              "html": HtmlResponse(),
5.              "xml": XmlResponse()
6.          },
```

Les classes de réponses sont dans le dossier **[responses]** de l'arborescence du serveur :



Chaque classe de réponse implémente l'interface **[InterfaceResponse]** suivante :

```
1.  from abc import ABC, abstractmethod
2.
3.  from flask.wrappers import Response
4.  from werkzeug.local import LocalProxy
5.
6.  class InterfaceResponse(ABC):
7.
8.      @abstractmethod
9.      def build_http_response(self, request: LocalProxy, session: LocalProxy, config: dict, status_code: int,
10.                             résultat: dict) -> (Response, int):
11.         pass
```

- lignes 8-11 : l'interface **[InterfaceResponse]** définit une unique méthode **[build_http_response]** ayant les paramètres suivants :
  - **[request, session, config]** : ce sont les paramètres reçus par le contrôleur de l'action ;
  - **[résultat, status_code]** : ce sont les résultats produits par le contrôleur de l'action ;

Nous allons présenter la réponse jSON. Elle est produite par la classe **[JsonResponse]** suivante :

```
1.  import json
2.
3.  from flask import make_response
4.  from flask.wrappers import Response
5.  from werkzeug.local import LocalProxy
6.
7.  from InterfaceResponse import InterfaceResponse
8.
9.  class JsonResponse(InterfaceResponse):
10.
11.     def build_http_response(self, request: LocalProxy, session: LocalProxy, config: dict, status_code: int,
12.                             résultat: dict) -> (Response, int):
13.         # résultats : le dictionnaire des résultats
14.         # status_code : le code de statut de la réponse HTTP
15.
16.         # on rend la réponse HTTP
17.         response = make_response(json.dumps(résultat, ensure_ascii=False))
18.         response.headers['Content-Type'] = 'application/json; charset=utf-8'
19.         return response, status_code
```

Nous connaissons ce code que nous avons rencontré de nombreuses fois. C'est le code de la fonction **[json_response]** du module **[myutils]**.

30.9 Premiers tests

Dans le code étudié, nous avons rencontré trois codes d'état :

- 700 : /init-session a réussi ;
- 701 : /init-session a échoué ;
- 101 : requête invalide soit parce que la session n'a pas été initialisée soit parce que l'utilisateur n'est pas authentifié ;

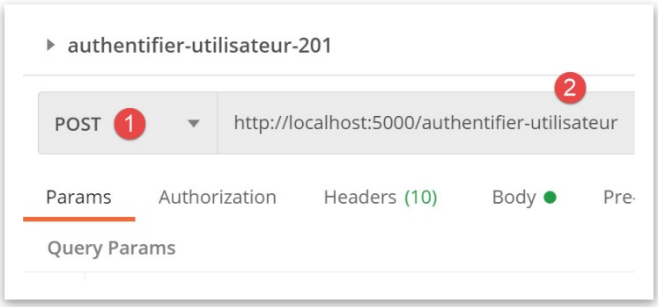
Nous allons essayer de les obtenir avec une session jSON.

- nous lançons le seveur web, le SGBD, le serveur de mails ;

- nous lançons un client Postman ;

Test 1

Nous montrons tout d'abord une requête invalide parce que la session n'a pas été initialisé :



- **[1-2]** : la requête **[POST http://localhost:5000/authentifier-utilisateur]** est une route valide :

```
# authentifier-utilisateur
@app.route('/authentifier-utilisateur', methods=['POST'])
def authentifier_utilisateur() -> tuple:
    # on exécute le contrôleur associé à l'action
    return front_controller()
```

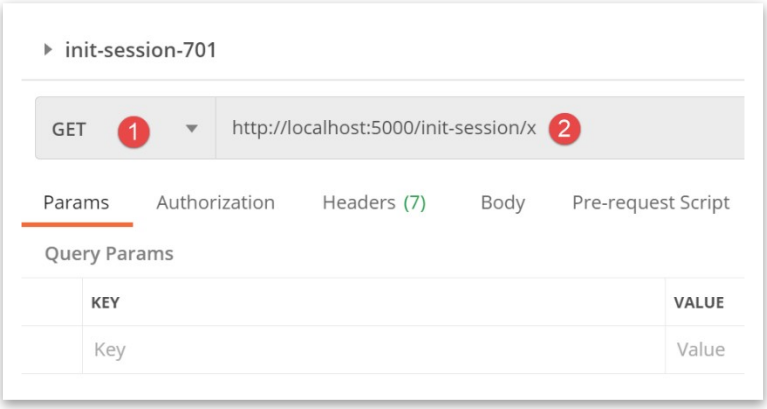
  mais elle n'est acceptée que si la session a été initialisée auparavant avec l'action **[/init-session]**.

Exécutons la requête et voyons le résultat envoyé par le serveur :



- **[1-2]** : on a obtenu une réponse jSON. Lorsque le type de réponse n'a pas encore été fixé par le client, le serveur utilise le jSON pour répondre ;
- **[3-5]** : le dictionnaire jSON de la réponse ;
  - **[action]** : l'action qui a été exécutée ;
  - **[état]** : le code d'état de la réponse. Un code **[x01]** indique une erreur ;
  - **[réponse]** : est adaptée à chaque action. Ici elle contient un message d'erreur ;

Maintenant initialisons une session avec un type de réponse incorrect :



- **[1-2]** est une route correcte :

```
# init-session
@app.route('/init-session/<string:type_response>', methods=['GET'])
def init_session(type_response: str) -> tuple:
    # on exécute le contrôleur associé à l'action
    return front_controller()
```

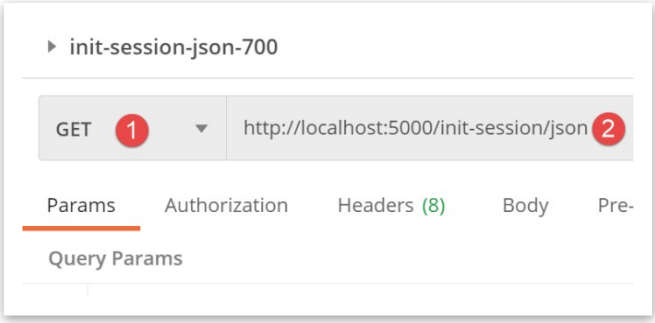
Elle va donc entrer dans le tunnel de traitement des requêtes du serveur MVC. Néanmoins elle devrait être refusée au cours de ce traitement parce que le type de session demandé est incorrect.

La réponse est la suivante :



- en **[4]**, un code d'erreur **[x01]** ;
- en **[5]**, l'explication de l'erreur ;

Maintenant, initialisons une session jSON :



La réponse est la suivante :

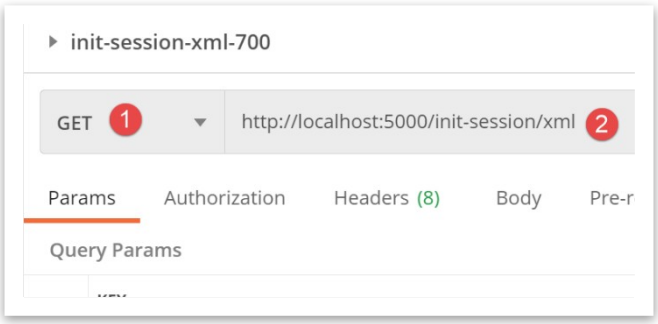


Maintenant, initialisons une session XML. La réponse jSON va être remplacée par une réponse XML générée par la classe **[XmlResponse]** suivante :

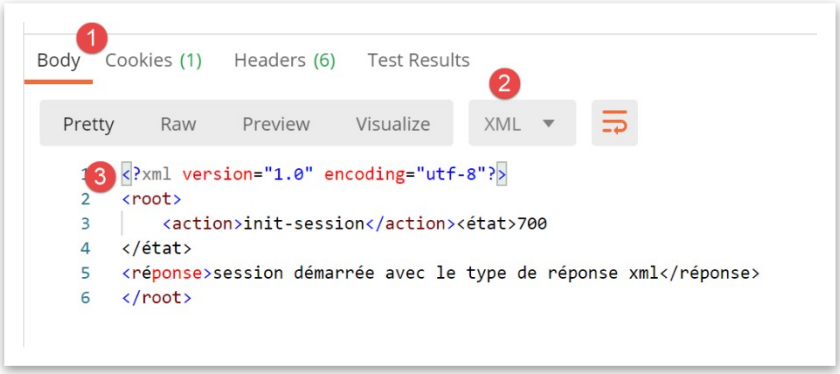
```
1.  import xmltodict
2.  from flask import make_response
3.  from flask.wrappers import Response
4.  from werkzeug.local import LocalProxy
5.
6.  from InterfaceResponse import InterfaceResponse
7.  from Logger import Logger
8.
9.  class XmlResponse(InterfaceResponse):
10.
11.     def build_http_response(self, request: LocalProxy, session: LocalProxy, config: dict, status_code: int,
12.                             résultat: dict) -> (Response, int):
13.         # résultats : le dictionnaire des résultats
14.         # status_code : le code de statut de la réponse HTTP
15.
16.         # résultat : le dictionnaire à transformer en chaîne XML
17.         xml_string = xmltodict.unparse({"root": résultat})
18.         # on rend la réponse HTTP
19.         response = make_response(xml_string)
20.         response.headers['Content-Type'] = 'application/xml; charset=utf-8'
21.         return response, status_code
```

C'est du code que nous connaissons, celui de la fonction **[xml_response]** du module partagé **[myutils]**.

Nous initialisons une session XML :



Le résultat du serveur est alors le suivant :



Nous obtenons la même réponse qu'en jSON mais cette fois-ci la réponse est habilllée en XML.

30.10 L'action `[authentifier-utilisateur]`

L'action **`[authentifier-utilisateur]`** permet d'authentifier un utilisateur désirant utiliser l'application de calcul de l'impôt. Sa route est définie de la façon suivante dans le script **`[main]`** :

```
# authentifier-utilisateur
@app.route('/authentifier-utilisateur', methods=['POST'])
def authentifier_utilisateur() -> tuple:
    # on exécute le contrôleur associé à l'action
    return front_controller()
```

Le serveur attend deux paramètres postés :

- **`[user]`** : l'identifiant de l'utilisateur ;
- **`[password]`** : son mot de passe ;

La liste des utilisateurs autorisés est définie dans la configuration **`[config]`** :

```
        # utilisateurs autorisés à utiliser l'application
        "users": [
            {
                "login": "admin",
                "password": "admin"
            }
        ],
```

Ici, nous avons une liste à un élément.

L'action **`[authentifier-utilisateur]`** est traitée par le contrôleur **`[AuthentifierUtilisateurController]`** suivant :

```
from flask_api import status
from werkzeug.local import LocalProxy

from InterfaceController import InterfaceController
from Logger import Logger

class AuthentifierUtilisateurController(InterfaceController):

    def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
        # on récupère les éléments du path
        dummy, action = request.path.split('/')

        # les paramètres du POST
        post_params = request.form
        # code de statut de la réponse HTTP
        status_code = None
        # au départ pas d'erreurs
        erreur = False
        erreurs = []
        # il faut un POST avec deux paramètres
        if len(post_params) != 2:
            erreur = True
            status_code = status.HTTP_400_BAD_REQUEST
            erreurs.append("méthode POST requise, paramètre [action] dans l'URL, paramètres postés [user, password]")
        if not erreur:
            # on récupère les paramètres du POST
```

```
27.             # paramètre [user]
28.             user = post_params.get("user")
29.             if user is None:
30.                 erreur = True
31.                 erreurs.append("paramètre [user] manquant")
32.             # paramètre [password]
33.             password = post_params.get("password")
34.             if password is None:
35.                 erreur = True
36.                 erreurs.append("paramètre [password] manquant")
37.             # erreur ?
38.             if erreur:
39.                 status_code = status.HTTP_400_BAD_REQUEST
40.         # erreur ?
41.         if not erreur:
42.             # on vérifie la validité du couple (user, password)
43.             users = config['users']
44.             i = 0
45.             nbusers = len(users)
46.             trouvé = False
47.             while not trouvé and i < nbusers:
48.                 trouvé = user == users[i]["login"] and password == users[i]["password"]
49.                 i += 1
50.             # trouvé ?
51.             if not trouvé:
52.                 # on note l'erreur
53.                 erreur = True
54.                 status_code = status.HTTP_401_UNAUTHORIZED
55.                 erreurs.append(f"Echec de l'authentification")
56.             else:
57.                 # on note dans la session qu'on a trouvé l'utilisateur
58.                 session["user"] = True
59.         # c'est fini
60.         if not erreur:
61.             # retour sans erreur
62.             résultat = {"action": action, "état": 200, "réponse": f"Authentification réussie"}
63.             return résultat, status.HTTP_200_OK
64.         else:
65.             # retour avec erreur
66.             return {"action": action, "état": 201, "réponse": erreurs}, status_code
67.
```

- ligne 14 : on récupère les paramètres du POST ;
- ligne 19 : la liste des erreurs trouvées dans la requête ;
- lignes 20-24 : on vérifie qu'il y a bien deux paramètres postés ;
- lignes 27-31 : on vérifie la présence d'un paramètre **[users]** ;
- lignes 32-36 : on vérifie la présence d'un paramètre **[password]** ;
- lignes 38-39 : si les paramètres postés sont erronés, on prépare une réponse HTTP 400 BAD REQUEST ;
- lignes 40-58 : on vérifie que les identifiants **[user, password]** sont ceux d'un utilisateur autorisé à utiliser l'application ;
- lignes 51-55 : si l'utilisateur (user, password) n'est pas autorisé à utiliser l'application, on prépare une réponse HTTP 401 UNAUTHORIZED ;
- lignes 56-58 : s'il est autorisé, alors on note avec la clé **[user]** dans la session qu'il s'est authentifié ;

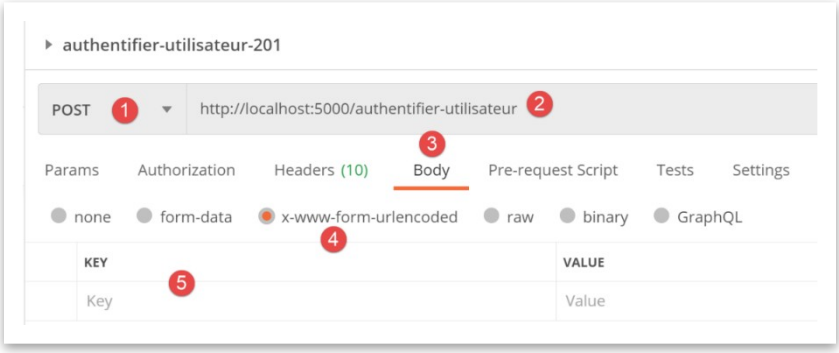
On notera que si l'utilisateur était authentifié avec les identifiants **[identifiants1]** et qu'il échoue à s'authentifier avec les identifiants **[identifiants2]**, il reste néanmoins authentifié avec les identifiants **[identifiants1]**.

Faisons des tests Postman :

- on lance le serveur web, le SGBD et le serveur de mails ;
- avec le client Postman :
  - on démarre une session jSON ;
  - puis on s'authentifie ;

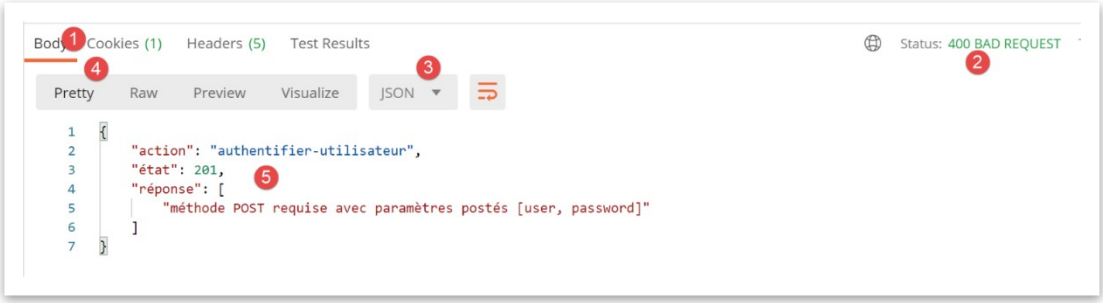
Voilà différents cas.

Cas 1 : POST sans paramètres postés



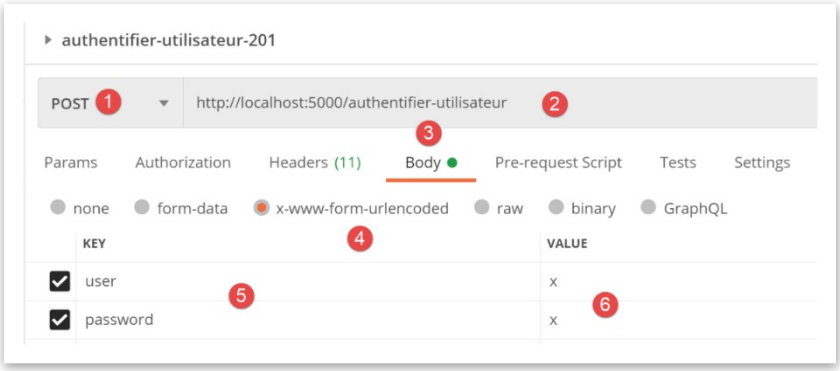
- en **[3-5]**, le POST n'a pas de corps ;

Le résultat de la requête est le suivant :



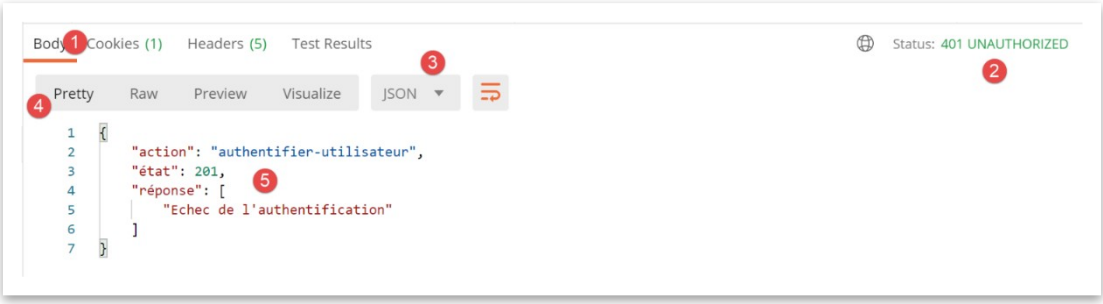
- en **[2]**, on a eu une réponse HTTP 400 BAD REQUEST ;
- en **[5]**, on a eu un code d'erreur **[201]** ;

Cas 2 : POST avec identifiants erronés



- en **[6]**, les identifiants sont erronés ;

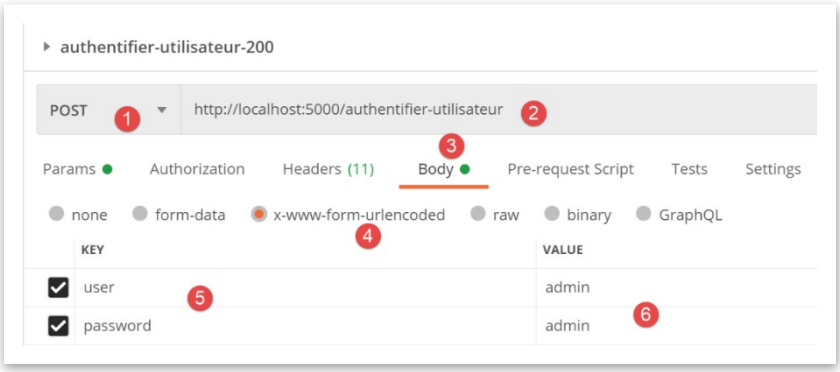
Le serveur envoie la réponse suivante :



- en **[2]**, la réponse HTTP 401 UNAUTHORIZED ;

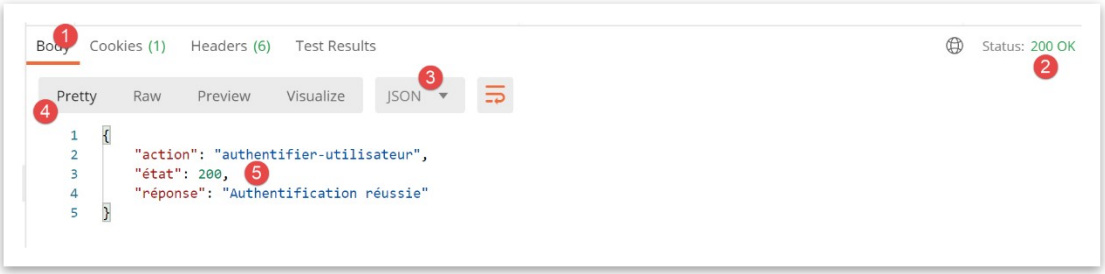
- en **[5]**, la réponse d'erreur ;

Cas 2 : POST avec identifiants corrects



- en **[6]**, les identifiants sont corrects ;

La réponse du serveur est la suivante :



- en **[2]**, un réponse HTTP 200 OK ;
- en **[5]**, la réponse de réussite ;

30.11 L'action `[calculer_impot]`

L'action **[calculer_impot]** permet de calculer l'impôt d'un contribuable. Sa route est définie de la façon suivante dans le script **[main]** :

```
# calculer-impot
@app.route('/calculer-impot', methods=['POST'])
def calculer_impot() -> tuple:
    # on exécute le contrôleur associé à l'action
    return front_controller()
```

Le serveur attend trois paramètres postés :

- **[marié]** : oui / non ;
- **[enfants]** : nombre d'enfants du contribuable ;
- **[salaire]** : salaire annuel du contribuable ;

Le contrôleur **[CalculerImpotController]** traite l'action **[calculer_impot]** :

```
import re

from flask_api import status
from werkzeug.local import LocalProxy

from InterfaceController import InterfaceController
from TaxPayer import TaxPayer

class CalculerImpotController(InterfaceController):

    def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
        # on récupère les éléments du path
        dummy, action = request.path.split('/')

```

```
        # pas d'erreur au départ
        erreur = False
        erreurs = []
        # les paramètres du POST
        post_params = request.form
        # il faut un POST avec trois paramètres
        if len(post_params) != 3:
            erreur = True
            erreurs.append(
                "méthode POST requise avec les paramètres postés [marié, enfants, salaire]")
        # on analyse les paramètres postés
        if not erreur:
            # paramètre marié
            marié = post_params.get("marié")
            if marié is None:
                erreurs.append("paramètre [marié] manquant")
            else:
                # le paramètre est-il valide ?
                marié = marié.lower()
                if marié != "oui" and marié != "non":
                    erreur = True
                    erreurs.append(f"valeur [{marié}] invalide pour le paramètre [marié (oui/non)]")
            # paramètre [enfants]
            enfants = post_params.get("enfants")
            if enfants is None:
                erreur = True
                erreurs.append("paramètre [enfants] manquant")
            else:
                # le paramètre est-il valide ?
                enfants = enfants.strip()
                match = re.match(r"\d+", enfants)
                if not match:
                    erreur = True
                    erreurs.append(f"valeur [{enfants}] invalide pour le paramètre [enfants (entier>=0)]")
            # paramètre salaire
            salaire = post_params.get("salaire")
            if salaire is None:
                erreur = True
                erreurs.append("paramètre [salaire] manquant")
            else:
                # le paramètre est-il valide ?
                salaire = salaire.strip()
                match = re.match(r"\d+", salaire)
                if not match:
                    erreur = True
                    erreurs.append(f"valeur [{salaire}] invalide pour le paramètre [salaire (entier>=0)]")
        # erreur ?
        if erreur:
            status_code = status.HTTP_400_BAD_REQUEST
            résultat = {"action": action, "état": 301, "réponse": erreurs}
            # on rend le résultat
            return résultat, status_code

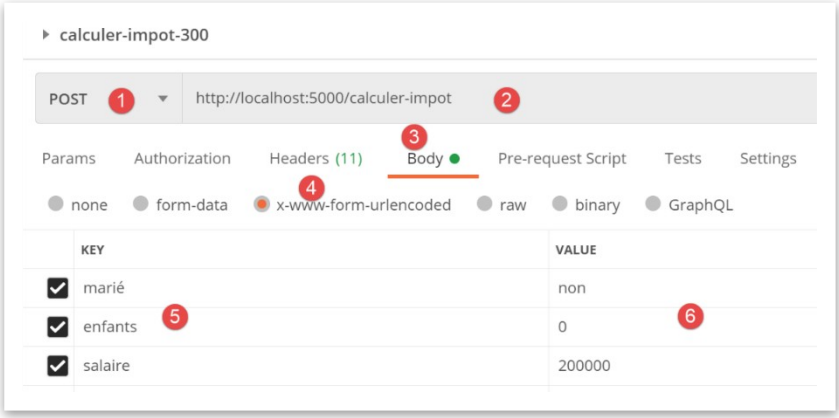
        # calcul de l'impôt
        # on récupère la couche [métier] et le dictionnaire [adminData]
        métier = config["layers"]["métier"]
        admin_data = config["admindata"]
        # calcul de l'impôt
        taxpayer = TaxPayer().fromdict({'marié': marié, 'enfants': enfants, 'salaire': salaire})
        métier.calculate_tax(taxpayer, admin_data)
        # n° de la simulation
        id_simulation = session.get('id_simulation', 0)
        id_simulation += 1
        session['id_simulation'] = id_simulation
        # on met le résultat en session ous la forme du dictionnaire d'un TaxPayer
        simulation = taxpayer.fromdict({'id': id_simulation}).asdict()
        # on ajoute le résultat à la liste des simulations déjà faites et on met celle-ci en session
        simulations = session.get("simulations", [])
        simulations.append(simulation)
        session["simulations"] = simulations
        # résultat
        résultat = {"action": action, "état": 300, "réponse": simulation}
        status_code = status.HTTP_200_OK

        # on rend le résultat
        return résultat, status_code
```

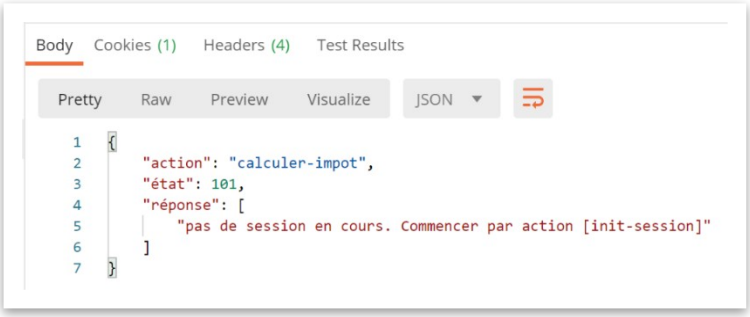
- ligne 13 : on récupère le nom de l'action en cours ;
- ligne 17: on va cumuler les erreurs dans une liste ;
- ligne 19 : on récupère les paramètres postés. Ceux-ci sont postés sous la forme **[x-www-form-urlencoded]** et c'est pourquoi on les récupère dans **[request.form]**. S'ils avaient été postés en jSON on les aurait récupérés dans **[request.data]** ;
- lignes 21-24 : on vérifie qu'il y a bien trois paramètres postés ;
- lignes 27-36 : vérification de la présence et de la validité du paramètre posté **[marié]** ;
- lignes 37-48 : vérification de la présence et de la validité du paramètre posté **[enfants]** ;
- lignes 49-60 : vérification de la présence et de la validité du paramètre posté **[salaire]** ;
- lignes 62-66 : s'il y a eu erreur, on envoie une réponse d'erreur 400 BAD REQUEST avec un code d'état **[301]** ;
- lignes 69-71 : s'il n'y a pas eu erreur, on se prépare à calculer l'impôt. Pour cela,
  - ligne 70 : on récupère une référence sur la couche **[métier]** ;
  - ligne 71 : on récupère les données de l'administration fiscale dans la configuration du serveur ;
- lignes 72-74 : l'impôt du contribuable est calculé ;
- lignes 75-77 : on va compter le nombre de calculs d'impôt faits par l'utilisateur ;
  - ligne 76 : on récupère en session, le n° du dernier calcul fait. On appelle ici **[simulation]** le résultat d'un calcul ;
  - ligne 77 : on incrémente le n° de la dernière simulation ;
  - ligne 78 : on remet ce n° en session ;
- lignes 79-84 : pour suivre les calculs faits par l'utilisateur, on va mettre dans sa session, la liste des simulations qu'il a faites ;
- ligne 80 : une simulation sera le dictionnaire d'un objet TaxPayer dont la propriété **[id]** aura pour valeur le n° de la simulation ;
- lignes 82-84 : la simulation courante est ajoutée à la liste des simulations présente en session ;
- lignes 86-87 : on prépare une réponse HTTP de réussite ;
- ligne 90 : on rend le résultat ;

Faisons quelques tests : le serveur web, le SGBD, le serveur de mails, un client Postman sont lancés.

Cas 1 : faire un calcul d'impôt alors que la session n'est pas initialisée

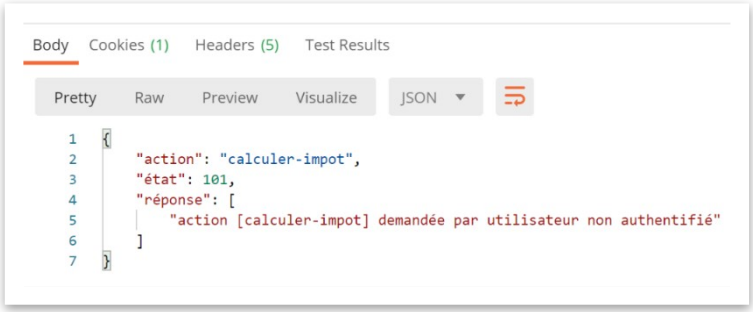


La réponse est la suivante :



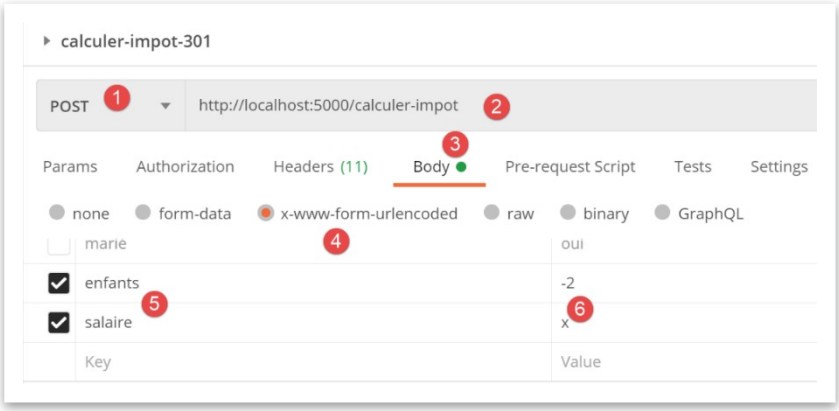
Cas 2 : faire un calcul d'impôt sans être authentifié

On lance tout d'abord une session jSON avec **[/init-session/json]**. Puis on fait la même requête que précédemment. La réponse est alors la suivante :



Cas 3 : faire un calcul d'impôt avec des paramètres manquants

On initialise une session jSON, on s'authentifie puis on fait la requête suivante :

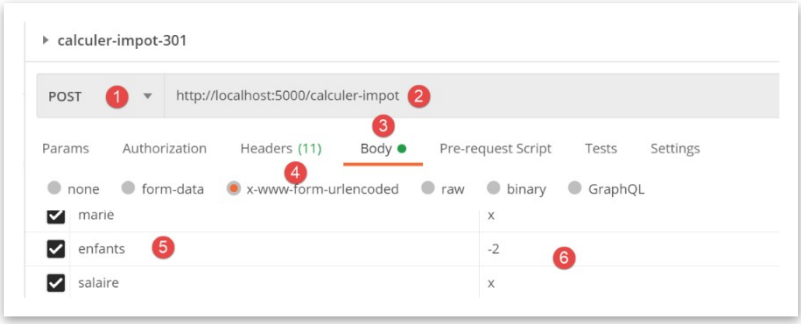


- en **[5]**, il manque le paramètre **[marié]** ;

La réponse est la suivante :



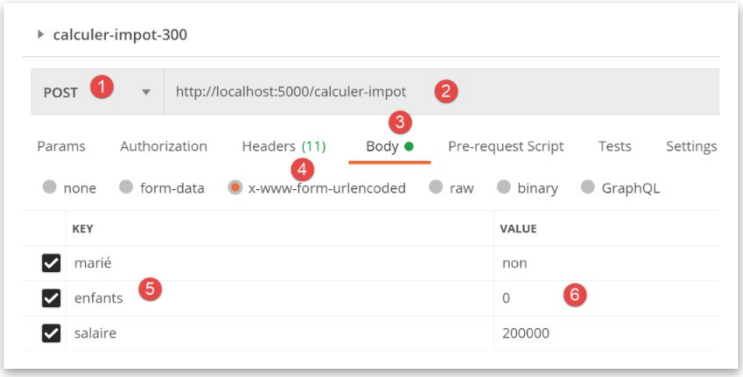
Cas 4 : faire un calcul d'impôt avec des paramètres erronés



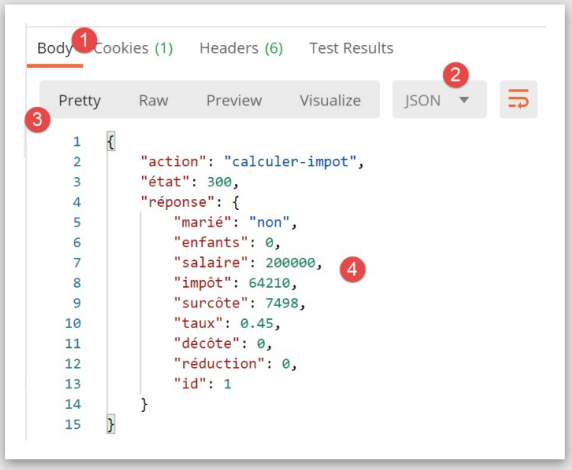
La réponse du serveur est la suivante :



Cas 4 : faire un calcul d'impôt avec des paramètres corrects



La réponse du serveur est la suivante :



30.12 L'action `[lister-simulations]`

L'action **`[lister-simulations]`** permet à un utilisateur de connaître la liste des simulations qu'il a faites depuis le début de la session. Sa route est définie de la façon suivante dans le script **`[main]`** :

```
# lister-simulations
@app.route('/lister-simulations', methods=['GET'])
def lister_simulations() -> tuple:
    # on exécute le contrôleur associé à l'action
    return front_controller()
```

Le serveur n'attend aucun paramètre. L'action **`[lister-simulations]`** est traitée par le contrôleur **`[ListerSimulationsController]`** suivant :

```
from flask_api import status
from werkzeug.local import LocalProxy

```

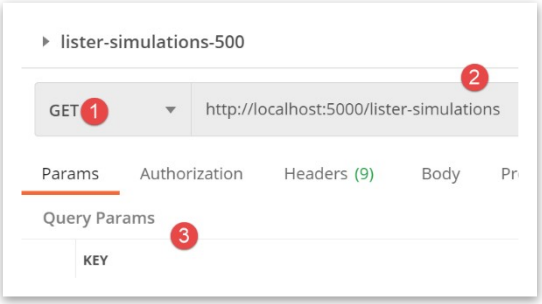
```
4.  from InterfaceController import InterfaceController
5.
6.  class ListerSimulationsController(InterfaceController):
7.
8.      def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
9.          # on récupère les éléments du path
10.         dummy, action = request.path.split('/')
11.
12.         # on récupère la liste des simulations dans la session
13.         simulations = session.get("simulations", [])
14.         # on rend le résultat
15.         return {"action": action, "état": 500,
16.                 "réponse": simulations}, status.HTTP_200_OK
```

- ligne 13 : la liste des simulations est prise dans la session ;
- lignes 15-16 : on retourne une réponse de réussite ;

Faisons le test Postman suivant :

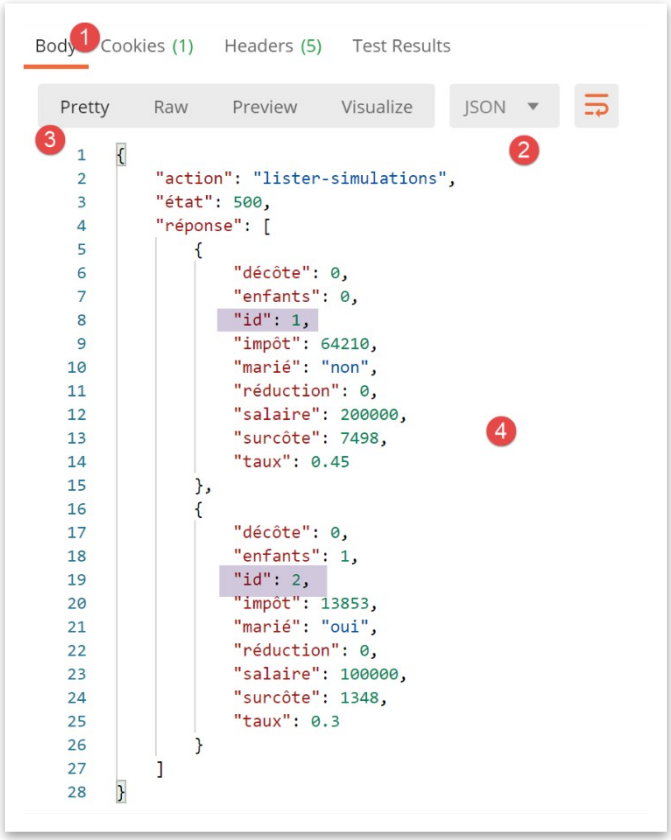
- on lance une session jSON ;
- on s'authentifie ;
- on fait deux calculs d'impôt ;
- on demande la liste des simulations ;

La requête est la suivante :



- en **[3]**, il n'y a aucun paramètre ;

La réponse du serveur est la suivante :



- en **[4]**, la liste des simulations de l'utilisateur ;

30.13 L'action `[supprimer-simulation]`

L'action **[supprimer-simulation]** permet à un utilisateur de supprimer une des simulations de sa liste de simulations. Sa route est définie de la façon suivante dans le script **[main]** :

```
# supprimer-simulation
@app.route('/supprimer-simulation/<int:numero>', methods=['GET'])
def supprimer_simulation(numero: int) -> tuple:
    # on exécute le contrôleur associé à l'action
    return front_controller()
```

Le serveur attend un unique paramètre, le n° de la simulation à supprimer. L'action **[supprimer-simulation]** est traitée par le contrôleur **[SupprimerSimulationController]** suivant :

```
from flask_api import status
from werkzeug.local import LocalProxy

from InterfaceController import InterfaceController

class SupprimerSimulationController(InterfaceController):

    def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
        # on récupère les éléments du path
        dummy, action, numéro = request.path.split('/')

        # le paramètre [numéro] est un entier positif ou nul d'après sa route
        numéro = int(numéro)
        # la simulation id=numéro doit exister dans la liste des simulations
        simulations = session.get("simulations", [])
        liste_simulations = list(filter(lambda simulation: simulation['id'] == numéro, simulations))
        if not liste_simulations:
            msg_erreur = f"la simulation n° [{numéro}] n'existe pas"
            # on rend l'erreur
            return {"action": action, "état": 601, "réponse": [msg_erreur]}, status.HTTP_400_BAD_REQUEST
        # suppression de la simulation id=numéro
        simulation = liste_simulations.pop(0)
        simulations.remove(simulation)
```

```
24.         # on remet les simulations dans la session
25.         session["simulations"] = simulations
26.         # on rend le résultat
27.         return {"action": action, "état": 600, "réponse": simulations}, status.HTTP_200_OK
```

- ligne 10 : on récupère les deux éléments du chemin de la requête. On les récupère en temps que chaîne de caractères ;
- ligne 13 : le paramètre **[numéro]** est transformé en entier. On sait que c'est possible à cause de la signature de sa route,

```
@app.route('/supprimer-simulation/<int:numero>', methods=['GET'])
```

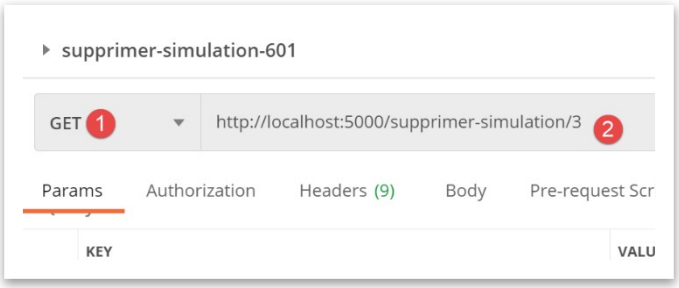
  On sait de plus que c'est un entier >=0. On ne peut pas en effet avoir une URL **[/supprimer-simulation/-4]**. Celle-ci est refusée par le serveur Flask ;
- ligne 15 : on récupère la liste des simulations dans la session ;
- ligne 16 : avec la fonction **[filter]**, on cherche la simulation ayant id==numéro. On obtient un objet **[filter]** qu'on convertit en type **[list]** ;
- lignes 17-20 : si le filtre n'a rien ramené, alors c'est que la simulation à supprimer n'existe pas. On rend une réponse d'erreur qui l'indique ;
- lignes 21-23 : on supprime la simulation ramenée par le filtre ;
- ligne 25 : on remet la nouvelle liste de simulations en session ;
- ligne 27 : on rend dans la réponse, la nouvelle liste de simulations ;

Nous faisons un test de réussite et un test d'échec. On fait des simulations puis on demande la liste des simulations :

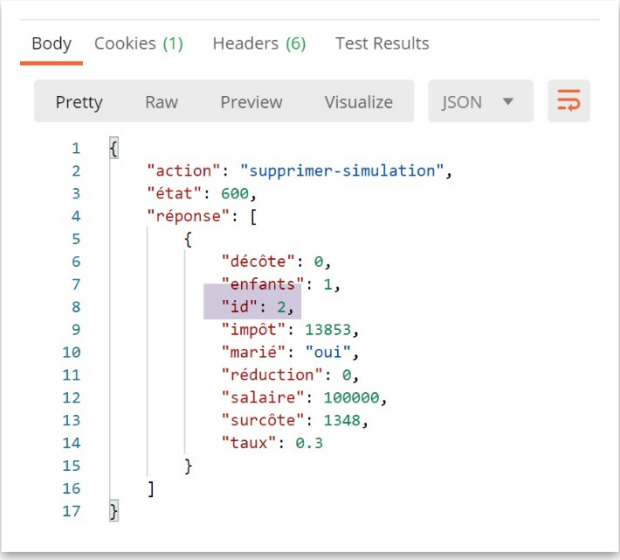


- les simulations ont ici les n°s 2 et 3 ;

On demande à supprimer la simulation ayant le n° 3.



La réponse est la suivante :



Maintenant, recommençons la même opération (suppression de la simulation d'id=3). La réponse est alors la suivante :



30.14 L'action `[fin-session]`

L'action **`[fin-session]`** permet à un utilisateur de terminer sa session de simulations. Sa route est définie de la façon suivante dans le script **`[main]`** :

```
# fin-session
@app.route('/fin-session', methods=['GET'])
def fin_session() -> tuple:
    # on exécute le contrôleur associé à l'action
    return front_controller()
```

Le serveur n'attend aucun paramètre. L'action est traitée par le contrôleur **`[FinSessionController]`** suivant :

```
from flask_api import status
from werkzeug.local import LocalProxy

from InterfaceController import InterfaceController

class FinSessionController(InterfaceController):

    def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
        # on récupère les éléments du path
        dummy, action = request.path.split('/')

        # on supprime toutes les clés la session courante
        session.clear()
        # on rend le résultat
        return {"action": action, "état": 400, "réponse": "session réinitialisée"}, status.HTTP_200_OK
```

- ligne 13 : on supprime toutes les clés de la session. Cela supprime :
  - **`[typeResponse]`** : le type des réponses HTTP (json, xml, html) ;

- **[id_simulation]** : n° de la dernière simulation faite ;
- **[simulations]** : la liste des simulations de l'utilisateur ;
- **[user]** : l'indicateur que l'utilisateur a été authentifié ;

- on rend la réponse ;

On peut se demander comment va être rendue la réponse HTTP de la ligne 15, maintenant que le type de réponse n'est plus dans la session. Pour le savoir, il faut revenir à la fonction |front_controller| du script principal **[main]** et la modifier de la façon suivante :

```
1.  …
2.          # on note le type de réponse souhaité si cette information est dans la session
3.          type_response1 = session.get('typeResponse', None)
4.          # on fait suivre la requête au contrôleur principal
5.          main_controller = config['controllers']["main-controller"]
6.          résultat, status_code = main_controller.execute(request, session, config)
7.          # on logue le résultat envoyé au client
8.          log = f"[front_controller] {résultat}\n"
9.          logger.write(log)
10.         # y-a-t-il eu une erreur fatale ?
11.         if status_code == status.HTTP_500_INTERNAL_SERVER_ERROR:
12.             # on envoie un mail à l'administrateur de l'application
13.             send_adminmail(config, log)
14.         # on détermine le type souhaité pour la réponse
15.         type_response2=session.get('typeResponse')
16.         if  type_response2 is None and type_response1 is None:
17.             # le type de session n'a pas encore été établi - ce sera du jSON
18.             type_response = 'json'
19.         elif type_response2 is not None:
20.             # le type de la réponse est connu et dans la session
21.             type_response = type_response2
22.         else:
23.             type_response=type_response1
24.         # on construit la réponse à envoyer
25.         response_builder = config["responses"][type_response]
26.         response, status_code = response_builder \
27.             .build_http_response(request, session, config, status_code, résultat)
28.         # on envoie la réponse
29.         return response, status_code
```

- ligne 3 : le type de la réponse actuellement en session est mémorisé ;
- ligne 6 : l'action est exécutée. S'il s'agit de :
  - **[fin-session]**, la clé **[typeResponse]** n'est alors plus dans la session ;
  - **[init-session]**, la clé **[typeResponse]** de la session a pu changer de valeur ;;
- lignes 14-20 : on doit émettre la réponse HTTP. Il nous faut savoir sous quelle forme :
  - lignes 16-18 : si le type de la réponse n'est définie ni par **[type_response1]** de la ligne 3, ni par **[type_response2]** de la ligne 15, alors le type de réponse n'était défini ni avant ni après l'action. On utilise alors du jSON (ligne 18) ;
  - lignes 19-21 : si **[type_response2]** existe, le type dans la session après l'action, alors c'est ce type qu'il faut utiliser ;
  - lignes 22-23 : sinon c'est **[type_response1]**, le type de réponse avant l'action (celle-ci est forcément **[fin-session]**) qu'il faut utiliser ;

30.15 L'action [get-admindata]

Nous abordons maintenant les deux URL réservées aux services jSON et XML :

Action	Rôle	Contexte d'exécution
/get-admindata	Rend les données fiscales permettant le calcul de l'impôt	Requête GET. N'est utilisée que si le type de la session est json ou xml. L'utilisateur doit être authentifié
/calculer-impots	Fait le calcul de l'impôt d'une liste de contribuables postés en jSON	Requête GET. N'est utilisée que si le type de la session est json ou xml. L'utilisateur doit être authentifié

L'URL **[/get-admindata]** est définie dans les routes du script principal **[main]** de la façon suivante :

```
1.  # get-admindata
2.  @app.route('/get-admindata', methods=['GET'])
3.  def get_admindata() -> tuple:
4.      # on exécute le contrôleur associé à l'action
5.      return front_controller()
```

La route **[/get-admindata]** est traitée par le contrôleur **[GetAdminDataController]** suivant :

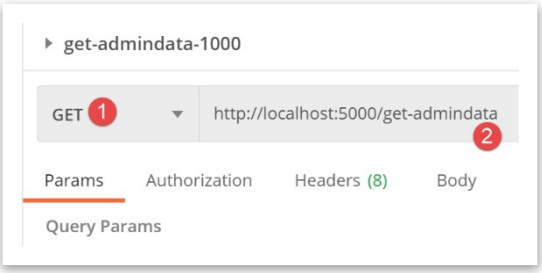
```
1.  # import des dépendances
2.
3.  from flask_api import status
```

```
4.  from werkzeug.local import LocalProxy
5.
6.  from InterfaceController import InterfaceController
7.
8.  class GetAdminDataController(InterfaceController):
9.
10.     def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
11.         # on récupère les éléments du path
12.         dummy, action = request.path.split('/')
13.         # seules les sessions json et xml sont acceptées
14.         type_response = session.get('typeResponse')
15.         if type_response != 'json' and type_response != 'xml':
16.             # on rend une réponse d'erreur
17.             return {
18.                         "action": action,
19.                         "état": 1001,
20.                         "réponse": ["cette action n'est possible que pour les sessions json ou xml"]
21.                     }, status.HTTP_400_BAD_REQUEST
22.         else:
23.             # on rend une réponse de réussite
24.             return {"action": action, "état": 1000, "réponse": config["adminData"].asdict()}, status.HTTP_200_OK
```

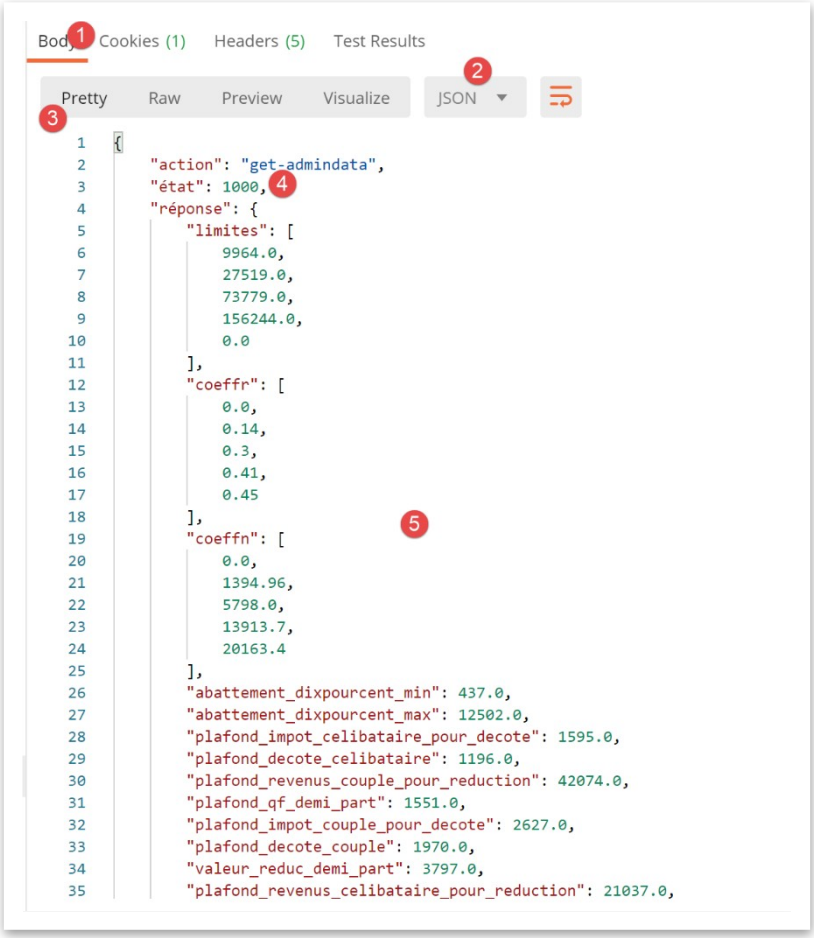
- lignes 13-21 : on vérifie qu'on est dans une session json ou xml ;
- ligne 24 : on rend le dictionnaire des données de l'administration fiscale qui dès le démarrage du serveur avaient été placées dans la configuration :

```
    # admindata sera une donnée de portée application en lecture seule
    config["admindata"] = config["layers"]["dao"].get_admindata()
```

Prenons un client Postman et demandons l'URL **[/get-admindata]**, après avoir démarré une session jSON et s'être authentifié :



La réponse du serveur est la suivante :



30.16 L'action `[calculer-impots]`

L'action **`[calculer-impots]`** calcule l'impôts d'une liste de contribuables trouvée dans le corps de la requête sous la forme d'une chaîne jSON. Nous connaissons déjà cette action : elle s'appelait **`[calculate_tax_in_bulk_mode]`** dans la version précédente.

Sa route est la suivante :

```
# calcul de l'impôt par lots
@app.route('/calculer-impots', methods=['POST'])
def calculer_impots():
    # on exécute le contrôleur associé à l'action
    return front_controller()
```

Cette action est traitée par le contrôleur **`[CalculerImpotsController]`** suivant :

```
import json

from flask_api import status
from werkzeug.local import LocalProxy

from ImpôtsError import ImpôtsError
from InterfaceController import InterfaceController
from TaxPayer import TaxPayer

class CalculerImpotsController(InterfaceController):

    def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
        # on récupère les éléments du path
        dummy, action = request.path.split('/')

        # seules les sessions json et xml sont acceptées
        type_response = session.get('typeResponse')
        if type_response != 'json' and type_response != 'xml':
            # on rend une réponse d'erreur
            return {
                        "action": action,
```

```
                        "état": 1501,
                        "réponse": ["cette action n'est possible que pour les sessions json ou xml"]
                    }, status.HTTP_400_BAD_REQUEST

        # on récupère le corps du post - on attend une liste de dictionnaires
        msg_erreur = None
        list_dict_taxpayers = None
        # le corps jSON du POST
        request_text = request.data
        try:
            # qu'on transforme en une liste de dictionnaires
            list_dict_taxpayers = json.loads(request_text)
        except BaseException as erreur:
            # on note l'erreur
            msg_erreur = f"le corps du POST n'est pas une chaîne jSON valide : {erreur}"
        # a-t-on une liste non vide ?
        if not msg_erreur and (not isinstance(list_dict_taxpayers, list) or len(list_dict_taxpayers) == 0):
            # on note l'erreur
            msg_erreur = "le corps du POST n'est pas une liste ou alors cette liste est vide"
        # a-t-on une liste de dictionnaires ?
        if not msg_erreur:
            erreur = False
            i = 0
            while not erreur and i < len(list_dict_taxpayers):
                erreur = not isinstance(list_dict_taxpayers[i], dict)
                i += 1
            # erreur ?
            if erreur:
                msg_erreur = "le corps du POST doit être une liste de dictionnaires"
        # erreur ?
        if msg_erreur:
            # on envoie une réponse d'erreur au client
            résultats = {"action": action, "état": 1501, "réponse": [msg_erreur]}
            return résultats, status.HTTP_400_BAD_REQUEST

        # on vérifie les TaxPayers un par un
        # au départ pas d'erreurs
        list_erreurs = []
        for dict_taxpayer in list_dict_taxpayers:
            # on crée un TaxPayer à partir de dict_taxpayer
            msg_erreur = None
            try:
                # l'opération suivante va éliminer les cas où les paramètres ne sont pas
                # des propriétés de la classe TaxPayer ainsi que les cas où leurs valeurs
                # sont incorrectes
                TaxPayer().fromdict(dict_taxpayer)
            except BaseException as erreur:
                msg_erreur = f"{erreur}"
            # certaines clés doivent être présentes dans le dictionnaire
            if not msg_erreur:
                # les clés [marié, enfants, salaire] doivent être présentes dans le dictionnaire
                keys = dict_taxpayer.keys()
                if 'marié' not in keys or 'enfants' not in keys or 'salaire' not in keys:
                    msg_erreur = "le dictionnaire doit inclure les clés [marié, enfants, salaire]"
            # des erreurs ?
            if msg_erreur:
                # on note l'erreur dans le TaxPayer lui-même
                dict_taxpayer['erreur'] = msg_erreur
                # on ajoute le TaxPayer à la liste des erreurs
                list_erreurs.append(dict_taxpayer)

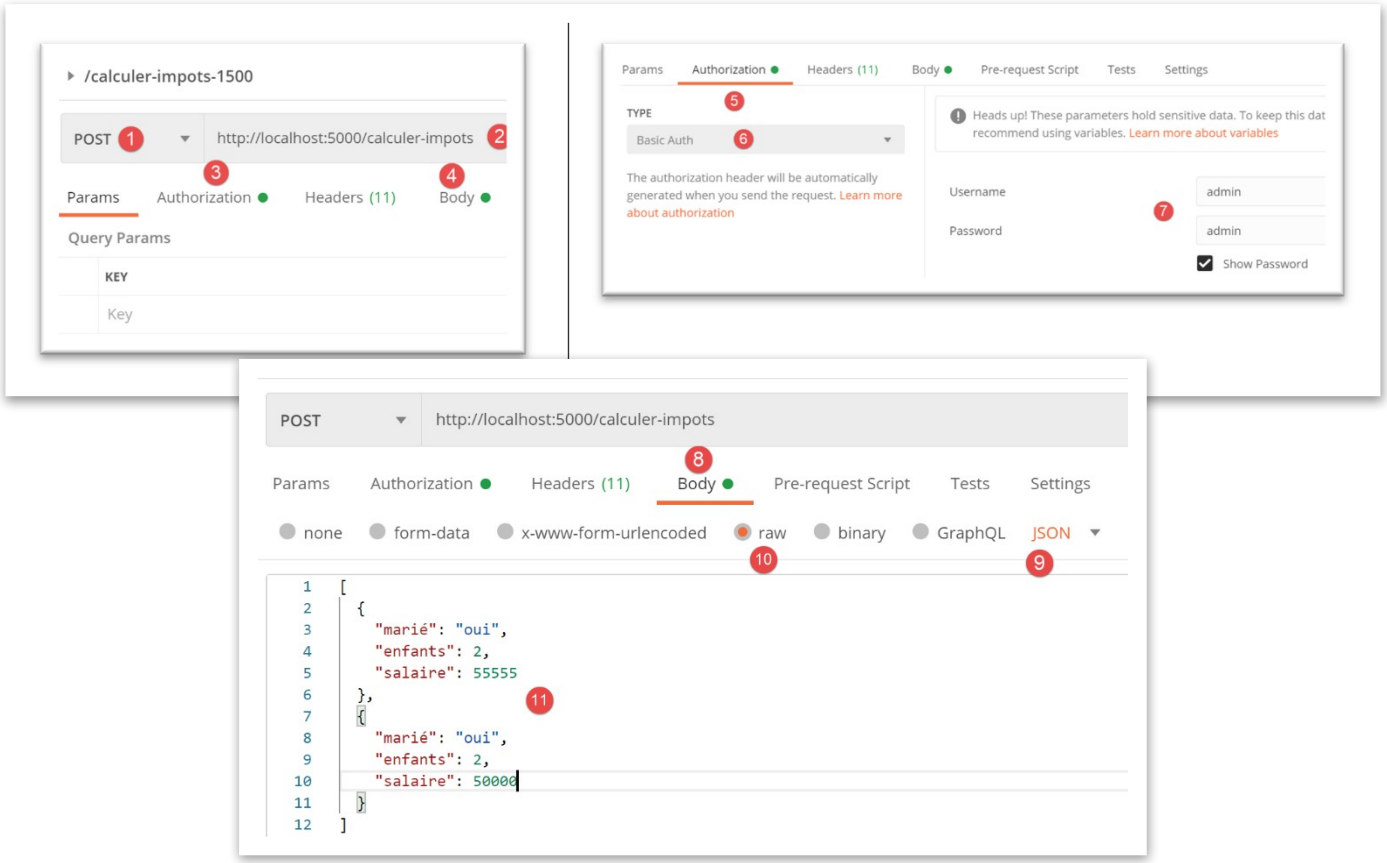
        # on a traité tous les taxpayers - y-a-t-il des erreurs ?
        if list_erreurs:
            # on envoie une réponse d'erreur au client
            résultats = {"action": action, "état": 1501, "réponse": list_erreurs}
            return résultats, status.HTTP_400_BAD_REQUEST

        # pas d'erreurs, on peut travailler
        # récupération des données de l'administration fiscale
        admindata = config["admindata"]
        métier = config["layers"]["métier"]
        try:
            # on traite les TaxPayer un à un
            list_taxpayers = []
            for dict_taxpayer in list_dict_taxpayers:
                # calcul de l'impôt
                taxpayer = TaxPayer().fromdict(
                    {'marié': dict_taxpayer['marié'], 'enfants': dict_taxpayer['enfants'],
                     'salaire': dict_taxpayer['salaire']})
                métier.calculate_tax(taxpayer, admindata)
                # on mémorise le résultat en tant que dictionnaire
                list_taxpayers.append(taxpayer.asdict())
            # on ajoute list_taxpayers aux simulations actuelles en donnant à chaque simulation un n°
            simulations = session.get("simulations", [])
            id_simulation = session.get("id_simulation", 0)
            for simulation in list_taxpayers:
```

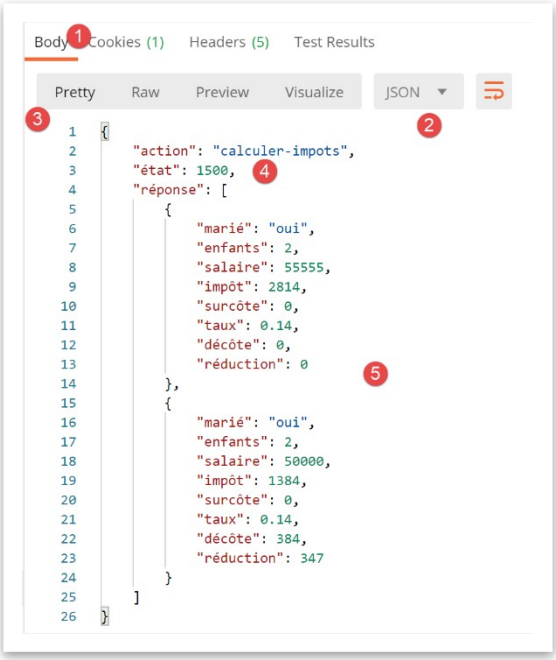
```
108.                # on donne un n° à chaque simulation
109.                id_simulation += 1
110.                simulation['id'] = id_simulation
111.                # on l'ajoute à la liste actuelle des simulations
112.                simulations.append(simulation)
113.            # on remet le tout en session
114.            session["simulations"] = simulations
115.            session["id_simulation"] = id_simulation
116.            # on envoie la réponse au client
117.            return {"action": action, "état": 1500, "réponse": list_taxpayers}, status.HTTP_200_OK
118.        except ImpôtsError as erreur:
119.            # on envoie une réponse d'erreur au client
120.            return {"action": action, "état": 1501, "réponse": [f"{erreur}"]}, status.HTTP_500_INTERNAL_SERVER_ERROR
```

- lignes 16-24 : on vérifie qu'on est bien dans une session json ou xml
- lignes 26-120 : ce code nous est globalement connu. C'est celui de la fonction |index_controller| de la version 10 de l'application qui a été aménagé pour répondre aux spécifications de l'interface **`[InterfaceController]`** implémentée ;
- lignes 104-115 : le code ajouté pour tenir compte du nouvel environnement de ce contrôleur. On vient de faire des calculs d'impôt. Il nous faut mémoriser les résultats dans la liste des simulations maintenues en session ;
- ligne 105 : on récupère la liste des simulations en session ;
- ligne 106 : on récupère le n° de la dernière simulation faite ;
- lignes 107-112 : on parcourt la liste des dictionnaires des résultats du calcul de l'impôt, à chacun d'eux on attribue un n° **`[id]`** de simulation et chaque dictionnaire est ajouté à la liste des simulations ;
- lignes 113-115 : la nouvelle liste des simulations ainsi que le n° de la dernière simulation faite sont remis en session ;

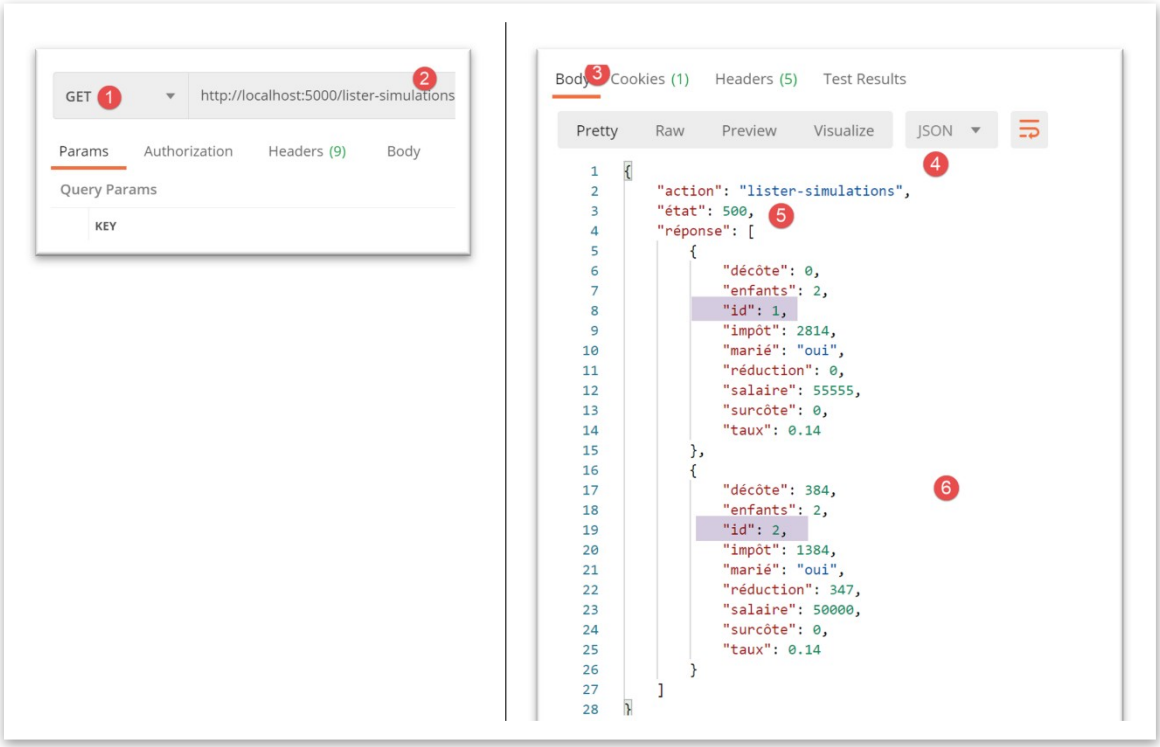
Nous faisons le test Postman suivant, après avoir initialisé une session jSON et s'être authentifié :



La réponse du serveur est la suivante :



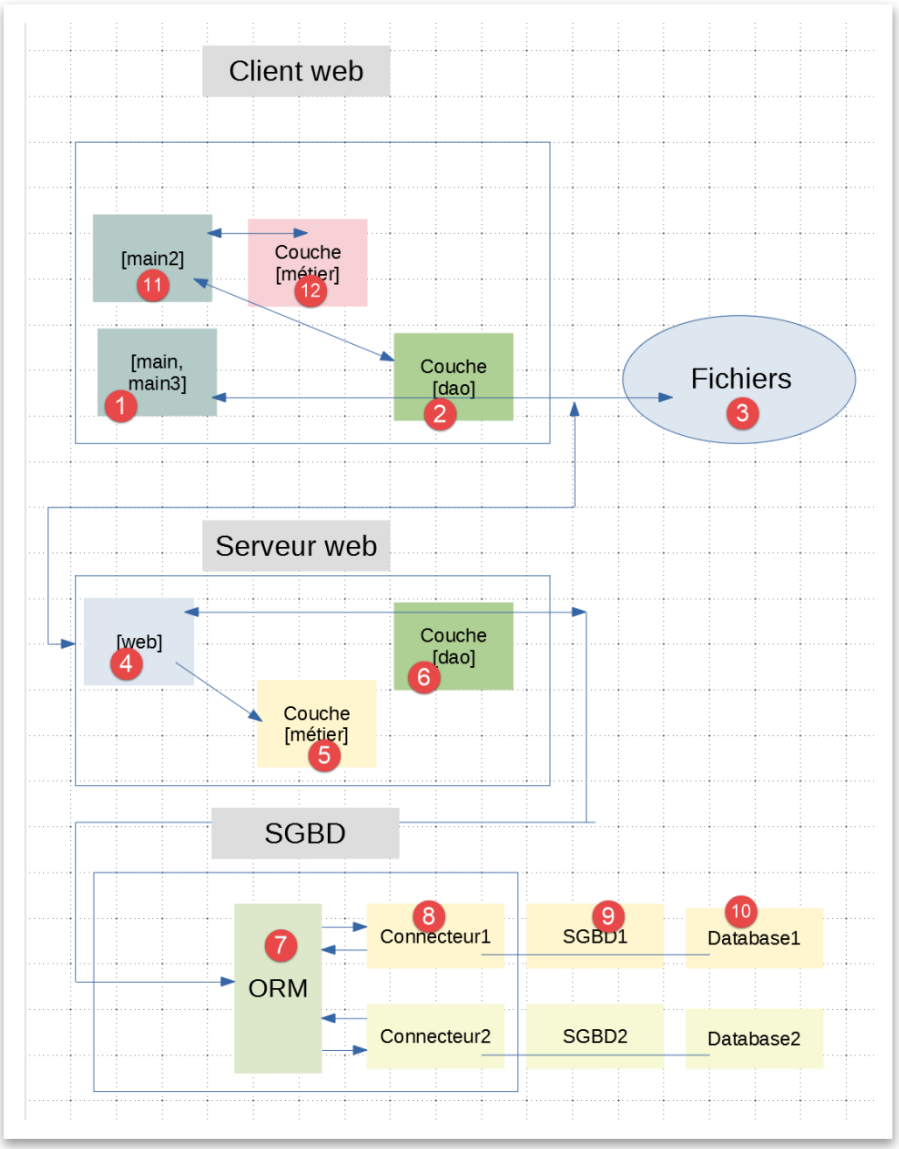
Si maintenant, on demande la liste des simulations :



On remarquera que dans la liste résultat de **`[/calcul-impots]`**, les contribuables n'ont pas d'attribut **`[id]`** alors que dans la liste des simulations, chaque simulation a un n° qui l'identifie.

31 Clients web pour les services jSON et XML de la version 12

Nous allons écrire trois applications console clientes des services jSON et XML du serveur web que nous venons d'écrire. Nous reprenons l'architecture client / serveur de la version 11 :



Nous écrirons trois scripts console :

- les scripts **[main]** et **[main3]** utiliseront la couche **[métier]** du serveur ;
- le script **[main2]** utilisera la couche **[métier]** du client ;

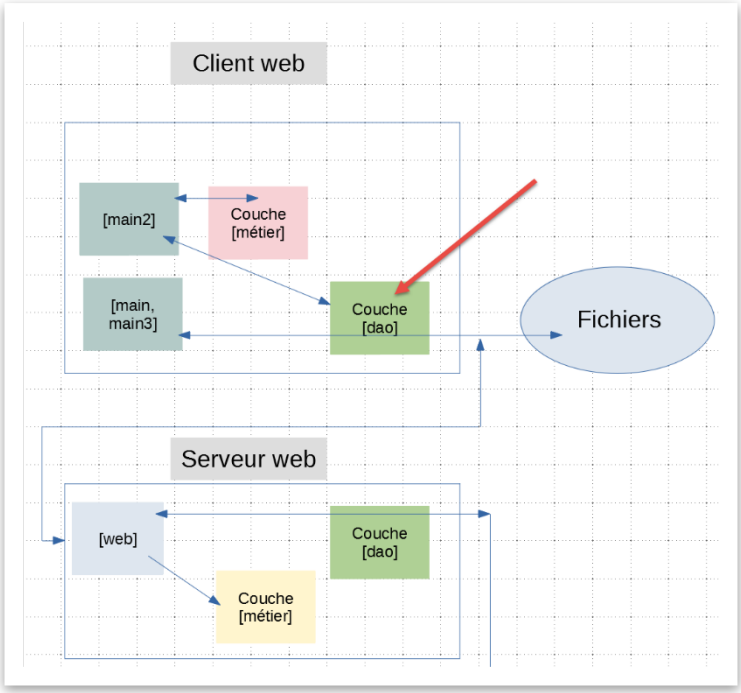
31.1 L'arborescence des scripts des clients

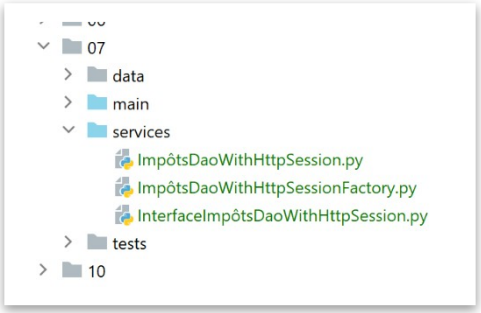
Le dossier **[http-clients/07]** est obtenu initialement par recopie du dossier **[http-clients/06]**. Il est ensuite modifié.



- en **[1]** : les données exploitées ou créées par le client ;
- en **[2]**, la configuration et les scripts console du client ;
- en **[3]**, la couche **[dao]** du client ;
- en **[4]**, le dossier des tests de la couche **[dao]** du client ;

31.2 La couche [dao] des clients





31.2.1 Interface

La couche **[dao]** implémentera l'interface **[InterfaceImpôtsDaoWithHttpSession]** suivante :

```
1.  from abc import abstractmethod
2.
3.  from AbstractImpôtsDao import AbstractImpôtsDao
4.  from AdminData import AdminData
5.  from TaxPayer import TaxPayer
6.
7.  class InterfaceImpôtsDaoWithHttpSession(AbstractImpôtsDao):
8.
9.      # calcul de l'impôt par unité
10.     @abstractmethod
11.     def calculate_tax(self, taxpayer: TaxPayer):
12.         pass
13.
14.     # calcul de l'impôt par lots
15.     @abstractmethod
16.     def calculate_tax_in_bulk_mode(self, taxpayers: list):
17.         pass
18.
19.     # initialisation d'une session
20.     @abstractmethod
21.     def init_session(self, type_session: str):
22.         pass
23.
24.     # fin de session
25.     @abstractmethod
26.     def end_session(self):
27.         pass
28.
29.     # authentification
30.     @abstractmethod
31.     def authenticate_user(self, user: str, password: str):
32.         pass
33.
34.     # liste des simulations
35.     @abstractmethod
36.     def get_simulations(self) -> list:
37.         pass
38.
39.     # supprimer une simulation
40.     @abstractmethod
41.     def delete_simulation(self, id: int) -> list:
42.         pass
43.
44.     # obtenir les données permettant le calcul de l'impôt
45.     @abstractmethod
46.     def get_admindata(self) -> AdminData:
47.         pass
```

Chaque méthode de l'interface correspond à une URL de service du serveur de calcul de l'impôt.

- ligne 7 : l'interface étend la classe **[AbstractDao]** qui gère les accès au système de fichiers ;

La correspondance entre les méthodes et les URL de service est faite dans le fichier de configuration **[config]** :

```
1.          # le serveur de calcul de l'impôt
2.          "server": {
3.              "urlServer": "http://127.0.0.1:5000",
4.              "user": {
5.                  "login": "admin",
6.                  "password": "admin"
7.              },
8.              "url_services": {
9.                  "calculate-tax": "/calculer-impot",
10.                 "get-admindata": "/get-admindata",
11.                 "calculate-tax-in-bulk-mode": "/calculer-impots",
12.                 "init-session": "/init-session",
13.                 "end-session": "/fin-session",
14.                 "authenticate-user": "/authentifier-utilisateur",
15.                 "get-simulations": "/lister-simulations",
16.                 "delete-simulation": "/supprimer-simulation"
17.             }
```

31.2.2 Implémentation

L'interface **[InterfaceImpôtsDaoWithHttpSession]** est implémentée par la classe **[ImpôtsDaoWithHttpSession]** suivante :

```
1.  # imports
2.  import json
3.
4.  import requests
5.  import xmltodict
6.  from flask_api import status
7.
8.  from AbstractImpôtsDao import AbstractImpôtsDao
9.  from AdminData import AdminData
10. from ImpôtsError import ImpôtsError
11. from InterfaceImpôtsDaoWithHttpSession import InterfaceImpôtsDaoWithHttpSession
12. from TaxPayer import TaxPayer
13.
14. class ImpôtsDaoWithHttpSession(InterfaceImpôtsDaoWithHttpSession):
15.
16.     # constructeur
17.     def __init__(self, config: dict):
18.         # initialisation parent
19.         AbstractImpôtsDao.__init__(self, config)
20.         # mémorisation éléments de la configuration
21.         # config générale
22.         self.__config = config
23.         # serveur
24.         self.__config_server = config["server"]
25.         # services
26.         self.__config_services = config["server"]['url_services']
27.         # mode debug
28.         self.__debug = config["debug"]
29.         # logger
30.         self.__logger = None
31.         # cookies
32.         self.__cookies = None
33.         # type de session (json, xml)
34.         self.__session_type = None
35.
36.         # étape request / response
37.      def get_response(self, method: str, url_service: str, data_value: dict = None, json_value=None):
38.         # [method] : méthode HTTP GET ou POST
39.         # [url_service] : URL de service
40.         # [data] : paramètres du POST en x-www-form-urlencoded
41.         # [json] : paramètres du POST en json
42.         # [cookies]: cookies à inclure dans la requête
43.
44.         # on doit avoir une session XML ou JSON, sinon on ne pourra pas gérer la réponse
45.         if self.__session_type not in ['json', 'xml']:
46.             raise ImpôtsError(73, "il n'y a pas de session valide en cours")
47.
48.         # connexion
49.         if method == "GET":
50.             # GET
51.             response = requests.get(url_service, cookies=self.__cookies)
52.         else:
53.             # POST
54.             response = requests.post(url_service, data=data_value, json=json_value, cookies=self.__cookies)
55.
56.         # mode debug ?
57.         if self.__debug:
58.             # logueur
59.             if not self.__logger:
60.                 self.__logger = self.__config['logger']
61.             # on logue
62.             self.__logger.write(f"{response.text}\n")
63.
64.         # résultat
65.         if self.__session_type == "json":
66.             résultat = json.loads(response.text)
67.         else:  # xml
68.             résultat = xmltodict.parse(response.text[39:])['root']
69.
70.         # on récupère les cookies de la réponse s'il y en a
71.         if response.cookies:
72.             self.__cookies = response.cookies
73.
74.         # code de statut
75.         status_code = response.status_code
76.
77.         # si code de statut différent de 200 OK
78.         if status_code != status.HTTP_200_OK:
79.             raise ImpôtsError(35, résultat['réponse'])
80.
81.         # on rend le résultat
```

```
82.         return résultat['réponse']
83.
84.     def init_session(self, session_type: str):
85.         # on note le type de la session
86.         self.__session_type = session_type
87.
88.         # url de service
89.         config_server = self.__config_server
90.         url_service = f"{config_server['urlServer']}{self.__config_services['init-session']}/{session_type}"
91.
92.         # exécution requête
93.         self.get_response("GET", url_service)
94. …
```

- lignes 16-34 : le constructeur de la classe ;
- ligne 19 : la classe parent est initialisée ;
- lignes 21-28 : on mémorise certaines données de la configuration ;
- lignes 29-34 : on crée trois propriétés utilisées dans les méthodes de la classe ;
- lignes 36-82 : la méthode **[get_response]** factorise ce qui est commun à toutes les méthodes de la couche **[dao]** : l'envoi d'une requête HTTP et la récupération de la réponse HTTP du serveur ;
- lignes 38-42 : définition des 5 paramètres de la méthode **[get_response]** ;
- ligne 42 : on notera que parce que le serveur maintient une session, le client a besoin de lire / renvoyer des cookies ;
- lignes 44-46 : on vérifie qu'il y a bien une session en cours valide ;
- ligne 51 : cas du GET. On renvoie les cookies reçus ;
- ligne 54 : cas du POST. Celui-ci peut avoir deux types de paramètres :
  - le type **[x-www-form-urlencoded]**. C'est le cas des URL **[/calculer-impot]** et **[/authentifier-utilisateur]**. On utilise alors le paramètre **[data_value]** reçu par la méthode ;
  - le type **[json]**. C'est le cas de l'URL **[/calculer-impots]**. On utilise alors le paramètre **[json_value]** reçu par la méthode ;

  Là également, le cookie de session est renvoyé.
- lignes 56-62 : si on est en mode **[debug]**, la réponse du serveur est loguée. Ce log est important car il permet de savoir exactement ce qu'a renvoyé le serveur ;
- lignes 64-68 : selon que l'on est en mode json ou xml, la réponse texte du serveur est transformée en dictionnaire. Prenons l'exemple de l'URL **[/init-session]** :

  La réponse jSON est la suivante :

  ```
  2020-08-03 11:45:21.218116, MainThread : {"action": "init-session", "état": 700, "réponse": ["session démarrée avec le type de réponse json"]}
  ```

  La réponse XML est la suivante :

  ```
  2020-08-03 11:45:54.671871, MainThread : <?xml version="1.0" encoding="utf-8"?>
  <root><action>init-session</action><état>700</état><réponse>session démarrée avec le type de réponse xml</réponse></root>
  ```

  Le code des lignes 64-68 assure que dans les deux cas, on obtient dans **[résultat]** un dictionnaire avec les clés **[action, état, réponse]** ;
- lignes 70-72 : si la réponse contient des cookies on les récupère. Il faudra les renvoyer lors de la prochaine requête ;
- lignes 74-79 : si le statut HTTP de la réponse est différent de 200, on lève une exception avec le message d'erreur contenu dans résultat**['réponse']**. Ce peut être une erreur ou une liste d'erreurs ;
- lignes 81-82 : on rend la réponse du serveur au code appelant ;

[init_session]

- ligne 84 : la méthode **[init_session]** sert à fixer le type de session json ou xml que veut commencer le client avec le serveur ;
- ligne 86 : on note le type de session désiré au sein de la classe. En effet, toutes les méthodes ont besoin de cette information pour décoder correctement la réponse du serveur ;
- lignes 88-90 : à l'aide de la configuration de l'application, on détermine l'URL de service qu'il faut interroger ;
- ligne 93 : l'URL de service est interrogée. On ne récupère pas le résultat de la méthode **[get_response]** :
  - si elle lance une exception, alors l'opération a échoué. L'exception n'est pas gérée ici et remontera directement au code appelant qui arrêtera alors le client avec un message d'erreur ;
  - si elle ne lance pas d'exception, alors c'est que l'initialisation de la session a réussi ;

[authenticate_user]

```
1.     def authenticate_user(self, user: str, password: str):
2.         # url de service
```

```
3.          config_server = self.__config_server
4.          url_service = f"{config_server['urlServer']}{self.__config_services['authenticate-user']}"
5.          post_params = {
6.              "user": user,
7.              "password": password
8.          }
9.
10.         # exécution requête
11.         self.get_response("POST", url_service, post_params)
```

- la méthode **[authenticate_user]** sert à s'authentifier auprès du serveur. Pour cela elle reçoit les identifiants de connexion **[user, password]** en ligne 1 ;
- lignes 2-4 : on détermine l'URL de service à interroger ;
- lignes 5-8 : les paramètres du POST puisque l'URL **[/authentifier-utilisateur]** attend un POST avec des paramètres **[user, password]** ;
- ligne 11 : la requête est exécutée. Là encore on ne récupère pas la réponse du serveur. C'est l'exception lancée par **[get_response]** qui indique si on a réussi ou pas ;

[calculate_tax]

```
1.      def calculate_tax(self, taxpayer: TaxPayer):
2.          # url de service
3.          config_server = self.__config_server
4.          url_service = f"{config_server['urlServer']}{self.__config_services['calculate-tax']}"
5.          # paramètres du POST
6.          post_params = {
7.              "marié": taxpayer.marié,
8.              "enfants": taxpayer.enfants,
9.              "salaire": taxpayer.salaire
10.         }
11.
12.         # exécution requête
13.         response = self.get_response("POST", url_service, post_params)
14.         # mise à jour du TaxPayer avec la réponse
15.         taxpayer.fromdict(response)
```

- la méthode **[calculate_tax]** permet de calculer l'impôt d'un contribuable **[taxpayer]** passé en paramètre. Ce paramètre est modifié par la méthode (ligne 15) et constitue donc le résultat de la méthode ;
- lignes 2-4 : on définit l'URL de service à interroger ;
- lignes 6-10 : les paramètres du POST à émettre. En effet, l'URL de service **[/calculer-impot]** attend un POST avec les paramètres **[marié, enfants, salaire]** ;
- lignes 12-13 : la requête est exécutée et la réponse du serveur récupérée. L'URL de service **[/calculer-impot]** renvoie un dictionnaire avec les clés **[impôt, décôte, surcôte, réduction, taux]** de l'impôt ;
- ligne 15 : le dictionnaire **[response]** obtenu est utilisé pour mettre à jour le contribuable **[taxpayer]** ;

[calculate_tax_in_bulk_mode]

```
1.      # calcul de l'impôt en mode bulk
2.      def calculate_tax_in_bulk_mode(self, taxpayers: list):
3.          # on laisse remonter les exceptions
4.
5.          # on transforme les taxpayers en liste de dictionnaires
6.          # on ne garde que les propriétés [marié, enfants, salaire]
7.          list_dict_taxpayers = list(
8.              map(lambda taxpayer:
9.                  taxpayer.asdict(included_keys=[
10.                         '_TaxPayer__marié',
11.                         '_TaxPayer__enfants',
12.                         '_TaxPayer__salaire']),
13.                 taxpayers))
14.
15.         # url de service
16.         config_server = self.__config_server
17.         url_service = f"{config_server['urlServer']}{self.__config_services['calculate-tax-in-bulk-mode']}"
18.
19.         # exécution requête
20.         list_dict_taxpayers2 = self.get_response("POST", url_service, data_value=None, json_value=list_dict_taxpayers)
21.         # lorsqu'il n'y a qu'un contribuable et qu'on est dans une session xml, [list_dict_taxpayers2] n'est pas une lis
    te
22.         # dans ce cas on en fait une liste
23.         if not isinstance(list_dict_taxpayers2, list):
24.             list_dict_taxpayers2 = [list_dict_taxpayers2]
25.         # on met à jour la liste initiale des taxpayers avec les résultats reçus
26.         for i in range(len(taxpayers)):
27.             # mise à jour de taxpayers[i]
```

```
28.             taxpayers[i].fromdict(list_dict_taxpayers2[i])
29.         # ici le paramètre [taxpayers] a été mis à jour avec les résultats du serveur
```

- ligne 2 : la méthode reçoit une liste de contribuables de type TaxPayer ;
- lignes 7-13 : cette liste d'élements de type **[TaxPayer]** est transformée en liste de dictionnaires **[marié, enfants, salaire]** ;
- lignes 15-17 : on fixe l'URL de service ;
- lignes 19-20 : on exécute une requête POST dont le corps jSON est formé de la liste des dictionnaires créée ligne 7. On récupère la réponse du serveur ;
- lignes 23-24 : les tests ont montré un problème lorsque la session est de type XML :
  - si la liste initiale des contribuables a N éléments (N>1) on obtient comme résultat une liste de N dictionnaires de type **[OrderedDict]** ;
  - si la liste initiale n'a qu'un élément, on obtient non pas une liste mais un élément de type **[OrderedDict]** ;

  lignes 23-24 : si on est dans ce dernier cas (1 élément), on transforme le résultat en liste de 1 élément ;
- lignes 25-28 : cette liste de dictionnaires reçus contient l'impôt de chaque contribuable de la liste initiale. On met à jour alors chacun d'eux avec les résultats reçus ;

[get_simulations]

```
1.      def get_simulations(self) -> list:
2.          # url de service
3.          config_server = self.__config_server
4.          url_service = f"{config_server['urlServer']}{self.__config_services['get-simulations']}"
5.
6.          # exécution requête
7.          return self.get_response("GET", url_service)
```

- ligne 1 : la méthode demande la liste des simulations faites dans la session courante ;
- ligne 2 : la méthode rend la réponse du serveur ;

[delete_simulation]

```
1.      def delete_simulation(self, id: int) -> list:
2.          # url de service
3.          config_server = self.__config_server
4.          url_service = f"{config_server['urlServer']}{self.__config_services['delete-simulation']}/{id}"
5.
6.          # exécution requête
7.          return self.get_response("GET", url_service)
```

- ligne 1 : la méthode supprime la simulation dont on passe l'identifiant ;
- ligne 7 : elle rend la réponse du serveur, la liste des simulations restantes après la suppression demandée ;

[get-admindata]

```
1.      def get_admindata(self) -> AdminData:
2.          # on laisse remonter les exceptions
3.
4.          # url de service
5.          config_server = self.__config_server
6.          url_service = f"{config_server['urlServer']}{self.__config_services['get-admindata']}"
7.
8.          # exécution requête
9.          résultat = self.get_response("GET", url_service)
10.
11.         # résultat est un dictionnaire de valeurs de type str si session xml
12.         if self.__session_type == 'xml':
13.             # nouveau dictionnaire
14.             résultat2 = {}
15.             # on passe tout en numérique
16.             for key, value in résultat.items():
17.                 # certains éléments du dictionnaire sont des listes
18.                 if isinstance(value, list):
19.                     values = []
20.                     for value2 in value:
21.                         values.append(float(value2))
22.                     résultat2[key] = values
23.                 else:
24.                     # d'autres de simples éléments
25.                     résultat2[key] = float(value)
26.         else:
27.             résultat2 = résultat
```

```
28.         # résultat de type AdminData
29.         return AdminData().fromdict(résultat2)
```

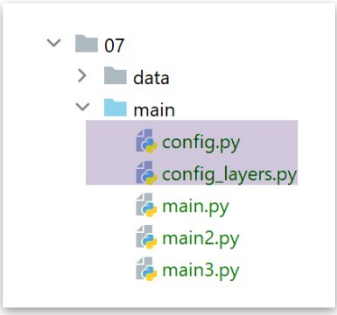
- ligne 1 : la méthode demande au serveur les constantes fiscales permettant le calcul de l'impôt ;
- ligne 29 : elle rend un type **[AdminData]** ;
- ligne 9 : on récupère la réponse du serveur sous la forme d'un dictionnaire. Les tests montrent qu'il y a un problème lorsque la session est une session XML : au lieu d'être des valeurs numériques, les valeurs du dictionnaire sont des chaînes de caractères. Nous avions signalé ce problème lors de l'étude du module **[xmltodict]** et constaté que c'était un fonctionnement normal. **[xmltodict]** n'a pas d'informations de types dans le flux XML qu'on lui donne. Ceci dit, dans ce cas précis, il faut convertir en numérique toutes les valeurs du dictionnaire reçu. Celui-ci contient trois listes **[limites, coeffr, coeffn]** et une suite de propriétés numériques ;
- lignes 13-25 : création d'un dictionnaire **[résultat2]** à valeurs numériques à partir du dictionnaire **[résultat]** à valeurs de type chaîne de caractères ;
- ligne 29 : le dictionnaire **[resultat2]** est utilisé pour initialiser un type **[AdminData]** ;

31.2.3 La factory de la couche [dao]

Nos clients seront multi-threadés. Comme la couche **[dao]** est implémentée par une classe ayant un état en lecture / écriture (= des propriétés en lecture / écriture), chaque thread doit avoir sa propre couche **[dao]** ou alors il faut synchroniser l'accès aux données partagées entre les threads. Ici nous prenons la première solution. Nous utilisons une classe **[ImpôtsDaoWithHttpSessionFactory]** capable de créer des instances de la couche **[dao]** :

```
1.  from ImpôtsDaoWithHttpSession import ImpôtsDaoWithHttpSession
2.
3.  class ImpôtsDaoWithHttpSessionFactory:
4.
5.      def __init__(self, config: dict):
6.          # on mémorise le paramètre
7.          self.__config = config
8.
9.      def new_instance(self):
10.         # on rend une instance de la couche [dao]
11.         return ImpôtsDaoWithHttpSession(self.__config)
```

31.3 Configuration des clients



Les clients sont configurés par les fichiers **[config]** et **[config_layers]**. Le fichier **[config]** est le suivant :

```
1.  def configure(config: dict) -> dict:
2.      import os
3.
4.      # étape 1 ------
5.
6.      # dossier de ce fichier
7.      script_dir = os.path.dirname(os.path.abspath(__file__))
8.
9.      # chemin racine
10.     root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
11.
12.     # dépendances absolues
13.     absolute_dependencies = [
14.         # dossiers du projet
15.         # BaseEntity, MyException
16.         f"{root_dir}/classes/02/entities",
17.         # InterfaceImpôtsDao, InterfaceImpôtsMétier, InterfaceImpôtsUi
```

```
        f"{root_dir}/impots/v04/interfaces",
        # AbstractImpôtsdao, ImpôtsConsole, ImpôtsMétier
        f"{root_dir}/impots/v04/services",
        # ImpotsDaoWithAdminDataInDatabase
        f"{root_dir}/impots/v05/services",
        # AdminData, ImpôtsError, TaxPayer
        f"{root_dir}/impots/v04/entities",
        # Constantes, tranches
        f"{root_dir}/impots/v05/entities",
        # ImpôtsDaoWithHttpSession, ImpôtsDaoWithHttpSessionFactory, InterfaceImpôtsDaoWithHttpSession
        f"{script_dir}/../services",
        # scripts de configuration
        script_dir,
        # Logger
        f"{root_dir}/impots/http-servers/02/utilities",
    ]

    # on fixe le syspath
    from myutils import set_syspath
    set_syspath(absolute_dependencies)

    # étape 2 ------
    # configuration de l'application avec des constantes
    config.update({
        # fichier des contribuables
        "taxpayersFilename": f"{script_dir}/../data/input/taxpayersdata.txt",
        # fichier des résultats
        "resultsFilename": f"{script_dir}/../data/output/résultats.json",
        # fichier des erreurs
        "errorsFilename": f"{script_dir}/../data/output/errors.txt",
        # fichier de logs
        "logsFilename": f"{script_dir}/../data/logs/logs.txt",
        # le serveur de calcul de l'impôt
        "server": {
            "urlServer": "http://127.0.0.1:5000",
            "user": {
                "login": "admin",
                "password": "admin"
            },
            "url_services": {
                "calculate-tax": "/calculer-impot",
                "get-admindata": "/get-admindata",
                "calculate-tax-in-bulk-mode": "/calculer-impots",
                "init-session": "/init-session",
                "end-session": "/fin-session",
                "authenticate-user": "/authentifier-utilisateur",
                "get-simulations": "/lister-simulations",
                "delete-simulation": "/supprimer-simulation"
            }
        },
        # mode debug
        "debug": True
    }
    )

    # étape 3 ------
    # instanciation des couches
    import config_layers
    config['layers'] = config_layers.configure(config)

    # on rend la configuation
    return config
```

Le fichier **[config_layers]** est le suivant :

```
def configure(config: dict) -> dict:
    # instanciation des couches de l'applicatuon

    # couche [métier]
    from ImpôtsMétier import ImpôtsMétier
    métier = ImpôtsMétier()

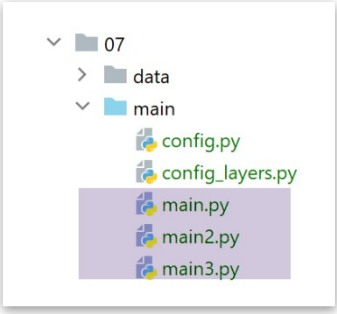
    # factory de la couche dao
    from ImpôtsDaoWithHttpSessionFactory import ImpôtsDaoWithHttpSessionFactory
    dao_factory = ImpôtsDaoWithHttpSessionFactory(config)

    # on rend la configuration des couches
```

```
    return {
        "dao_factory": dao_factory,
        "métier": métier
    }
```

- les clients n'auront pas un accès direct à la couche **[dao]**. Pour en avoir une, ils devront passer par la factory de la couche **[dao]** ;

31.4 Le client [main]



Le client **[main]** permet de tester les URL **[/init-session, /authentifier-utilisateur, /calculer-impots, /fin-session]** :

```
# on attend un paramètre json ou xml
import sys

syntaxe = f"{sys.argv[0]} json / xml"
erreur = len(sys.argv) != 2
if not erreur:
    session_type = sys.argv[1].lower()
    erreur = session_type != "json" and session_type != "xml"
if erreur:
    print(f"syntaxe : {syntaxe}")
    sys.exit()

# on configure l'application
import config
config = config.configure({"session_type": session_type})

# dépendances
from ImpôtsError import ImpôtsError
import random
import sys
import threading
from Logger import Logger

# exécution de la couche [dao] dans un thread
# taxpayers est une liste de contribuables
def thread_function(config: dict, taxpayers: list):
    # on récupère la factory de la couche [dao]
    dao_factory = config['layers']['dao_factory']
    # on crée une instance de couche [dao]
    dao = dao_factory.new_instance()
    # type de la session
    session_type = config['session_type']
    # nombre de contribuables
    nb_taxpayers = len(taxpayers)
    # log
    logger.write(f"début du calcul de l'impôt des {nb_taxpayers} contribuables\n")
    # on initialise la session
    dao.init_session(session_type)
    # on s'authentifie
    dao.authenticate_user(config['server']['user']['login'], config['server']['user']['password'])
    # on calcule l'impôt des contribuables
    dao.calculate_tax_in_bulk_mode(taxpayers)
    # fin de session
    dao.end_session()
    # log
    logger.write(f"fin du calcul de l'impôt des {nb_taxpayers} contribuables\n")

# liste des threads du client
threads = []
logger = None
# code
try:
    # logger
    logger = Logger(config["logsFilename"])
```

```
55.     # on le mémorise dans la config
56.     config["logger"] = logger
57.     # log de début
58.     logger.write("début du calcul de l'impôt des contribuables\n")
59.     # on récupère la factory de la couche [dao]
60.     dao_factory = config["layers"]["dao_factory"]
61.     # on crée une instance de la couche [dao]
62.     dao = dao_factory.new_instance()
63.     # lecture des données des contribuables
64.     taxpayers = dao.get_taxpayers_data()["taxpayers"]
65.     # des contribuables ?
66.     if not taxpayers:
67.         raise ImpôtsError(36, f"Pas de contribuables valides dans le fichier {config['taxpayersFilename']}")
68.     # calcul de l'impôt des contribuables avec plusieurs threads
69.     i = 0
70.     l_taxpayers = len(taxpayers)
71.     while i < len(taxpayers):
72.         # chaque thread va traiter de 1 à 4 contribuables
73.         nb_taxpayers = min(l_taxpayers - i, random.randint(1, 4))
74.         # la liste des contribuables traités par le thread
75.         thread_taxpayers = taxpayers[slice(i, i + nb_taxpayers)]
76.         # on incrémente i pour le thread suivant
77.         i += nb_taxpayers
78.         # on crée le thread
79.         thread = threading.Thread(target=thread_function, args=(config, thread_taxpayers))
80.         # on l'ajoute à la liste des threads du script principal
81.         threads.append(thread)
82.         # on lance le thread - cette opération est asynchrone - on n'attend pas le résultat du thread
83.         thread.start()
84.     # le thread principal attend la fin de tous les threads qu'il a lancés
85.     for thread in threads:
86.         thread.join()
87.     # ici tous les threads ont fini leur travail - chacun a modifié un ou plusieurs objets [taxpayer]
88.     # on enregistre les résultats dans le fichier jSON
89.     dao.write_taxpayers_results(taxpayers)
90.     # fin
91. except BaseException as erreur:
92.     # affichage de l'erreur
93.     print(f"L'erreur suivante s'est produite : {erreur}")
94. finally:
95.     # on ferme le logueur
96.     if logger:
97.         # log de fin
98.         logger.write("fin du calcul de l'impôt des contribuables\n")
99.         # fermeture du logueur
100.        logger.close()
101.    # on a fini
102.    print("Travail terminé...")
103.    # fin des threads qui pourraient encore exister si on s'est arrêté sur erreur
104.    sys.exit()
```

- lignes 4-11 : le client attend un paramètre lui indiquant le type de session, json ou xml, à utiliser avec le serveur ;
- lignes 13-15 : le client est configuré ;
- lignes 48-104 : ce code est connu. Il a été utilisé de nombreuses fois. Il répartit les contribuables pour lesquels on veut calculer l'impôt sur plusieurs threads ;
- ligne 26 : la méthode **[thread_function]** est la méthode exécutée par chaque thread pour calculer l'impôt des contribuables qui lui ont été attribués ;
- lignes 27-30 : chaque thread a sa propre couche **[dao]** ;
- le calcul de l'impôt se fait en quatre étapes :
  - lignes 37-38 : initialisation d'une session, json ou xml, avec le serveur ;
  - lignes 39-40 : authentification auprès du serveur ;
  - lignes 41-42 : calcul de l'impôt ;
  - lignes 43-44 : fermeture de la session avec le serveur ;

Lorsqu'on exécute ce code en mode **[json]**, on obtient les logs suivants :

```
1.  2020-08-03 14:28:34.320751, MainThread : début du calcul de l'impôt des contribuables
2.  2020-08-03 14:28:34.328749, Thread-1 : début du calcul de l'impôt des 4 contribuables
3.  2020-08-03 14:28:34.328749, Thread-2 : début du calcul de l'impôt des 4 contribuables
4.  2020-08-03 14:28:34.333592, Thread-3 : début du calcul de l'impôt des 3 contribuables
5.  2020-08-03 14:28:34.368651, Thread-2 : {"action": "init-session", "état": 700, "réponse": ["session démarrée avec le
    type de réponse json"]}
6.  2020-08-03 14:28:34.375699, Thread-1 : {"action": "init-session", "état": 700, "réponse": ["session démarrée avec le
    type de réponse json"]}
7.  2020-08-03 14:28:34.377432, Thread-3 : {"action": "init-session", "état": 700, "réponse": ["session démarrée avec le
    type de réponse json"]}
8.  2020-08-03 14:28:34.385653, Thread-2 : {"action": "authentifier-utilisateur", "état": 200, "réponse": "Authentification
    réussie"}
9.  2020-08-03 14:28:34.392656, Thread-1 : {"action": "authentifier-utilisateur", "état": 200, "réponse": "Authentification
    réussie"}
```

```
10. 2020-08-03 14:28:34.396377, Thread-3 : {"action": "authentifier-utilisateur", "état": 200, "réponse": "Authentification
    réussie"}
11. 2020-08-03 14:28:34.406528, Thread-2 : {"action": "calculer-impots", "état": 1500, "réponse": [{"marié": "non",
    "enfants": 3, "salaire": 100000, "impôt": 16782, "surcôte": 7176, "taux": 0.41, "décôte": 0, "réduction":
    0, "id": 1}, {"marié": "oui", "enfants": 3, "salaire": 100000, "impôt": 9200, "surcôte": 2180, "taux": 0.3,
    "décôte": 0, "réduction": 0, "id": 2}, {"marié": "oui", "enfants": 5, "salaire": 100000, "impôt": 4230,
    "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0, "id": 3}, {"marié": "non", "enfants": 0,
    "salaire": 100000, "impôt": 22986, "surcôte": 0, "taux": 0.41, "décôte": 0, "réduction": 0, "id": 4}]}
12. 2020-08-03 14:28:34.413837, Thread-1 : {"action": "calculer-impots", "état": 1500, "réponse": [{"marié": "oui",
    "enfants": 2, "salaire": 55555, "impôt": 2814, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0,
    "id": 1}, {"marié": "oui", "enfants": 2, "salaire": 50000, "impôt": 1384, "surcôte": 0, "taux": 0.14,
    "décôte": 384, "réduction": 347, "id": 2}, {"marié": "oui", "enfants": 3, "salaire": 50000, "impôt": 0,
    "surcôte": 0, "taux": 0.14, "décôte": 720, "réduction": 0, "id": 3}, {"marié": "non", "enfants": 2,
    "salaire": 100000, "impôt": 19884, "surcôte": 4480, "taux": 0.41, "décôte": 0, "réduction": 0, "id": 4}]}
13. 2020-08-03 14:28:34.416695, Thread-3 : {"action": "calculer-impots", "état": 1500, "réponse": [{"marié": "oui",
    "enfants": 2, "salaire": 30000, "impôt": 0, "surcôte": 0, "taux": 0.0, "décôte": 0, "réduction": 0, "id":
    1}, {"marié": "non", "enfants": 0, "salaire": 200000, "impôt": 64210, "surcôte": 7498, "taux": 0.45,
    "décôte": 0, "réduction": 0, "id": 2}, {"marié": "oui", "enfants": 3, "salaire": 200000, "impôt": 42842,
    "surcôte": 17283, "taux": 0.41, "décôte": 0, "réduction": 0, "id": 3}]}
14. 2020-08-03 14:28:34.425747, Thread-2 : {"action": "fin-session", "état": 400, "réponse": "session réinitialisée"}
15. 2020-08-03 14:28:34.425747, Thread-2 : fin du calcul de l'impôt des 4 contribuables
16. 2020-08-03 14:28:34.428956, Thread-1 : {"action": "fin-session", "état": 400, "réponse": "session réinitialisée"}
17. 2020-08-03 14:28:34.428956, Thread-1 : fin du calcul de l'impôt des 4 contribuables
18. 2020-08-03 14:28:34.428956, Thread-3 : {"action": "fin-session", "état": 400, "réponse": "session réinitialisée"}
19. 2020-08-03 14:28:34.428956, Thread-3 : fin du calcul de l'impôt des 3 contribuables
20. 2020-08-03 14:28:34.428956, MainThread : fin du calcul de l'impôt des contribuables
```

On voit ci-dessus le parcours du thread **[Thread-2]**.

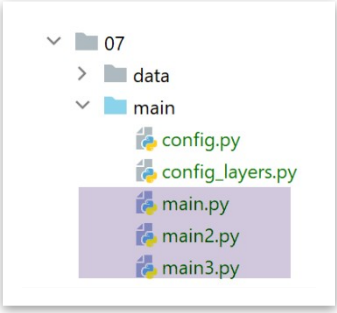
Si on exécute **[main]** en mode XML, les logs sont les suivants :

```
1.  2020-08-03 14:32:48.495316, MainThread : début du calcul de l'impôt des contribuables
2.  2020-08-03 14:32:48.496452, Thread-1 : début du calcul de l'impôt des 2 contribuables
3.  2020-08-03 14:32:48.498992, Thread-2 : début du calcul de l'impôt des 2 contribuables
4.  2020-08-03 14:32:48.498992, Thread-3 : début du calcul de l'impôt des 4 contribuables
5.  2020-08-03 14:32:48.498992, Thread-4 : début du calcul de l'impôt des 3 contribuables
6.  2020-08-03 14:32:48.538637, Thread-1 : <?xml version="1.0" encoding="utf-8"?>
7.  <root><action>init-session</action><état>700</état><réponse>session démarrée avec le type de réponse xml</réponse></root>
8.  2020-08-03 14:32:48.540783, Thread-4 : <?xml version="1.0" encoding="utf-8"?>
9.  <root><action>init-session</action><état>700</état><réponse>session démarrée avec le type de réponse xml</réponse></root>
10. 2020-08-03 14:32:48.547811, Thread-3 : <?xml version="1.0" encoding="utf-8"?>
11. <root><action>init-session</action><état>700</état><réponse>session démarrée avec le type de réponse xml</réponse></root>
12. 2020-08-03 14:32:48.547811, Thread-2 : <?xml version="1.0" encoding="utf-8"?>
13. <root><action>init-session</action><état>700</état><réponse>session démarrée avec le type de réponse xml</réponse></root>
14. 2020-08-03 14:32:48.555184, Thread-1 : <?xml version="1.0" encoding="utf-8"?>
15. <root><action>authentifier-utilisateur</action><état>200</état><réponse>Authentification réussie</réponse></root>
16. 2020-08-03 14:32:48.564793, Thread-2 : <?xml version="1.0" encoding="utf-8"?>
17. <root><action>authentifier-utilisateur</action><état>200</état><réponse>Authentification réussie</réponse></root>
18. 2020-08-03 14:32:48.564793, Thread-3 : <?xml version="1.0" encoding="utf-8"?>
19. <root><action>authentifier-utilisateur</action><état>200</état><réponse>Authentification réussie</réponse></root>
20. 2020-08-03 14:32:48.568333, Thread-4 : <?xml version="1.0" encoding="utf-8"?>
21. <root><action>authentifier-utilisateur</action><état>200</état><réponse>Authentification réussie</réponse></root>
22. 2020-08-03 14:32:48.568333, Thread-1 : <?xml version="1.0" encoding="utf-8"?>
23. <root><action>calculer-impots</action><état>1500</état><réponse><marié>oui</marié><enfants>2</enfants><salaire>55555</
    salaire><impôt>2814</impôt><surcôte>0</surcôte><taux>0.14</taux><décôte>0</décôte><réduction>0</réduction><id>1</id></
    réponse><réponse><marié>oui</marié><enfants>2</enfants><salaire>50000</salaire><impôt>1384</impôt><surcôte>0</
    surcôte><taux>0.14</taux><décôte>384</décôte><réduction>347</réduction><id>2</id></réponse></root>
24. 2020-08-03 14:32:48.579205, Thread-2 : <?xml version="1.0" encoding="utf-8"?>
25. <root><action>calculer-impots</action><état>1500</état><réponse><marié>oui</marié><enfants>3</enfants><salaire>50000</
    salaire><impôt>0</impôt><surcôte>0</surcôte><taux>0.14</taux><décôte>720</décôte><réduction>0</réduction><id>1</id></
    réponse><réponse><marié>non</marié><enfants>2</enfants><salaire>100000</salaire><impôt>19884</impôt><surcôte>4480</
    surcôte><taux>0.41</taux><décôte>0</décôte><réduction>0</réduction><id>2</id></réponse></root>
26. 2020-08-03 14:32:48.579205, Thread-3 : <?xml version="1.0" encoding="utf-8"?>
27. <root><action>calculer-impots</action><état>1500</état><réponse><marié>non</marié><enfants>3</enfants><salaire>100000</
    salaire><impôt>16782</impôt><surcôte>7176</surcôte><taux>0.41</taux><décôte>0</décôte><réduction>0</réduction><id>1</id></
    réponse><réponse><marié>oui</marié><enfants>3</enfants><salaire>100000</salaire><impôt>9200</impôt><surcôte>2180</
    surcôte><taux>0.3</taux><décôte>0</décôte><réduction>0</réduction><id>2</id></réponse><réponse><marié>oui</
    marié><enfants>5</enfants><salaire>100000</salaire><impôt>4230</impôt><surcôte>0</surcôte><taux>0.14</taux><décôte>0</
    décôte><réduction>0</réduction><id>3</id></réponse><réponse><marié>non</marié><enfants>0</enfants><salaire>100000</
    salaire><impôt>22986</impôt><surcôte>0</surcôte><taux>0.41</taux><décôte>0</décôte><réduction>0</réduction><id>4</id></
    réponse></root>
28. 2020-08-03 14:32:48.588051, Thread-4 : <?xml version="1.0" encoding="utf-8"?>
29. <root><action>calculer-impots</action><état>1500</état><réponse><marié>oui</marié><enfants>2</enfants><salaire>30000</
    salaire><impôt>0</impôt><surcôte>0</surcôte><taux>0.0</taux><décôte>0</décôte><réduction>0</réduction><id>1</id></
    réponse><réponse><marié>non</marié><enfants>0</enfants><salaire>200000</salaire><impôt>64210</impôt><surcôte>7498</
    surcôte><taux>0.45</taux><décôte>0</décôte><réduction>0</réduction><id>2</id></réponse><réponse><marié>oui</
    marié><enfants>3</enfants><salaire>200000</salaire><impôt>42842</impôt><surcôte>17283</surcôte><taux>0.41</
    taux><décôte>0</décôte><réduction>0</réduction><id>3</id></réponse></root>
30. 2020-08-03 14:32:48.594058, Thread-1 : <?xml version="1.0" encoding="utf-8"?>
31. <root><action>fin-session</action><état>400</état><réponse>session réinitialisée</réponse></root>
32. 2020-08-03 14:32:48.595198, Thread-1 : fin du calcul de l'impôt des 2 contribuables
33. 2020-08-03 14:32:48.595198, Thread-2 : <?xml version="1.0" encoding="utf-8"?>
34. <root><action>fin-session</action><état>400</état><réponse>session réinitialisée</réponse></root>
35. 2020-08-03 14:32:48.595198, Thread-2 : fin du calcul de l'impôt des 2 contribuables
```

```
2020-08-03 14:32:48.595198, Thread-3 : <?xml version="1.0" encoding="utf-8"?>
<root><action>fin-session</action><état>400</état><réponse>session réinitialisée</réponse></root>
2020-08-03 14:32:48.595198, Thread-3 : fin du calcul de l'impôt des 4 contribuables
2020-08-03 14:32:48.603351, Thread-4 : <?xml version="1.0" encoding="utf-8"?>
<root><action>fin-session</action><état>400</état><réponse>session réinitialisée</réponse></root>
2020-08-03 14:32:48.603351, Thread-4 : fin du calcul de l'impôt des 3 contribuables
2020-08-03 14:32:48.603351, MainThread : fin du calcul de l'impôt des contribuables
```

Ci-dessus, le parcours du thread **[Thread-2]**.

31.5 Le client [main2]



Le client **[main2]** permet de tester les URL **[/init-session, /authentifier-utilisateur, /get-admindata, /fin-session]** :

```
# on attend un paramètre json ou xml
import sys

syntaxe = f"{sys.argv[0]} json / xml"
erreur = len(sys.argv) != 2
if not erreur:
    session_type = sys.argv[1].lower()
    erreur = session_type != "json" and session_type != "xml"
if erreur:
    print(f"syntaxe : {syntaxe}")
    sys.exit()

# on configure l'application
import config
config = config.configure({"session_type": session_type})

# dépendances
from ImpôtsError import ImpôtsError
from Logger import Logger

logger = None
# code
try:
    # logger
    logger = Logger(config["logsFilename"])
    # on le mémorise dans la config
    config["logger"] = logger
    # log de début
    logger.write("début du calcul de l'impôt des contribuables\n")
    # on récupère la factory de la couche [dao]
    dao_factory = config['layers']['dao_factory']
    # on crée une instance de la couche [dao]
    dao = dao_factory.new_instance()
    # on récupère les contribuables
    taxpayers = dao.get_taxpayers_data()["taxpayers"]
    # des contribuables ?
    if not taxpayers:
        raise ImpôtsError(36, f"Pas de contribuables valides dans le fichier {config['taxpayersFilename']}")
    # type de la session
    session_type = config['session_type']
    # on initialise la session
    dao.init_session(session_type)
    # on s'authentifie
    dao.authenticate_user(config['server']['user']['login'], config['server']['user']['password'])
    # on récupère les données de l'administration fiscale
    admindata = dao.get_admindata()
    # fin de session
    dao.end_session()
    # calcul de l'impôt des contribuables par la couche [métier]
    métier = config['layers']['métier']
    for taxpayer in taxpayers:
        métier.calculate_tax(taxpayer, admindata)
```

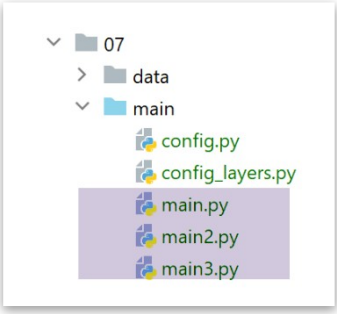
```
53.      # on enregistre les résultats dans le fichier jSON
54.      dao.write_taxpayers_results(taxpayers)
55.    except BaseException as erreur:
56.        # affichage de l'erreur
57.        print(f"L'erreur suivante s'est produite : {erreur}")
58.  finally:
59.      # on ferme le logueur
60.      if logger:
61.          # log de fin
62.          logger.write("fin du calcul de l'impôt des contribuables\n")
63.          # fermeture du logueur
64.          logger.close()
65.      # on a fini
66.      print("Travail terminé...")
```

- lignes 1-11 : on récupère le paramètre **[json, xml]** qui fixe le type de la session à établir avec le serveur ;
- lignes 13-15 : on configure le client ;
- lignes 30-33 : on crée une couche **[dao]** ;
- lignes 34-35 : avec elle, on récupère la liste des contribuables pour lesquels il faut calculer l'impôt ;
- on retrouve ensuite les quatre étapes du dialogue avec le serveur ;
  - lignes 41-42 : une session est démarrée avec le serveur ;
  - lignes 43-44 : on s'authentifie auprès du serveur ;
  - lignes 45-46 : on demande au serveur les constantes fiscales permettant le calcul de l'impôt ;
  - lignes 47-48 : on ferme la session avec le serveur ;
- lignes 49-52 : avec ces constantes, on est capables de calculer l'impôt des contribuables à l'aide de la couche **[métier]** locale au client ;
- lignes 53-54 : on enregistre les résultats obtenus ;

Pour une session XML, les résultats sont les suivants :

```
1.  2020-08-03 14:44:43.194294, MainThread : début du calcul de l'impôt des contribuables
2.  2020-08-03 14:44:43.231633, MainThread : <?xml version="1.0" encoding="utf-8"?>
3.  <root><action>init-session</action><état>700</état><réponse>session démarrée avec le type de réponse
    xml</réponse></root>
4.  2020-08-03 14:44:43.240872, MainThread : <?xml version="1.0" encoding="utf-8"?>
5.  <root><action>authentifier-utilisateur</action><état>200</état><réponse>Authentification
    réussie</réponse></root>
6.  2020-08-03 14:44:43.250061, MainThread : <?xml version="1.0" encoding="utf-8"?>
7.  <root><action>get-admindata</action><état>1000</état><réponse><limites>9964.0</limites><limites>27519.0</
    limites><limites>73779.0</limites><limites>156244.0</limites><limites>93749.0</limites><coeffr>0.0</
    coeffr><coeffr>0.14</coeffr><coeffr>0.3</coeffr><coeffr>0.41</coeffr><coeffr>0.45</coeffr><coeffn>0.0</
    coeffn><coeffn>1394.96</coeffn><coeffn>5798.0</coeffn><coeffn>13913.7</coeffn><coeffn>20163.4</
    coeffn><abattement_dixpourcent_min>437.0</
    abattement_dixpourcent_min><plafond_impot_couple_pour_decote>2627.0</
    plafond_impot_couple_pour_decote><plafond_decote_couple>1970.0</
    plafond_decote_couple><valeur_reduc_demi_part>3797.0</
    valeur_reduc_demi_part><plafond_revenus_celibataire_pour_reduction>21037.0</
    plafond_revenus_celibataire_pour_reduction><id>1</id><abattement_dixpourcent_max>12502.0</
    abattement_dixpourcent_max><plafond_impot_celibataire_pour_decote>1595.0</
    plafond_impot_celibataire_pour_decote><plafond_decote_celibataire>1196.0</
    plafond_decote_celibataire><plafond_revenus_couple_pour_reduction>42074.0</
    plafond_revenus_couple_pour_reduction><plafond_qf_demi_part>1551.0</plafond_qf_demi_part></réponse></root>
8.  2020-08-03 14:44:43.269850, MainThread : <?xml version="1.0" encoding="utf-8"?>
9.  <root><action>fin-session</action><état>400</état><réponse>session réinitialisée</réponse></root>
10. 2020-08-03 14:44:43.269850, MainThread : fin du calcul de l'impôt des contribuables
```

31.6 Le client [main3]



Le client **[main3]** permet de tester les URL **[/init-session, /calculer-impots, /get-simulations, /delete-simulation, /fin-session]** :

```
1.  # on attend un paramètre json ou xml
2.  import sys
3.
4.  syntaxe = f"{sys.argv[0]} json / xml"
5.  erreur = len(sys.argv) != 2
6.  if not erreur:
7.      session_type = sys.argv[1].lower()
8.      erreur = session_type != "json" and session_type != "xml"
9.  if erreur:
10.     print(f"syntaxe : {syntaxe}")
11.     sys.exit()
12.
13. # on configure l'application
14. import config
15. config = config.configure({"session_type": session_type})
16.
17. # dépendances
18. from ImpôtsError import ImpôtsError
19. import sys
20. from Logger import Logger
21.
22. logger = None
23. # code
24. try:
25.     # logger
26.     logger = Logger(config["logsFilename"])
27.     # on le mémorise dans la config
28.     config["logger"] = logger
29.     # log de début
30.     logger.write("début du calcul de l'impôt des contribuables\n")
31.     # on récupère la factory de la couche [dao]
32.     dao_factory = config["layers"]["dao_factory"]
33.     # on crée une instance de la couche [dao]
34.     dao = dao_factory.new_instance()
35.     # lecture des données des contribuables
36.     taxpayers = dao.get_taxpayers_data()["taxpayers"]
37.     # des contribuables ?
38.     if not taxpayers:
39.         raise ImpôtsError(36, f"Pas de contribuables valides dans le fichier {config['taxpayersFilename']}")
40.     # calcul de l'impôt des contribuables
41.     # nombre de contribuables
42.     nb_taxpayers = len(taxpayers)
43.     # log
44.     logger.write(f"début du calcul de l'impôt des {nb_taxpayers} contribuables\n")
45.     # on initialise la session
46.     dao.init_session(session_type)
47.     # on s'authentifie
48.     dao.authenticate_user(config['server']['user']['login'], config['server']['user']['password'])
49.     # on calcule l'impôt des contribuables
50.     dao.calculate_tax_in_bulk_mode(taxpayers)
51.     # on demande la liste des simulations
52.     simulations = dao.get_simulations()
53.     # on en supprime une sur deux
54.     for i in range(len(simulations)):
55.         if i % 2 == 0:
56.             # on supprime la simulation
57.             dao.delete_simulation(simulations[i]['id'])
58.     # fin de session
59.     dao.end_session()
60.     # consultez les logs pour voir les différents résultats (mode debug=True)
61. except BaseException as erreur:
62.     # affichage de l'erreur
63.     print(f"L'erreur suivante s'est produite : {erreur}")
64. finally:
65.     # on ferme le logueur
66.     if logger:
67.         # log de fin
68.         logger.write("fin du calcul de l'impôt des contribuables\n")
69.         # fermeture du logueur
70.         logger.close()
71.     # on a fini
72.     print("Travail terminé...")
```

- lignes 1-11 : on récupère le type de la session dans les paramètres du script ;
- lignes 13-15 : on configure l'application ;
- lignes 25-50 : on retrouve du code déjà expliqué à un moment ou à un autre ;
- lignes 51-52 : on demande la liste des simulations faites dans la session courante ;
- lignes 53-57 : on supprime une simulation sur deux ;
- lignes 58-59 : on termine la session ;

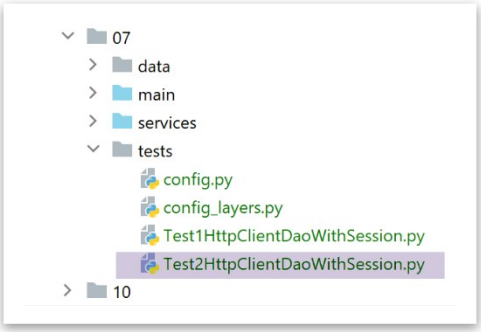
Lors d'une session jSON, les logs sont les suivants :

```
2020-08-03 15:01:52.702297, MainThread : début du calcul de l'impôt des contribuables
2020-08-03 15:01:52.702297, MainThread : début du calcul de l'impôt des 11 contribuables
2020-08-03 15:01:52.734806, MainThread : {"action": "init-session", "état": 700, "réponse": ["session démarrée avec le type de réponse json"]}
2020-08-03 15:01:52.747961, MainThread : {"action": "authentifier-utilisateur", "état": 200, "réponse": "Authentification réussie"}
2020-08-03 15:01:52.765721, MainThread : {"action": "calculer-impots", "état": 1500, "réponse": [{"marié": "oui", "enfants": 2, "salaire": 55555, "impôt": 2814, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0, "id": 1}, {"marié": "oui", "enfants": 2, "salaire": 50000, "impôt": 1384, "surcôte": 0, "taux": 0.14, "décôte": 384, "réduction": 347, "id": 2}, {"marié": "oui", "enfants": 3, "salaire": 50000, "impôt": 0, "surcôte": 0, "taux": 0.14, "décôte": 720, "réduction": 0, "id": 3}, {"marié": "non", "enfants": 2, "salaire": 100000, "impôt": 19884, "surcôte": 4480, "taux": 0.41, "décôte": 0, "réduction": 0, "id": 4}, {"marié": "non", "enfants": 3, "salaire": 100000, "impôt": 16782, "surcôte": 7176, "taux": 0.41, "décôte": 0, "réduction": 0, "id": 5}, {"marié": "oui", "enfants": 3, "salaire": 100000, "impôt": 9200, "surcôte": 2180, "taux": 0.3, "décôte": 0, "réduction": 0, "id": 6}, {"marié": "oui", "enfants": 5, "salaire": 100000, "impôt": 4230, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction": 0, "id": 7}, {"marié": "non", "enfants": 0, "salaire": 100000, "impôt": 22986, "surcôte": 0, "taux": 0.41, "décôte": 0, "réduction": 0, "id": 8}, {"marié": "oui", "enfants": 2, "salaire": 30000, "impôt": 0, "surcôte": 0, "taux": 0.0, "décôte": 0, "réduction": 0, "id": 9}, {"marié": "non", "enfants": 0, "salaire": 200000, "impôt": 64210, "surcôte": 7498, "taux": 0.45, "décôte": 0, "réduction": 0, "id": 10}, {"marié": "oui", "enfants": 3, "salaire": 200000, "impôt": 42842, "surcôte": 17283, "taux": 0.41, "décôte": 0, "réduction": 0, "id": 11}]}
2020-08-03 15:01:52.785505, MainThread : {"action": "lister-simulations", "état": 500, "réponse": [{"décôte": 0, "enfants": 2, "id": 1, "impôt": 2814, "marié": "oui", "réduction": 0, "salaire": 55555, "surcôte": 0, "taux": 0.14}, {"décôte": 384, "enfants": 2, "id": 2, "impôt": 1384, "marié": "oui", "réduction": 347, "salaire": 50000, "surcôte": 0, "taux": 0.14}, {"décôte": 720, "enfants": 3, "id": 3, "impôt": 0, "marié": "oui", "réduction": 0, "salaire": 50000, "surcôte": 0, "taux": 0.14}, {"décôte": 0, "enfants": 2, "id": 4, "impôt": 19884, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 4480, "taux": 0.41}, {"décôte": 0, "enfants": 3, "id": 5, "impôt": 16782, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 7176, "taux": 0.41}, {"décôte": 0, "enfants": 3, "id": 6, "impôt": 9200, "marié": "oui", "réduction": 0, "salaire": 100000, "surcôte": 2180, "taux": 0.3}, {"décôte": 0, "enfants": 5, "id": 7, "impôt": 4230, "marié": "oui", "réduction": 0, "salaire": 100000, "surcôte": 0, "taux": 0.14}, {"décôte": 0, "enfants": 0, "id": 8, "impôt": 22986, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 0, "taux": 0.41}, {"décôte": 0, "enfants": 2, "id": 9, "impôt": 0, "marié": "oui", "réduction": 0, "salaire": 30000, "surcôte": 0, "taux": 0.0}, {"décôte": 0, "enfants": 0, "id": 10, "impôt": 64210, "marié": "non", "réduction": 0, "salaire": 200000, "surcôte": 7498, "taux": 0.45}, {"décôte": 0, "enfants": 3, "id": 11, "impôt": 42842, "marié": "oui", "réduction": 0, "salaire": 200000, "surcôte": 17283, "taux": 0.41}]}
2020-08-03 15:01:52.801475, MainThread : {"action": "supprimer-simulation", "état": 600, "réponse": [{"décôte": 384, "enfants": 2, "id": 2, "impôt": 1384, "marié": "oui", "réduction": 347, "salaire": 50000, "surcôte": 0, "taux": 0.14}, {"décôte": 720, "enfants": 3, "id": 3, "impôt": 0, "marié": "oui", "réduction": 0, "salaire": 50000, "surcôte": 0, "taux": 0.14}, {"décôte": 0, "enfants": 2, "id": 4, "impôt": 19884, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 4480, "taux": 0.41}, {"décôte": 0, "enfants": 3, "id": 5, "impôt": 16782, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 7176, "taux": 0.41}, {"décôte": 0, "enfants": 3, "id": 6, "impôt": 9200, "marié": "oui", "réduction": 0, "salaire": 100000, "surcôte": 2180, "taux": 0.3}, {"décôte": 0, "enfants": 5, "id": 7, "impôt": 4230, "marié": "oui", "réduction": 0, "salaire": 100000, "surcôte": 0, "taux": 0.14}, {"décôte": 0, "enfants": 0, "id": 8, "impôt": 22986, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 0, "taux": 0.41}, {"décôte": 0, "enfants": 2, "id": 9, "impôt": 0, "marié": "oui", "réduction": 0, "salaire": 30000, "surcôte": 0, "taux": 0.0}, {"décôte": 0, "enfants": 0, "id": 10, "impôt": 64210, "marié": "non", "réduction": 0, "salaire": 200000, "surcôte": 7498, "taux": 0.45}, {"décôte": 0, "enfants": 3, "id": 11, "impôt": 42842, "marié": "oui", "réduction": 0, "salaire": 200000, "surcôte": 17283, "taux": 0.41}]}
2020-08-03 15:01:52.810129, MainThread : {"action": "supprimer-simulation", "état": 600, "réponse": [{"décôte": 384, "enfants": 2, "id": 2, "impôt": 1384, "marié": "oui", "réduction": 347, "salaire": 50000, "surcôte": 0, "taux": 0.14}, {"décôte": 0, "enfants": 2, "id": 4, "impôt": 19884, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 4480, "taux": 0.41}, {"décôte": 0, "enfants": 3, "id": 5, "impôt": 16782, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 7176, "taux": 0.41}, {"décôte": 0, "enfants": 3, "id": 6, "impôt": 9200, "marié": "oui", "réduction": 0, "salaire": 100000, "surcôte": 2180, "taux": 0.3}, {"décôte": 0, "enfants": 5, "id": 7, "impôt": 4230, "marié": "oui", "réduction": 0, "salaire": 100000, "surcôte": 0, "taux": 0.14}, {"décôte": 0, "enfants": 0, "id": 8, "impôt": 22986, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 0, "taux": 0.41}, {"décôte": 0, "enfants": 2, "id": 9, "impôt": 0, "marié": "oui", "réduction": 0, "salaire": 30000, "surcôte": 0, "taux": 0.0}, {"décôte": 0, "enfants": 0, "id": 10, "impôt": 64210, "marié": "non", "réduction": 0, "salaire": 200000, "surcôte": 7498, "taux": 0.45}, {"décôte": 0, "enfants": 3, "id": 11, "impôt": 42842, "marié": "oui", "réduction": 0, "salaire": 200000, "surcôte": 17283, "taux": 0.41}]}
2020-08-03 15:01:52.818803, MainThread : {"action": "supprimer-simulation", "état": 600, "réponse": [{"décôte": 384, "enfants": 2, "id": 2, "impôt": 1384, "marié": "oui", "réduction": 347, "salaire": 50000, "surcôte": 0, "taux": 0.14}, {"décôte": 0, "enfants": 2, "id": 4, "impôt": 19884, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 4480, "taux": 0.41}, {"décôte": 0, "enfants": 3, "id": 6, "impôt": 9200, "marié": "oui", "réduction": 0, "salaire": 100000, "surcôte": 2180, "taux": 0.3}, {"décôte": 0, "enfants": 5, "id": 7, "impôt": 4230, "marié": "oui", "réduction": 0, "salaire": 100000, "surcôte": 0, "taux": 0.14}, {"décôte": 0, "enfants": 0, "id": 8, "impôt": 22986, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 0, "taux": 0.41}, {"décôte": 0, "enfants": 2, "id": 9, "impôt": 0, "marié": "oui", "réduction": 0, "salaire": 30000, "surcôte": 0, "taux": 0.0}, {"décôte": 0, "enfants": 0, "id": 10, "impôt": 64210, "marié": "non", "réduction": 0, "salaire": 200000, "surcôte": 7498, "taux": 0.45}, {"décôte": 0, "enfants": 3, "id": 11, "impôt": 42842, "marié": "oui", "réduction": 0, "salaire": 200000, "surcôte": 17283, "taux": 0.41}]}
```

```
10. 2020-08-03 15:01:52.834604, MainThread : {"action": "supprimer-simulation", "état": 600, "réponse":
    [{"décôte": 384, "enfants": 2, "id": 2, "impôt": 1384, "marié": "oui", "réduction": 347, "salaire": 50000,
    "surcôte": 0, "taux": 0.14}, {"décôte": 0, "enfants": 2, "id": 4, "impôt": 19884, "marié": "non",
    "réduction": 0, "salaire": 100000, "surcôte": 4480, "taux": 0.41}, {"décôte": 0, "enfants": 3, "id": 6,
    "impôt": 9200, "marié": "oui", "réduction": 0, "salaire": 100000, "surcôte": 2180, "taux": 0.3}, {"décôte":
    0, "enfants": 0, "id": 8, "impôt": 22986, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 0,
    "taux": 0.41}, {"décôte": 0, "enfants": 2, "id": 9, "impôt": 0, "marié": "oui", "réduction": 0, "salaire":
    30000, "surcôte": 0, "taux": 0.0}, {"décôte": 0, "enfants": 0, "id": 10, "impôt": 64210, "marié": "non",
    "réduction": 0, "salaire": 200000, "surcôte": 7498, "taux": 0.45}, {"décôte": 0, "enfants": 3, "id": 11,
    "impôt": 42842, "marié": "oui", "réduction": 0, "salaire": 200000, "surcôte": 17283, "taux": 0.41}]}
11. 2020-08-03 15:01:52.843803, MainThread : {"action": "supprimer-simulation", "état": 600, "réponse":
    [{"décôte": 384, "enfants": 2, "id": 2, "impôt": 1384, "marié": "oui", "réduction": 347, "salaire": 50000,
    "surcôte": 0, "taux": 0.14}, {"décôte": 0, "enfants": 2, "id": 4, "impôt": 19884, "marié": "non",
    "réduction": 0, "salaire": 100000, "surcôte": 4480, "taux": 0.41}, {"décôte": 0, "enfants": 3, "id": 6,
    "impôt": 9200, "marié": "oui", "réduction": 0, "salaire": 100000, "surcôte": 2180, "taux": 0.3}, {"décôte":
    0, "enfants": 0, "id": 8, "impôt": 22986, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 0,
    "taux": 0.41}, {"décôte": 0, "enfants": 0, "id": 10, "impôt": 64210, "marié": "non", "réduction": 0,
    "salaire": 200000, "surcôte": 7498, "taux": 0.45}, {"décôte": 0, "enfants": 3, "id": 11, "impôt": 42842,
    "marié": "oui", "réduction": 0, "salaire": 200000, "surcôte": 17283, "taux": 0.41}]}
12. 2020-08-03 15:01:52.851855, MainThread : {"action": "supprimer-simulation", "état": 600, "réponse":
    [{"décôte": 384, "enfants": 2, "id": 2, "impôt": 1384, "marié": "oui", "réduction": 347, "salaire": 50000,
    "surcôte": 0, "taux": 0.14}, {"décôte": 0, "enfants": 2, "id": 4, "impôt": 19884, "marié": "non",
    "réduction": 0, "salaire": 100000, "surcôte": 4480, "taux": 0.41}, {"décôte": 0, "enfants": 3, "id": 6,
    "impôt": 9200, "marié": "oui", "réduction": 0, "salaire": 100000, "surcôte": 2180, "taux": 0.3}, {"décôte":
    0, "enfants": 0, "id": 8, "impôt": 22986, "marié": "non", "réduction": 0, "salaire": 100000, "surcôte": 0,
    "taux": 0.41}, {"décôte": 0, "enfants": 0, "id": 10, "impôt": 64210, "marié": "non", "réduction": 0,
    "salaire": 200000, "surcôte": 7498, "taux": 0.45}]}
13. 2020-08-03 15:01:52.863165, MainThread : {"action": "fin-session", "état": 400, "réponse": "session
    réinitialisée"}
14. 2020-08-03 15:01:52.863165, MainThread : fin du calcul de l'impôt des contribuables
```

- ligne 6 : nous avons 11 simulations ;
- ligne 12 : après les différentes suppressions, il n'y en a plus que 5 ;

31.7 La classe de test `[Test2HttpClientDaoWithSession]`



La classe **`[Test2HttpClientDaoWithSession]`** teste la couche **`[dao]`** des clients de la façon suivante :

```
1.  import unittest
2.
3.  from ImpôtsError import ImpôtsError
4.  from Logger import Logger
5.  from TaxPayer import TaxPayer
6.
7.  class Test2HttpClientDaoWithSession(unittest.TestCase):
8.
9.      def test_init_session_json(self) -> None:
10.         print('test_init_session_json')
11.         erreur = False
12.         try:
13.             dao.init_session('json')
14.         except ImpôtsError as ex:
15.             print(ex)
16.             erreur = True
17.         # il ne doit oas y avoir d'erreur
18.         self.assertFalse(erreur)
19.
20.     def test_init_session_xml(self) -> None:
21.         print('test_init_session_xml')
22.         erreur = False
```

```
        try:
            dao.init_session('xml')
        except ImpôtsError as ex:
            print(ex)
            erreur = True
        # in ne doit pas y avoir d'erreur
        self.assertFalse(erreur)

    def test_init_session_xxx(self) -> None:
        print('test_init_session_xxx')
        erreur = False
        try:
            dao.init_session('xxx')
        except ImpôtsError as ex:
            print(ex)
            erreur = True
        # il doit y avoir une erreur
        self.assertTrue(erreur)

    def test_authenticate_user_success(self) -> None:
        print('test_authenticate_user_success')
        # init session
        dao.init_session('json')
        # test
        erreur = False
        try:
            dao.authenticate_user(config['server']['user']['login'], config['server']['user']['password'])
        except ImpôtsError as ex:
            print(ex)
            erreur = True
        # il ne doit pas y avoir d'erreur
        self.assertFalse(erreur)

    def test_authenticate_user_failed(self) -> None:
        print('test_authenticate_user_failed')
        # init session
        dao.init_session('json')
        # test
        erreur = False
        try:
            dao.authenticate_user('x', 'y')
        except ImpôtsError as ex:
            print(ex)
            erreur = True
        # il doit y avoir une erreur
        self.assertTrue(erreur)

    def test_get_simulations(self) -> None:
        print('test_get_simulations')
        # init session
        dao.init_session('json')
        # authentification
        dao.authenticate_user('admin', 'admin')
        # calcul impôt
        # {'marié': 'oui', 'enfants': 2, 'salaire': 55555,
        # 'impôt': 2814, 'surcôte': 0, 'décôte': 0, 'réduction': 0, 'taux': 0.14}
        taxpayer = TaxPayer().fromdict({"marié": "oui", "enfants": 2, "salaire": 55555})
        dao.calculate_tax(taxpayer)
        # get_simulations
        simulations = dao.get_simulations()
        # vérifications
        # il doit y avoir 1 simulation
        self.assertEqual(1, len(simulations))
        simulation = simulations[0]
        # vérification de l'impôt calculé
        self.assertAlmostEqual(simulation['impôt'], 2815, delta=1)
        self.assertEqual(simulation['décôte'], 0)
        self.assertEqual(simulation['réduction'], 0)
        self.assertAlmostEqual(simulation['taux'], 0.14, delta=0.01)
        self.assertEqual(simulation['surcôte'], 0)

    def test_delete_simulation(self) -> None:
        print('test_delete_simulation')
        # init session
        dao.init_session('json')
        # authentification
        dao.authenticate_user('admin', 'admin')
```

```
100.        # calcul impôt
101.        taxpayer = TaxPayer().fromdict({"marié": "oui", "enfants": 2, "salaire": 55555})
102.        dao.calculate_tax(taxpayer)
103.        # get_simulations
104.        simulations = dao.get_simulations()
105.        # delete_simulation
106.        dao.delete_simulation(simulations[0]['id'])
107.        # get_simulations
108.        simulations = dao.get_simulations()
109.        # vérification - il ne doit plus y avoir de simulations
110.        self.assertEqual(0, len(simulations))
111.        # on supprime une simulation qui n'existe pas
112.        erreur = False
113.        try:
114.            dao.delete_simulation(100)
115.        except ImpôtsError as ex:
116.            print(ex)
117.            erreur = True
118.        # il doit y avoir une erreur
119.        self.assertTrue(erreur)
120.
121.if __name__ == '__main__':
122.    # on configure l'application
123.    import config
124.    config = config.configure({})
125.
126.    # logger
127.    logger = Logger(config["logsFilename"])
128.    # on le mémorise dans la config
129.    config["logger"] = logger
130.
131.    # couche [dao]
132.    dao_factory = config['layers']['dao_factory']
133.    dao = dao_factory.new_instance()
134.
135.    # on exécute les méthodes de test
136.    print("tests en cours...")
137.    unittest.main()
```

- la couche **[dao]** envoie une requête au serveur, réceptionne sa réponse et la met en forme pour la rendre au code appelant. Lorsque le serveur envoie un réponse avec un code de statut différent de 200, la couche **[dao]** lance une exception. Aussi un certain nombre de tests consiste à savoir s'il y a eu exception ou pas ;
- lignes 9-18 : on initialise une session jSON. On ne doit pas avoir d'erreur ;
- lignes 20-29 : on initialise une session XML. On ne doit pas avoir d'erreur ;
- lignes 31-40 : on initialise une session avec un type incorrect. on doit avoir une erreur ;
- lignes 42-54 : on s'authentifie avec les bons identifiants. On ne doit pas avoir d'erreur ;
- lignes 56-68 : on s'authentifie avec des identifiants incorrects. On doit avoir une erreur ;
- lignes 70-92 : on fait un calcul d'impôt puis on demande la liste des simulations. On doit en avoir une. Par ailleurs on verifie que cette simulation contient bien l'impôt demandé ;
- lignes 94-119 : on fait une simulation qu'on supprime ensuite. Puis on essaie ensuite de supprimer une simulation alors qu'il n'y en a plus. On doit avoir une erreur ;
- lignes 121-137 : le test est lancé comme un script console classique ;
- lignes 122-124 : on configure l'application ;
- lignes 126-129 : on configure le logger. Cela nous permettra de suivre les logs ;
- lignes 131-133 : on instancie la couche **[dao]** qui va être testée ;
- lignes 135-137 : on lance les tests ;

Les résultats console sont les suivants :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/http-clients/07/tests/
    Test2HttpClientDaoWithSession.py
2.  tests en cours...
3.  test_authenticate_user_failed
4.  ..MyException[35, ["Echec de l'authentification"]]
5.  test_authenticate_user_success
6.  test_delete_simulation
7.  MyException[35, ["la simulation n° [100] n'existe pas"]]
8.  test_get_simulations
9.  test_init_session_json
10. test_init_session_xml
11. test_init_session_xxx
12. MyException[73, il n'y a pas de session valide en cours]
```

```
13. ----------------------------------------------------------------------
14. Ran 7 tests in 0.171s
15.
16. OK
17.
18. Process finished with exit code 0
```

32 Le mode HTML de la version 12

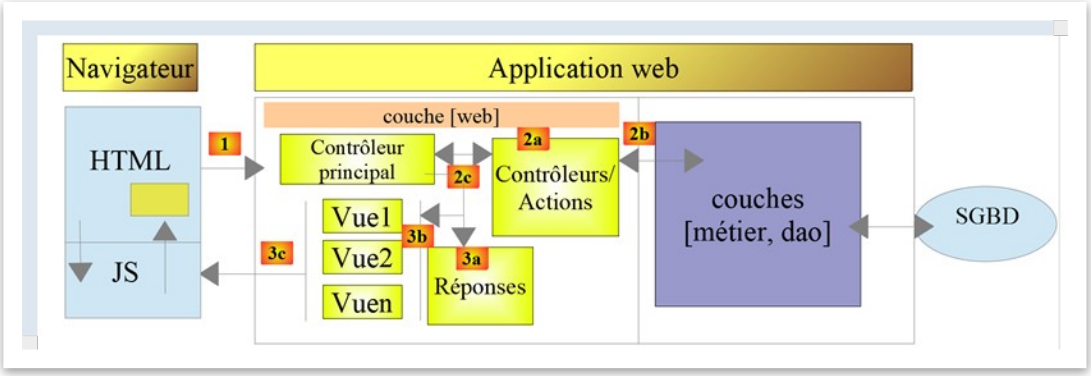
Nous avions indiqué au début de la version 12 que nous allions développer l'application en plusieurs temps. Nous avions écrit :

- à partir des vues de l'application HTML, nous allons définir les actions que doit implémenter l'application web. Nous allons ici utiliser les vues réelles mais ce pourrait être simplement des vues sur papier ;
- à partir de ces actions, nous allons définir les URL de service de l'application HTML ;
- nous allons implémenter ces URL de service avec un serveur délivrant du jSON. Cela permet de définir l'ossature du serveur web sans se préoccuper des pages HTML à délivrer. Nous testerons ces URL de service avec Postman ;
- nous testerons ensuite notre serveur jSON avec un client console ;
- une fois que le serveur jSON aura été validé, nous passerons à l'ériture de l'application HTML ;

Nous avons un serveur jSON et XML opérationnel. On peut maintenant passer au serveur HTML. Nous allons voir que celui-ci reprend toute l'architecture développée pour le serveur jSON / XML et leur ajoute une gestion de vues HTML.

32.1 Architecture MVC

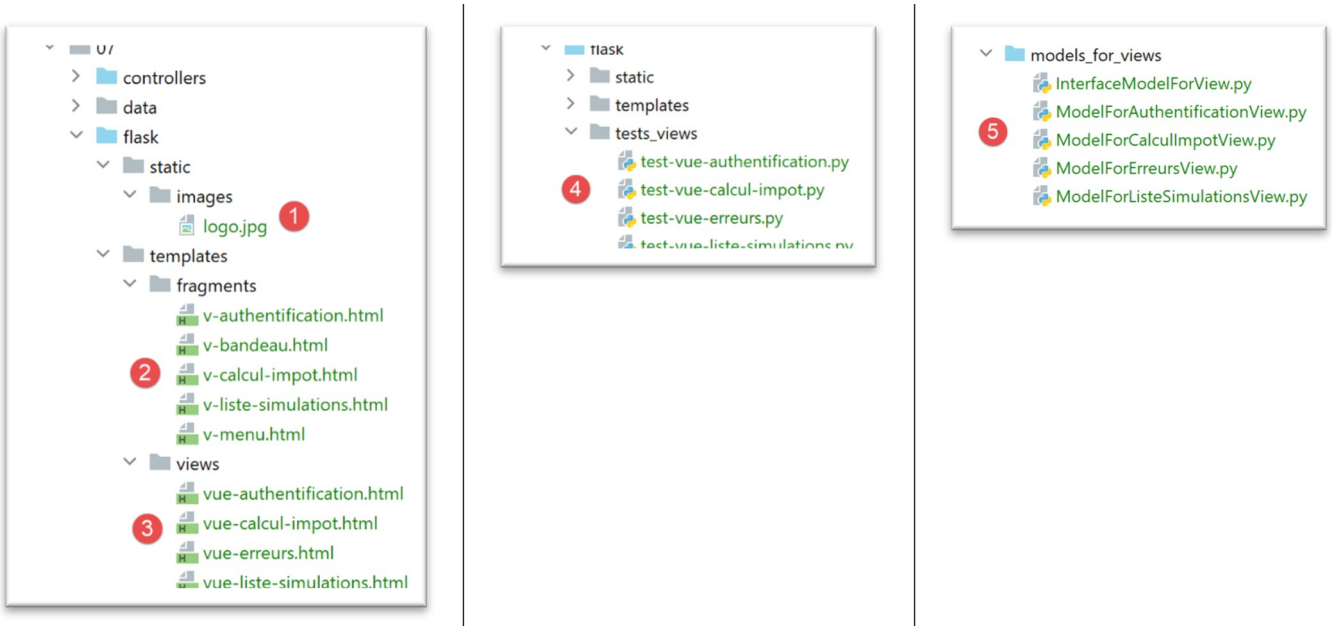
Nous allons implémenter le modèle d'architecture dit MVC (Modèle – Vue – Contrôleur) de la façon suivante :



Le traitement d'une demande d'un client se déroulera de la façon suivante :

- 1 - demande
  Les URL demandées seront de la forme *http://machine:port/action/param1/param2/…* Le **[Contrôleur principal]** utilisera un fichier de configuration pour " router " la demande vers le bon contrôleur. Pour cela, il utilisera le champ **[action]** de l'URL. Le reste de l'URL **[param1/param2/…]** est formé de paramètres facultatifs qui seront transmis à l'action. Le **C** de MVC est ici la chaîne **[Contrôleur principal, Contrôleur / Action]**. Si aucun contrôleur ne peut traiter l'action demandée, le serveur web répondra que l'URL demandée n'a pas été trouvée.
- 2 - traitement
  - l'action choisie **[2a]** peut exploiter les paramètres *parami* que le **[Contrôleur principal]** lui a transmis. Ceux-ci pourront provenir de deux sources :
    - du chemin **[/param1/param2/…]** de l'URL,
    - de paramètres postés dans le corps de la requête du client ;
  - dans le traitement de la demande de l'utilisateur, l'action peut avoir besoin de la couche **[métier] [2b]**. Une fois la demande du client traitée, celle-ci peut appeler diverses réponses. Un exemple classique est :
    - une réponse d'erreur si la demande n'a pu être traitée correctement ;
    - une réponse de confirmation sinon ;
  - le **[Contrôleur / Action]** rendra sa réponse **[2c]** au contrôleur principal ainsi qu'un code d'état. Ces codes d'état représenteront de façon unique l'état dans lequel se trouve l'application. Ce sera soit un code de réussite, soit un code d'erreur ;
- 3 - réponse
  - selon que le client a demandé une réponse jSON, XML ou HTML, le **[Contrôleur principal]** instanciera **[3a]** le type de réponse appropriée et demandera à celle-ci d'envoyer la réponse au client. Le **[Contrôleur principal]** lui transmettra et la réponse et le code d'état fournis par le **[Contrôleur / Action]** qui a été exécuté ;
  - si la réponse souhaitée est de type jSON ou XML, la réponse sélectionnée mettra en forme la réponse du **[Contrôleur / Action]** qu'on lui a donnée et l'enverra **[3c]**. Le client capable d'exploiter cette réponse peut être un script console Python ou un script Javascript logé dans une page HTML ;
  - si la réponse souhaitée est de type HTML, la réponse sélectionnée sélectionnera **[3b]** une des **vues** HTML **[Vuei]** à l'aide du code d'état qu'on lui a donné. C'est le **V** de MVC. A un code d'état correspond une unique vue. Cette vue **V** va afficher la réponse du **[Contrôleur / Action]** qui a été exécuté. Elle habille avec du HTML, CSS, Javascript les données de cette réponse. On appelle ces données le **modèle de la vue**. C'est le **M** de MVC. Le client est alors le plus souvent un navigateur ;

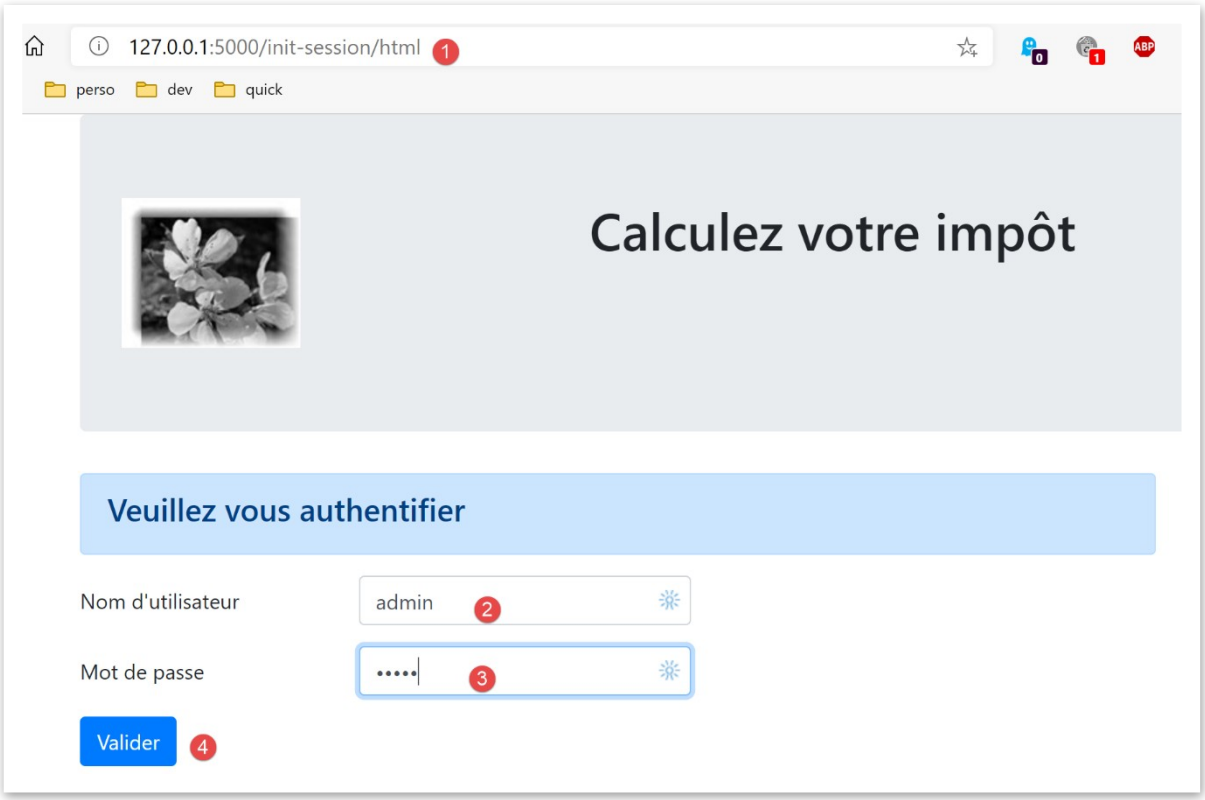
32.2 L'arborescence des scripts du serveur HTML



- en **[1]**, les éléments statiques du serveur HTML ;
- en **[2-3]**, les vues V du serveur HTML. Les fragments **[2]** sont des éléments réutilisables dans les vues **[3]** ;
- en **[4]**, un dossier qui servira aux tests des vues de façon statique ;
- en **[5]**, le dossier des modèles M des vues V, le M de MVC ;

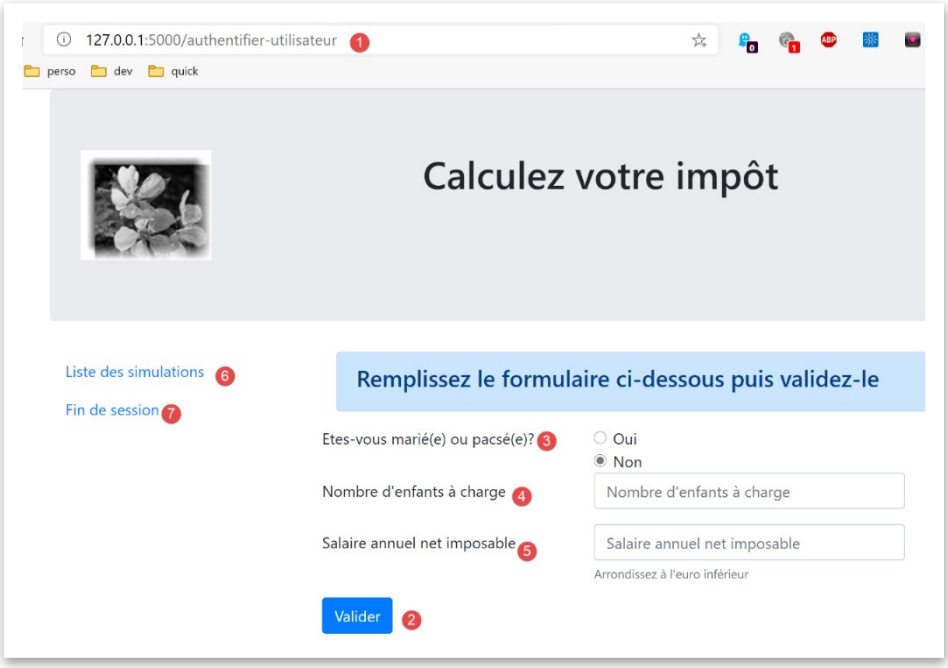
32.3 Présentation des vues

L'application web HTML utilise quatre vues. La 1ère vue est la vue d'authentification :



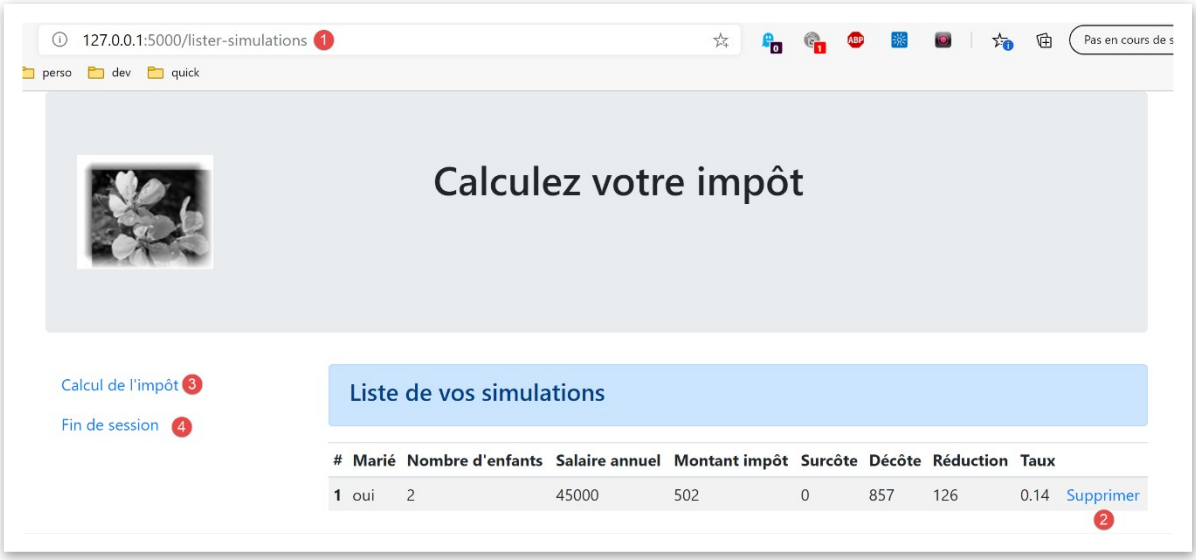
- l'action qui mène à cette 1ère vue est l'action **[/init-session] [1]** ;
- le clic sur le bouton **[Valider]** déclenche l'action **[/authentifier-utilisateur]** avec deux paramètres postés **[2-3]** ;

La vue du calcul de l'impôt :



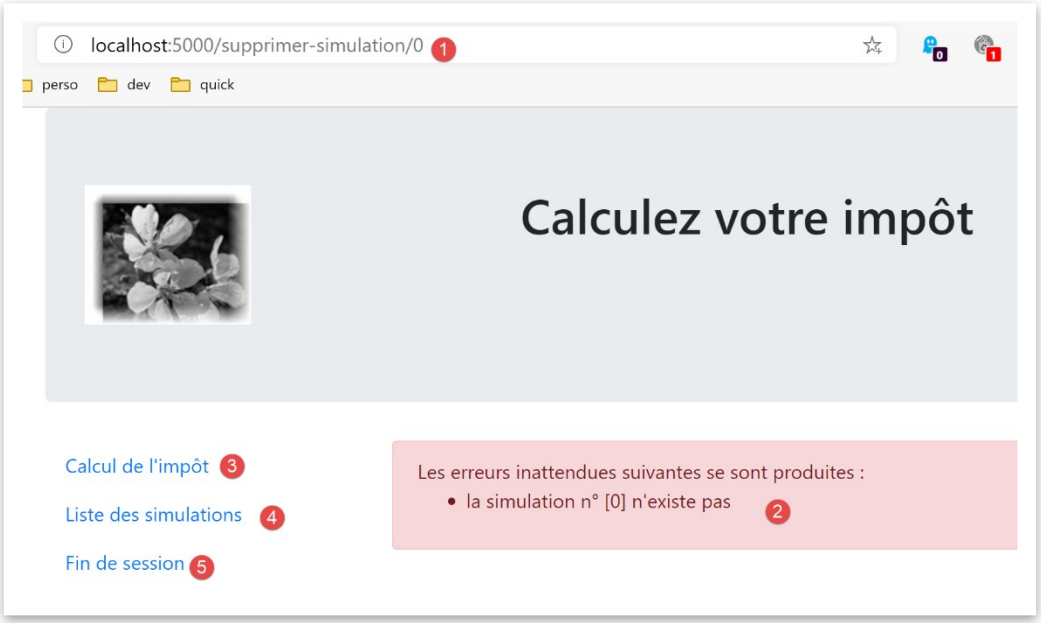
- en **[1]**, l'action **[/authentifier-utilisateur]** qui amène cette vue ;
- en **[2]**, le clic sur le bouton **[Valider]** déclenche l'exécution de l'action **[/calculer-impot]** avec trois paramètres postés **[2-5]** ;
- le clic sur le lien **[6]** déclenche l'action **[/lister-simulations]** sans paramètres ;
- le clic sur le lien **[7]** déclenche l'action **[/fin-session]** sans paramètres ;

La 3ième vue est celle des simulations faites par l'utilisateur authentifié :



- en **[1]**, l'action **[/lister-simulations]** qui amène cette vue ;
- en **[2]**, un clic sur le lien **[Supprimer]** déclenche l'action **[/supprimer-simulation]** avec un paramètre, le n° de la simulation à supprimer dans la liste ;
- un clic sur le lien **[3]** déclenche l'action **[/afficher-calcul-impot]** sans paramètres qui réaffiche la vue du calcul de l'impôt ;
- un clic sur le lien **[4]** déclenche l'action **[/fin-session]** sans paramètres ;

La 4ième vue sera appelée la vue des erreurs inattendues :



- en **[1]** : l'utilisateur a tapé lui-même l'URL. Or dans cet exemple il n'y avait pas de simulations. On reçoit donc le message d'erreur **[2]**. On connaît ce message. On l'avait en jSON / XML. On appellera ce type d'erreur, erreur inattendue, car elle ne peut pas se produire en utilisation normale de l'application. C'est lorsque l'utilisateur tape lui-même les URL qu'elles peuvent de produire ;
- en cas d'erreur inattendue, les liens **[3-5]** permettent de revenir à l'une des trois autres vues ;

Rappelons les différentes URL de service du serveur jSON / XML :

Action	Rôle	Contexte d'exécution
/init-session	Sert à fixer le type (json, xml, html) des réponses souhaitées	Requête **GET** Peut être émise à tout moment
/authentifier-utilisateur	Autorise ou non un utilisateur à se connecter	Requête **POST.** La requête doit avoir deux paramètres postés **[user, password]** Ne peut être émise que si le type de la session (json, xml, html) est connu
/calculer-impot	Fait une simulation de calcul d'impôt	Requête **POST.** La requête doit avoir trois paramètres postés **[marié, enfants, salaire]** Ne peut être émise que si le type de la session (json, xml, html) est connu et l'utilisateur authentifié
/lister-simulations	Demande à voir la liste des simulations opérées depuis le début de la session	Requête **GET.** Ne peut être émise que si le type de la session (json, xml, html) est connu et l'utilisateur authentifié
/supprimer-simulation/numéro	Supprime une simulation de la liste des simulations	Requête **GET.** Ne peut être émise que si le type de la session (json, xml, html) est connu et l'utilisateur authentifié
/afficher-calcul-impot	Affiche la page HTML du calul de l'impôt	Requête **GET**. Ne peut être émise que si le type de la session (json, xml, html) est connu et l'utilisateur authentifié
/fin-session	Termine la session de simulations.	Techniquement l'ancienne session web est supprimée et une nouvelle session est créée Ne peut être émise que si le type de la session (json, xml, html) est connu et l'utilisateur authentifié

Ces différentes URL de service seront également utilisées pour le serveur HTML.

32.4 Configuration des vues

Une action est traitée par un contrôleur. Ce contrôleur rend un tuple (résultat, status_code) où :

- **[résultat]** est un dictionnaire de clés **[action, état, réponse]** ;
- **[status_code]** est le code de statut de la réponse HTTP qui sera faite au client ;

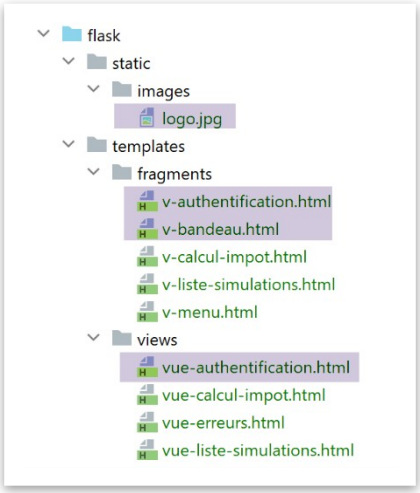
Dans une session HTML, la page affichée suite à une action, dépend du code d'état rendu par le cntrôleur. Cette dépendance est matérialisée dans la configuration **[config]** de la façon suivante :

```
1.          # les vues HTML et leurs modèles dépendent de l'état rendu par le contrôleur
2.          "views": [
3.              {
4.                  # vue d'authentification
5.                  "états": [
6.                      # /init-session réussite
7.                      700,
8.                      # /authentifier-utilisateur échec
9.                      201
10.                 ],
11.                 "view_name": "views/vue-authentification.html",
12.                 "model_for_view": ModelForAuthentificationView()
13.             },
14.             {
15.                 # vue du calcul de l'impôt
16.                 "états": [
17.                     # /authentifier-utilisateur réussite
18.                     200,
19.                     # /calculer-impot réussite
20.                     300,
21.                     # /calculer-impot échec
22.                     301,
23.                     # /afficher-calcul-impot
24.                     800
25.                 ],
26.                 "view_name": "views/vue-calcul-impot.html",
27.                 "model_for_view": ModelForCalculImpotView()
28.             },
29.             {
30.                 # vue de la liste des simulations
31.                 "états": [
32.                     # /lister-simulations
33.                     500,
34.                     # /supprimer-simulation
35.                     600
36.                 ],
37.                 "view_name": "views/vue-liste-simulations.html",
38.                 "model_for_view": ModelForListeSimulationsView()
39.             }
40.         ],
41.
42.         # vue des erreurs inattendues
43.         "view-erreurs": {
44.             "view_name": "views/vue-erreurs.html",
45.             "model_for_view": ModelForErreursView()
46.         },
47.
48.         # redirections
49.         "redirections": [
50.             {
51.                 "états": [
52.                     400,  # /fin-session réussite
53.                 ],
54.                 # redirection vers
55.                 "to": "/init-session/html",
56.             }
57.         ],
58.     }
```

- ligne 2-40 : **[views]** est une liste de vues. Raisonnons sur la vue des lignes 3-13 :
  - ligne 11 : la vue V affichée ;
  - ligne 12 : l'instance de classe chargée de générer le modèle M de cette vue ;
  - lignes 5-10 : les états qui mènent à cette vue ;
- lignes 3-13 : la vue d'authentification ;
- lignes 14-28 : la vue du calcul de l'impôt ;
- lignes 29-39 : la vue de la liste des simulations ;
- lignes 42-46 : la vue des erreurs inattendues ;
- lignes 49-57 : certains états mènent à une vue via une redirection. C'est le cas de l'état 400 qui est celui de l'action **[/fin-session]** réussie. Il faut alors rediriger le client vers l'action **[http://machine:port/chemin/init-session/html]** ;

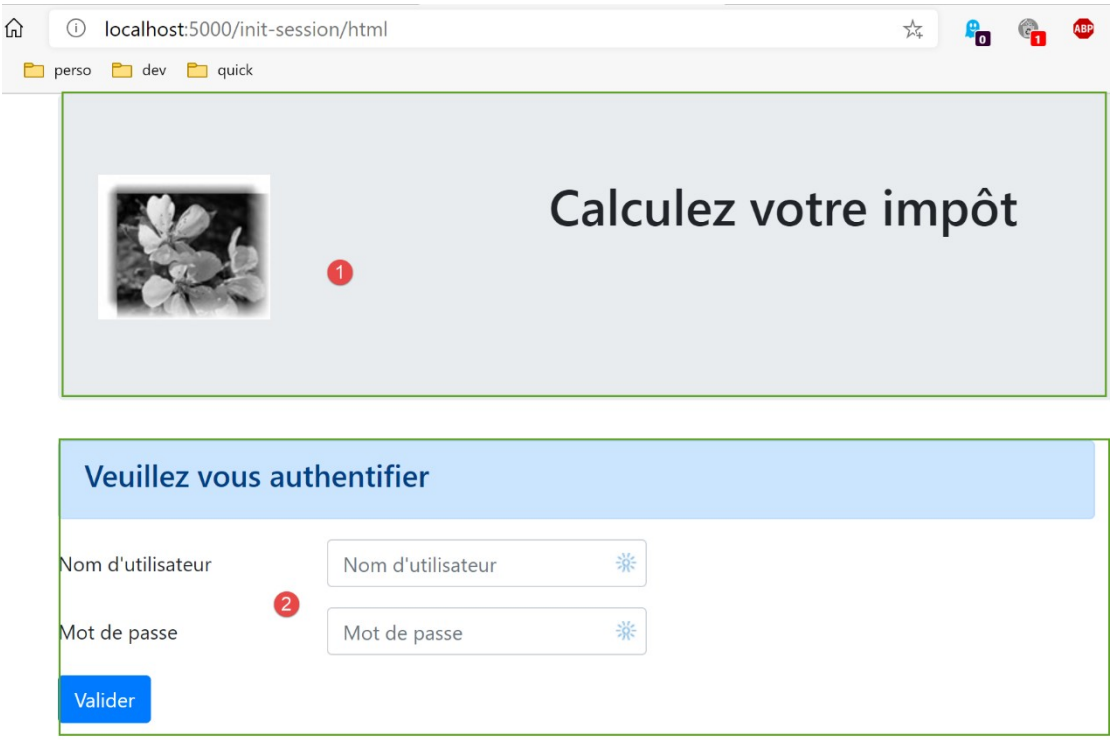
Nous présentons maintenant les différentes vues.

32.5 La vue d'authentification



32.5.1 Présentation de la vue

La vue d'authentification est la suivante :



La vue est composée de deux éléments qu'on appellera des fragments :

- le fragment **[1]** est généré par le fragment **[v-bandeau.html]** ;
- le fragment **[2]** est généré par le fragment **[v-authentification.html]** ;

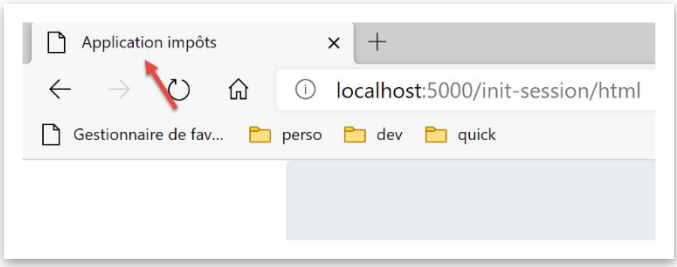
La vue d'authentification est générée par la page **[vue-authentification.html]** suivante :

```
1.  <!-- document HTML -->
2.  <!doctype html>
3.  <html lang="fr">
4.  <head>
5.      <!-- Required meta tags -->
6.      <meta charset="utf-8">
```

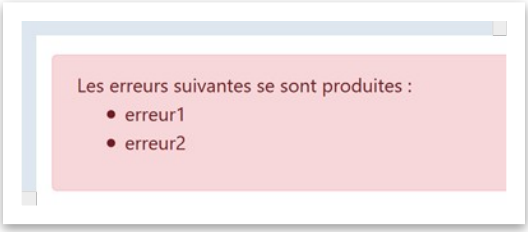
```
7.      <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
8.      <!-- Bootstrap CSS -->
9.      <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
10.     <title>Application impôts</title>
11. </head>
12.
13. <body>
14. <div class="container">
15.     <!-- bandeau -->
16.     {% include "fragments/v-bandeau.html" %}
17.     <!-- ligne à deux colonnes -->
18.     <div class="row">
19.         <div class="col-md-9">
20.             {% include "fragments/v-authentification.html" %}
21.         </div>
22.     </div>
23.     <!-- si erreur - on affiche une alerte d'erreur -->
24.     {% if modèle.error %}
25.     <div class="row">
26.         <div class="col-md-9">
27.             <div class="alert alert-danger" role="alert">
28.                 Les erreurs suivantes se sont produites :
29.                 <ul>{{modèle.erreurs|safe}}</ul>
30.             </div>
31.         </div>
32.     </div>
33.     {% endif %}
34. </div>
35. </body>
36. </html>
```

Commentaires

- ligne 2 : un document HTML commence avec cette ligne ;
- lignes 3-36 : la page HTML est encapsulée dans les balises <**html**> </**html**> ;
- lignes 4-11 : entête (**head**) du document HTML ;
- ligne 6 : la balise <**meta charset**> indique ici que le document est codé en UTF-8 ;
- ligne 7 : la balise <**meta name='viewport'**> fixe l'affichage initial de la vue : sur toute la largeur de l'écran qui l'affiche (width) à sa taille initiale (**initial-scale**) sans redimensionnement pour s'adapter à une taille plus petite d'écran (**shrink-to-fit**) ;
- ligne 9 : la balise <**link rel='stylesheet'**> fixe le fichier CSS qui gouverne l'apparence de la vue. Nous utilisons ici le framework CSS Bootstrap 4.4.1 **[https://getbootstrap.com/docs/4.0/getting-started/introduction/]** ;
- ligne 10 : la balise <**title**> fixe le titre de la page :



- lignes 13-35 : le corps de la page web est encapsulé dans les balises **<body></body>** ;
- lignes 14-34 : la balise délimite une section de la page affichée. Les attributs **[class]** utilisés dans la vue se réfèrent tous au framework CSS Bootstrap. La balise <**div class='container'**> (ligne 14) délimite un conteneur Bootstrap ;
- ligne 26 : on inclut le fragment **[v-bandeau.html]**. Ce fragment génère le bandeau **[1]** de la page. Nous le décrirons bientôt ;
- lignes 18-22 : la balise <**div class='row'**> délimite une ligne Bootstrap. Ces lignes sont constituées de 12 colonnes ;
- ligne 19 : la balise <**div class='col-md-9'**> délimite une section de 9 colonnes ;
- ligne 20 : on inclut le fragment **[v-authentification.html]** qui affiche le formulaire d'authentification **[2]** de la page. Nous le décrirons bientôt ;
- lignes 24-33 : le code HTML de ces lignes n'est utilisé que si **[modèle.error]** vaut True. Nous procèderons toujours ainsi : le modèle d'une vue HTML sera encapsulé dans un dictionnaire **[modèle]** ;
- lignes 24-33 : l'authentification échoue si l'utilisateur entre des identifiants incorrects. Dans ce cas, la vue d'authentification est réaffichée avec un message d'erreur. L'attribut **[modèle.error]** indique s'il faut afficher ce message d'erreur ;
- lignes 27-30 : délimitent une zone à fond rose (**class="alert alert-danger"**) (ligne 27) ;



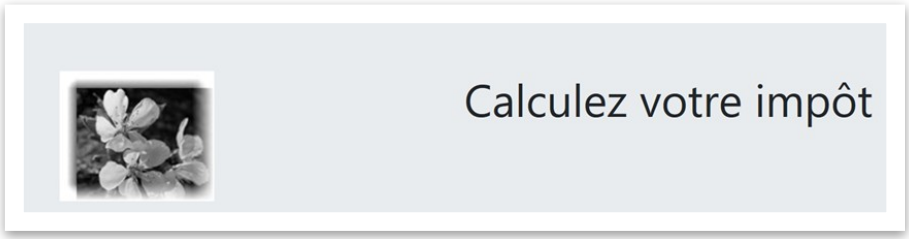
- ligne 28 : un texte ;
- ligne 29 : la balise HTML <**ul**> (unordered list) affiche une liste à puces. Chaque élément de la liste doit avoir la syntaxe ** élément **. On affiche ici la valeur de **[modèle.erreurs]**. Cette valeur est filtrée (présence de |) par le filtre **[safe]**. Par défaut, lorsqu'une chaîne de caractères doit être envoyée au navigateur, Flask 'neutralise' toutes les balises HTML qui pourraient s'y trouver afin que le navigateur ne les interprète pas. Mais parfois on veut les interpréter. Ce sera ici le cas où la chaîne **[modèle.erreurs]** contiendra des balises HTML <li> et qui servent à délimiter un élément de la liste. Dans ce cas, on utilise le filtre **[safe]** qui dit à Flask que la chaîne à afficher est sûre (safe) et que donc il ne doit pas neutraliser les balises HTML qu'il y trouvera ;

Retenons de ce code les éléments dynamiques à définir :

- **[modèle.error]** : pour afficher un message d'erreur ;
- **[modèle.erreurs]** : une liste (au sens HTML du terme) de messages d'erreur ;

32.5.2 Le fragment [v-bandeau.html]

Le fragment **[v-bandeau.html]** affiche le bandeau supérieur de toutes les vues de l'application web :



Le code du fragment **[v-bandeau.html]** est le suivant :

```
1.  <!-- Bootstrap Jumbotron -->
2.  <div class="jumbotron">
3.      <div class="row">
4.          <div class="col-md-4">
5.              <img src="{{ url_for('static', filename='images/logo.jpg') }}" alt="Cerisier en fleurs"/>
6.          </div>
7.          <div class="col-md-8">
8.              <h1>
9.                  Calculez votre impôt
10.             </h1>
11.         </div>
12.     </div>
13. </div>
```

Commentaires

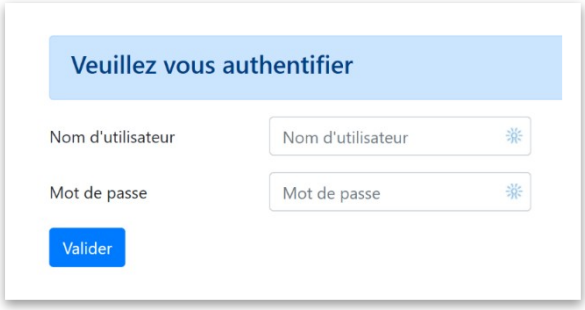
- lignes 2-13 : le bandeau est encapsulé dans une section Bootstrap de type Jumbotron **[ ]**. Cette classe Bootstrap stylise d'une façon particulière le contenu affiché pour le faire ressortir ;
- lignes 3-12 : une ligne Bootstrap ;
- lignes 4-6 : une image **[img]** est placée dans les quatre premières colonnes de la ligne ;
- ligne 5 : la syntaxe :

```
{{ url_for('static', filename='images/logo.jpg') }}
```

  utilise la fonction **[url_for]** de Flask. Ici, sa valeur sera l'URL du fichier **[images/logo.pg]** du dossier **[static]** ;
- lignes 7-11 : les 8 autres colonnes de la ligne (on rappelle qu'il y en a 12 au total) serviront à placer un texte (ligne 9) en gros caractères (<h1>, lignes 8-10) ;

32.5.3 Le fragment `[v-authentification.html]`

Le fragment **`[v-authentification.html]`** affiche le formulaire d'authentification de l'application web :

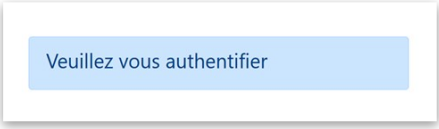


Le code du fragment **`[v-authentification.html]`** est le suivant :

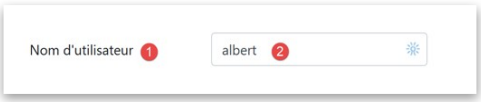
```
1.  <!-- formulaire HTML - on poste ses valeurs avec l'action [authentifier-utilisateur] -->
2.  <form method="post" action="/authentifier-utilisateur">
3.
4.      <!-- titre -->
5.      <div class="alert alert-primary" role="alert">
6.          <h4>Veuillez vous authentifier</h4>
7.      </div>
8.
9.      <!-- formulaire Bootstrap -->
10.     <fieldset class="form-group">
11.         <!-- 1ère ligne -->
12.         <div class="form-group row">
13.             <!-- libellé -->
14.             <label for="user" class="col-md-3 col-form-label">Nom d'utilisateur</label>
15.             <div class="col-md-4">
16.                 <!-- zone de saisie texte -->
17.                 <input type="text" class="form-control" id="user" name="user"
18.                        placeholder="Nom d'utilisateur" value="{{ modèle.login }}" required>
19.             </div>
20.         </div>
21.         <!-- 2ième ligne -->
22.         <div class="form-group row">
23.             <!-- libellé -->
24.             <label for="password" class="col-md-3 col-form-label">Mot de passe</label>
25.             <!-- zone de saisie texte -->
26.             <div class="col-md-4">
27.                 <input type="password" class="form-control" id="password" name="password"
28.                        placeholder="Mot de passe" required>
29.             </div>
30.         </div>
31.         <!-- bouton de type [submit] sur une 3ième ligne -->
32.         <div class="form-group row">
33.             <div class="col-md-2">
34.                 <button type="submit" class="btn btn-primary">Valider</button>
35.             </div>
36.         </div>
37.     </fieldset>
38.
39. </form>
```

Commentaires

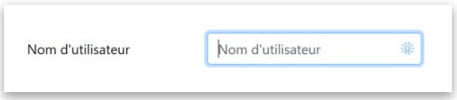
- lignes 2-39 : la balise <form> délimite un formulaire HTML. Celui-ci a en général les caractéristiques suivantes :
  - il définit des zones de saisies (balises <**input**> des lignes 17 et 27 ;
  - il a un bouton de type **`[submit]`** (ligne 34) qui envoie les valeurs saisies à l'URL indiquée dans l'attribut **`[action]`** de la balise **`[form]`** (ligne 2). La méthode HTTP utilisée pour requêter cette URL est précisée dans l'attribut **`[method]`** de la balise **`[form]`** (ligne 2) ;
  - ici, lorsque l'utilisateur va cliquer sur le bouton **`[Valider]`** (ligne 34), le navigateur va poster (ligne 2) les valeurs saisies dans le formulaire à l'URL **`[/authentifier-utilisateur]`** (ligne 2) ;
  - les valeurs postées sont les valeurs saisies par l'utilisateur dans les zones de saisie des lignes 17 et 27. Elles seront postées dans le corps de la requête HTTP que fera le navigateur sous la forme **`[x-www-forl-urlencoded]`**. Les noms des paramètres **`[user, password]`** sont ceux des attributs **`[name]`** des zones de saisie des lignes 17 et 27 ;
- ligne 5-7 : une section Bootstrap pour afficher un titre dans un fond bleu :



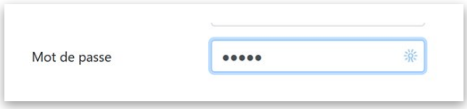
- lignes 10-37 : un formulaire Bootstrap. Tous les éléments du formulaire vont alors être stylisés d'une certaine façon ;
- lignes 12-20 : définissent la 1re ligne Bootstrap du formulaire :



- la ligne 14 définit le libellé **[1]** sur trois colonnes. L'attribut **[for]** de la balise **[label]** relie le libellé à l'attribut **[id]** de la zone de saisie de la ligne 17 ;
- lignes 15-19 : met la zone de saisie dans un ensemble de quatre colonnes ;
- lignes 17-18 : la balise HTML **[input]** décrit une zone de saisie. Elle a plusieurs paramètres :
  - **[type='text']** : c'est une zone de saisie texte. On peut y taper n'importe quoi ;
  - **[class='form-control']** : style Bootstrap pour la zone de saisie ;
  - **[id='user']** : identifiant de la zone de saisie. Cet identifiant est en général utilisé par le CSS et le code Javascript ;
  - **[name='user']** : nom de la zone de saisie. C'est sous ce nom que la valeur saisie par l'utilisateur sera postée par le navigateur **[user=xx]** ;
  - **[placeholder='invite']** : le texte affiché dans la zone de saisie lorsque l'utilisateur n'a encore rien tapé ;



  - **[value='valeur']** : le texte 'valeur' sera affiché dans la zone de saisie dès que celle-ci sera affichée, avant donc que l'utilisateur ne saisisse autre chose. Ce mécanisme est utilisé en cas d'erreur pour afficher la saisie qui a provoqué l'erreur. Ici cette valeur sera la valeur de la variable **[modèle.login]** ;
  - **[required]** : exige que l'utilisateur mette une valeur pour que le formulaire puisse être envoyé au serveur :
- lignes 21-30 : un code analogue pour la saisie du mot de passe ;
- ligne 27 : **[type='password']** fait qu'on a une zone de saisie texte (on peut taper n'importe quoi) mais les caractères tapés sont cachés :



- lignes 32-36 : une 3e ligne Bootstrap pour le bouton **[Valider]** ;



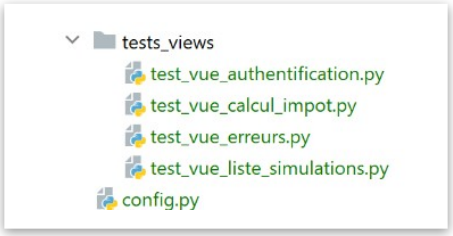
- ligne 34 : parce qu'il a l'attribut **[type=submit]**, un clic sur ce bouton déclenche l'envoi au serveur par le navigateur des valeurs saisies comme il a été expliqué précédemment. L'attribut CSS **[class="btn btn-primary"]** affiche un bouton bleu :

Il nous reste à expliquer une dernière chose. Ligne 2, l'attribut **[action="/authentifier-utilisateur"]** définit une URL incomplète (elle ne commence pas par http://machine:port/chemin). Dans notre exemple, toutes les URL de l'application sont de la forme **[http://machine:port/chemin/action/param1/param2/..]** où **[http://machine:port/chemin]** est la **racine** des URL de service. Dans **[action="/authentifier-utilisateur"]** nous avons une URL **absolue** ç-à-d mesurée à partir de la racine des URL. L'URL complète du POST est donc **[http://machine:port/chemin/authentifier-utilisateur]** et c'est ce qu'utilisera le navigateur.

On retiendra que ce fragment utilise le modèle **[modèle.login]**.

32.5.4 Tests visuels

On peut procéder aux tests des vues bien avant leur intégration dans l'application. Il s'agit ici de tester leur aspect visuel. Nous



rassemblerons toutes les vues de test dans le dossier **[tests_views]** du projet :

Pour tester la vue V **[vue-authentification.html]**, il nous faut créer le modèle M de données qu'elle va afficher. On le fait avec le script **[test_vue_authentification.py]** :

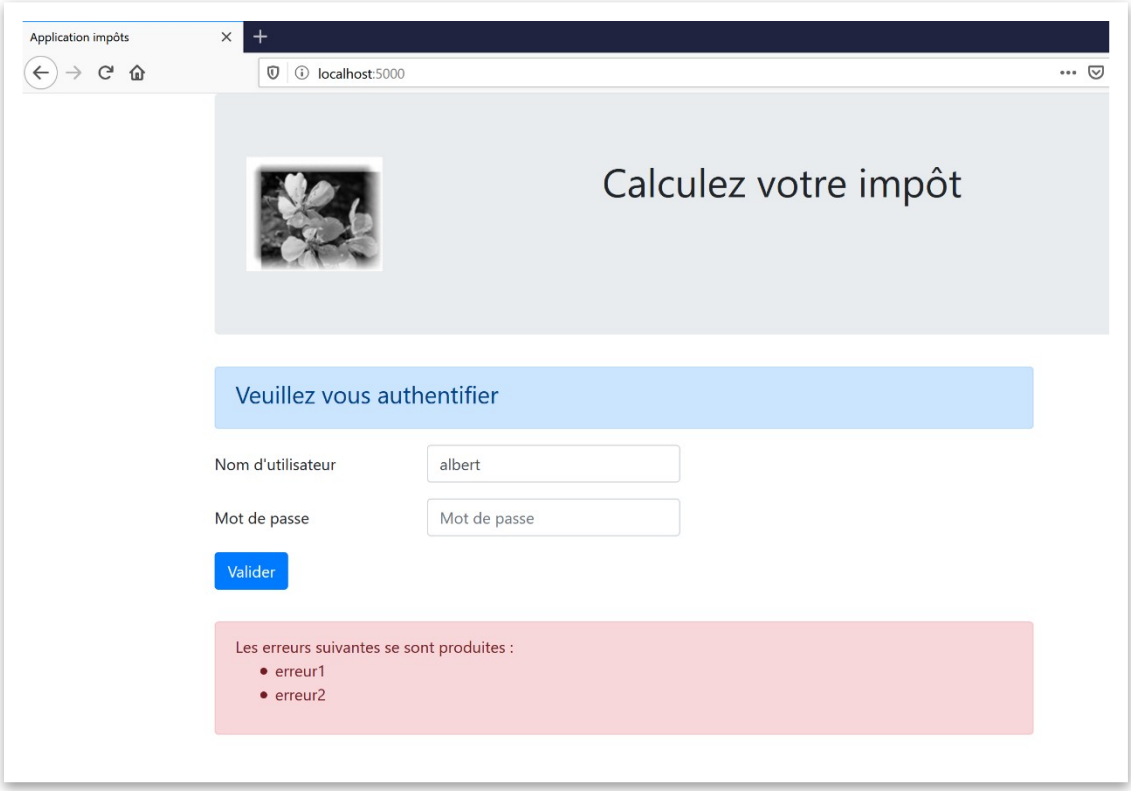
```
1.  from flask import Flask, render_template, make_response
2.
3.  # application Flask
4.  app = Flask(__name__, template_folder="../templates", static_folder="../static")
5.
6.  # Home URL
7.  @app.route('/')
8.  def index():
9.      # on encapsule les données de la pagé dans modèle
10.     modèle = {}
11.     # identifiant utilisateur
12.     modèle["login"] = "albert"
13.     # liste d'erreurs
14.     modèle["error"] = True
15.     erreurs = ["erreur1", "erreur2"]
16.     # on construit une liste HTML des erreurs
17.     content = ""
18.     for erreur in erreurs:
19.         content += f"<li>{erreur}</li>"
20.     modèle["erreurs"] = content
21.     # affichage de la page
22.     return make_response(render_template("views/vue-authentification.html", modèle=modèle))
23.
24. # main
25. if __name__ == '__main__':
26.     app.config.update(ENV="development", DEBUG=True)
27.     app.run()
1.
```

Commentaires

- lignes 1-3 : on crée une application Flask dont le seul but est d'afficher la vue **[vue-authentification.html]** (ligne 22) ;
- ligne 7 : l'application n'a qu'une unique URL de service ;
- lignes 9-20 : la vue d'authentification a des parties dynamiques contrôlées par l'objet **[modèle]**. On appelle cet objet le **modèle de la vue**. Selon l'une des deux définitions données pour le sigle MVC, on a là le M du MVC. Lors de la définition de la vue **[vue-authentification.html]**, nous avions identifié trois valeurs dynamiques :
  - **[modèle.error]** : booléen indiquant s'il faut afficher un message d'erreur ;
  - **[modèle.erreurs]** : une liste HTML de messages d'erreur ;
  - **[modèle.login]** : le login d'un utilisateur ;

  Il nous faut donc définir ces trois valeurs dynamiques.
- lignes 9-20 : on définit les trois éléments dynamiques de la vue d'authentification ;

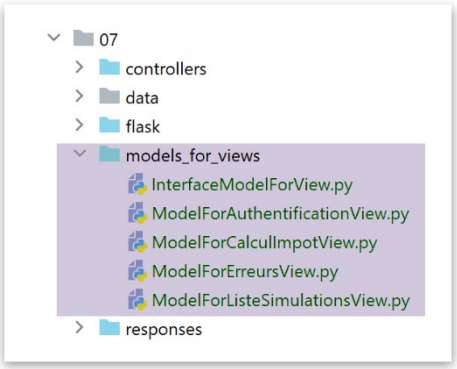
Pour faire le test, on lance le script **[tests_views/test_vue_authentification.py]** et on demande l'URL **[/localhost:5000/]** :



On poursuit ces tests visuels jusqu'à être satisfait du résultat.

32.5.5 Calcul du modèle de la vue

Une fois l'aspect visuel de la vue déterminé, on peut procéder au calcul du modèle de la vue en conditions réelles. Les modèles des vues seront générées par des classes rassemblées dans le dossier **[models_for_views]** :



Chaque classe générant un modèle de vue respectera l'interface **[InterfaceModelForView]** suivante :

```
1.  from abc import ABC, abstractmethod
2.
3.  from flask import Request
4.  from werkzeug.local import LocalProxy
5.
6.  class InterfaceModelForView(ABC):
7.
8.      @abstractmethod
9.      def get_model_for_view(self, request: Request, session: LocalProxy, config: dict, résultat: dict) -> dict:
10.         pass
```

- lignes 8-10 : la méthode **[get_model_for_view]** est chargée de produire un modèle de vue encapsulé dans un dictionnaire. Elle reçoit pour cela les informations suivantes :
  - **[request, session, config]** sont les mêmes paramètres utilisés par le contrôleur de l'action. Ils sont donc également transmis au modèle ;

- le contrôleur a produit un résultat **[résultat]** qui est également transmis au modèle. Ce résultat contient un élément important **[état]** qui indique comme s'est passée l'exécution de l'action en cours. Le modèle va utiliser cette information ;

Nous avons vu que dans la configuration **[config]** de l'application, les codes d'état rendus par les contrôleurs sont utilisés pour désigner la vue HTML à afficher :

```
1.          # les vues HTML et leurs modèles dépendent de l'état rendu par le contrôleur
2.          "views": [
3.              {
4.                  # vue d'authentification
5.                  "états": [
6.                      # /init-session réussite
7.                      700,
8.                      # /authentifier-utilisateur échec
9.                      201
10.                 ],
11.                 "view_name": "views/vue-authentification.html",
12.                 "model_for_view": ModelForAuthentificationView()
13.             },
14.             {
15.                 # vue du calcul de l'impôt
16.                 "états": [
17.                     # /authentifier-utilisateur réussite
18.                     200,
19.                     # /calculer-impot réussite
20.                     300,
21.                     # /calculer-impot échec
22.                     301,
23.                     # /afficher-calcul-impot
24.                     800
25.                 ],
26.                 "view_name": "views/vue-calcul-impot.html",
27.                 "model_for_view": ModelForCalculImpotView()
28.             },
29.             {
30.                 # vue de la liste des simulations
31.                 "états": [
32.                     # /lister-simulations
33.                     500,
34.                     # /supprimer-simulation
35.                     600
36.                 ],
37.                 "view_name": "views/vue-liste-simulations.html",
38.                 "model_for_view": ModelForListeSimulationsView()
39.             }
40.         ],
41.         # vue des erreurs inattendues
42.         "view-erreurs": {
43.             "view_name": "views/vue-erreurs.html",
44.             "model_for_view": ModelForErreursView()
45.         },
46.         # redirections
47.         "redirections": [
48.             {
49.                 "états": [
50.                     400,  # /fin-session réussite
51.                 ],
52.                 # redirection vers
53.                 "to": "/init-session/html",
54.             }
55.         ],
56.     }
```

Ce sont donc les codes d'état **[700, 201]** (lignes 7 et 9) qui font afficher la vue d'authentification. Pour retrouver la signification de ces codes, on peut s'aider des tests **[Postman]** réalisés sur l'application jSON :

- **[init-session-json-700]** : 700 est le code d'état à l'issue d'une action **[init-session]** réussie : on présente alors le formulaire d'authentification vide ;
- **[authentifier-utilisateur-201]** : 201 est le code d'état à l'issue d'une action **[authentifier-utilisateur]** ayant échoué (identifiants non reconnus) : on présente alors le formulaire d'authentification pour qu'il soit corrigé ;

Maintenant que nous savons à quels moments doit être affiché le formulaire d'authentification, on peut calculer son modèle dans **[ModelForAuthentificationView]** (ligne 12) :

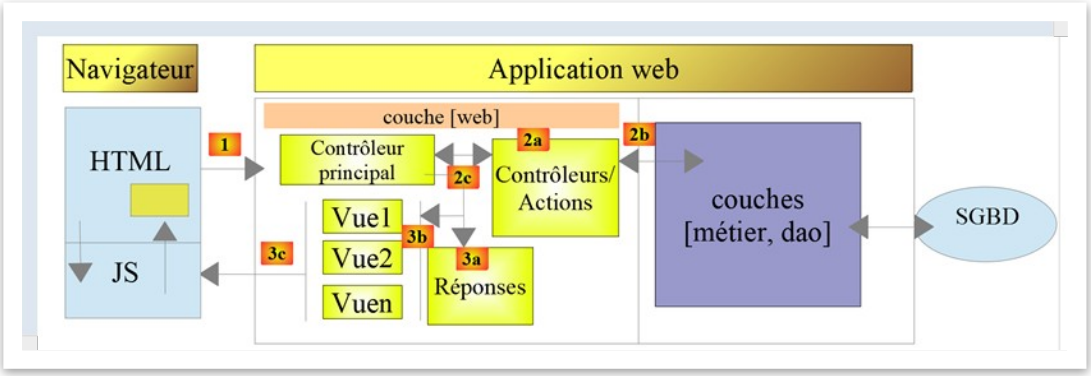
```
1.  from flask import Request
2.  from werkzeug.local import LocalProxy
3.
4.  from InterfaceModelForView import InterfaceModelForView
5.
6.  class ModelForAuthentificationView(InterfaceModelForView):
7.
8.      def get_model_for_view(self, request: Request, session: LocalProxy, config: dict, résultat: dict) -> dict:
9.          # on encapsule les données de la pagé dans modèle
10.         modèle = {}
11.         # état de l'application
12.         état = résultat["état"]
13.         # le modèle dépend de l'état
14.         if état == 700:
15.             # cas de l'affichage du formulaire vide
16.             modèle["login"] = ""
17.             # il n'y pas d'erreur à afficher
18.             modèle["error"] = False
19.         elif état == 201:
20.             # authentification erronée
21.             # on réaffiche l'utilisateur initialement saisi
22.             modèle["login"] = request.form.get("user")
23.             # il y a une erreur à afficher
24.             modèle["error"] = True
25.             # liste HTML de msg d'erreur
26.             erreurs = ""
27.             for erreur in résultat["réponse"]:
28.                 erreurs += f"<li>{erreur}</li>"
29.             modèle["erreurs"] = erreurs
30.
31.         # on rend le modèle
32.         return modèle
```

Commentaires

- ligne 8 : la méthode **[get_model_for_view]** de la vue d'authentification doit fournir un dictionnaire avec trois clés **[error, erreurs, login]**. Ce calcul se fait à partir du code d'état rendu par le contrôleur de l'action ;
- ligne 12 : on récupère le code d'état rendu par le contrôleur qui a traité l'action en cours ;
- lignes 14-29 : le modèle dépend de ce code d'état ;
- lignes 15-18 : cas où on doit afficher un formulaire d'authentification vierge ;
- lignes 20-29 : cas de l'authentification erronée : on raffiche l'identifiant saisi pour l'utilisateur et on affiche un message d'erreur. L'utilisateur au clavier peut alors réessayer une autre authentification ;
- ligne 22 : l'identifiant initialement saisi par l'utilisateur peut être retrouvé dans la requête du client ;
- ligne 24 : on signale qu'il y a des erreurs à afficher ;
- lignes 26-29 : en cas d'erreur, résultat**['réponse']** contient une liste d'erreurs ;

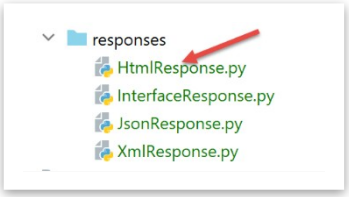
32.5.6 Génération des réponses HTML

Revenons au modèle MVC de l'application HTML :



- en 2 (2a, 2b) : le contrôleur exécute une action ;
- en 3 (3a, 3b, 3c) : une vue est choisie et envoyée au client ;

En **[3a]**, un type de réponse (jSON, XML, HTML) est choisi. Nous avons vu comment les réponses jSON et XML étaient générées mais pas encore les réponses HTML. Celles-ci sont générées par la classe **[HtmlResponse]** :



Rappelons comment dans le script principal **[main]** le type de réponse à faire à l'utilisateur est déterminé :

```
1.          ….
2.          # on construit la réponse à envoyer
3.          response_builder = config["responses"][type_response]
4.          response, status_code = response_builder \
5.              .build_http_response(request, session, config, status_code, résultat)
6.          # on envoie la réponse
7.          return response, status_code
```

où, ligne 3, config**['responses']** est le dictionnaire suivant :

```
1.          # les différents types de réponse (json, xml, html)
2.          "responses": {
3.              "json": JsonResponse(),
4.              "html": HtmlResponse(),
5.              "xml": XmlResponse()
6.          },
```

C'est donc la classe **[HtmlResponse]** qui génère la réponse HTML. Son code est le suivant :

```
1.  # dictionnaire des réponses HTML selon l'état contenu dans le résultat
2.
3.  from flask import make_response, render_template
4.  from flask.wrappers import Response
5.  from werkzeug.local import LocalProxy
6.
7.  from InterfaceResponse import InterfaceResponse
8.
9.  class HtmlResponse(InterfaceResponse):
10.
11.     def build_http_response(self, request: LocalProxy, session: LocalProxy, config: dict, status_code: int,
12.                             résultat: dict) -> (Response, int):
13.         # la réponse HTML dépend du code d'état rendu par le contrôleur
14.         état = résultat["état"]
15.
16.         # faut-il faire une redirection ?
17.         for redirection in config["redirections"]:
18.             # états nécessitant une redirection
19.             états = redirection["états"]
20.             if état in états:
21.                 # il faut faire une redirection
22.                 return redirect(f"/{redirection['to']}"), status.HTTP_302_FOUND
23.
24.         # à un état, correspond une vue
25.         # on cherche celle-ci dans la liste des vues
26.         views_configs = config["views"]
27.         trouvé = False
28.         i = 0
29.         # on parcourt la liste des vues
30.         nb_views = len(views_configs)
31.         while not trouvé and i < nb_views:
32.             # vue n° i
33.             view_config = views_configs[i]
34.             # états associés à la vue n° i
35.             états = view_config["états"]
36.             # est-ce que l'état cherché se trouve dans les états associés à la vue n° i
37.             if état in états:
38.                 trouvé = True
39.             else:
40.                 # vue suivante
41.                 i += 1
42.         # trouvé ?
43.         if not trouvé:
```

```
44.             # si aucune vue n'existe pour l'état actuel de l'application
45.             # on rend la vue d'erreurs
46.             view_config = config["view-erreurs"]
47.
48.         # on calcule le modèle de la vue à afficher
49.         model_for_view = view_config["model_for_view"]
50.         modèle = model_for_view.get_model_for_view(request, session, config, résultat)
51.         # on génère le code HTML de la réponse
52.         html = render_template(view_config["view_name"], modèle=modèle)
53.         # on construit la réponse HTTP
54.         response = make_response(html)
55.         response.headers['Content-Type'] = 'text/html; charset=utf-8'
56.         # on rend le résultat
57.         return response, status_code
```

- line 11 : la méthode **[build_http_response]** chargée de générer la réponse HTML reçoit les paramètres suivants :
  - **[request, session, dict]** : ce sont les paramètres utilisés par le contrôleur pour traiter l'action en cours ;
  - **[status_code, résultat]** sont les deux résultats produits par ce même contrôleur ;
- ligne 14 : comme nous l'avons dit, la réponse HTML du serveur dépend du code d'état contenu dans le dictionnaire **[résultat]** ;
- lignes 16-22 : on gère d'abord les redirections. Pour l'instant nous allons ignorer ce cas jusqu'au moment où on rencontrera un exemple de redirection. On notera que les redirections sont typiquement un cas d'usage du serveur HTML. On ne rencontre pas ce cas avec les serveurs jSON ouXML ;
- lignes 24-41 : on cherche parmi les vues celle dont la liste **[états]** contient l'état recherché ;
- lignes 42-46 : si aucune vue n'a été trouvée, il s'agit d'une erreur inattendue. Prenons un exemple. Dans le fonctionnement normal de l'application, l'action **[/supprimer-simulation]** ne doit jamais boguer. En effet, nous allons voir que cette suppression de simulations se fait à partir de liens générés par le code. Ces liens sont corrects et ne peuvent mener à une erreur. Cependant, nous l'avons vu, l'utilisateur peut taper directement l'URL **[/supprimer-simulation/id]** et provoquer ainsi une erreur. Dans ce cas, le contrôleur **[SupprimerSimulationController]** rend un code d'état 601. Or ce code d'état ne se trouve pas dans la liste des codes d'état menant à l'affichage d'une page HTML. Ce sera donc la vue d'erreur qui sera affichée. Elle est définie ainsi dans la configuration :

```
        # vue des erreurs inattendues
        "view-erreurs": {
            "view_name": "views/vue-erreurs.html",
            "model_for_view": ModelForErreursView()
        },
```

- ligne 49 : une fois qu'on connaît la vue à afficher, on récupère la classe générant son modèle. Elle aussi se trouve dans la configuration **[config]** ;
- ligne 50 : une fois cette classe trouvée, on génère le modèle de la vue ;
- ligne 52 : une fois le modèle M de la vue V calculé, on peut générer le code HTML de la vue ;
- lignes 54-55 : on construit la réponse HTTP de la réponse avec un corps HTML ;
- lignes 56-57 : on rend la réponse HTTP avec son code de statut ;

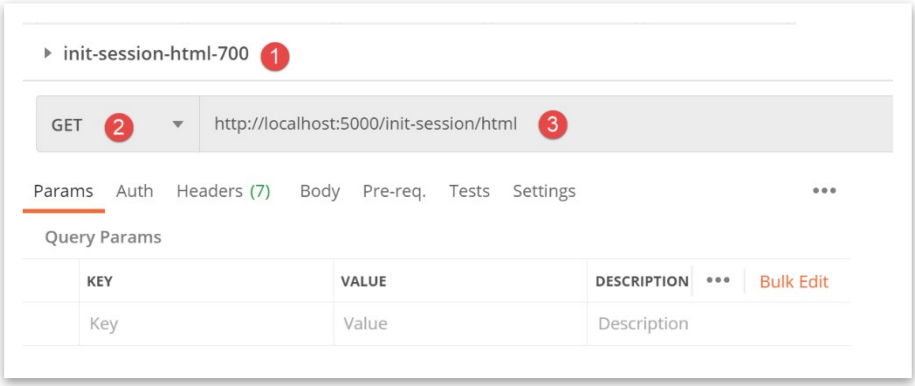
32.5.7 Tests [Postman]

Nous allons exécuter des requêtes produidant les codes **[700, 201]** qui affichent la vue d'authentification :

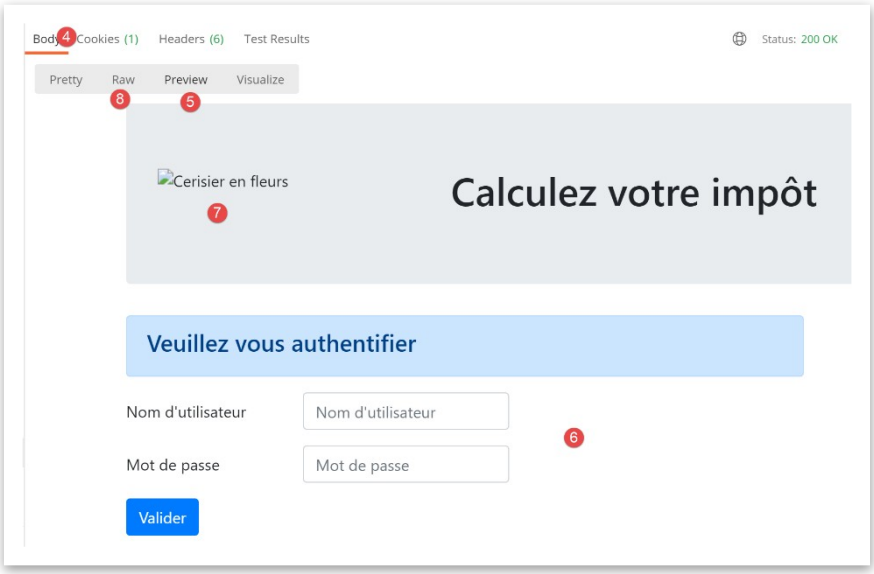
- **[init-session-html-700]** : 700 est le code d'état à l'issue d'une action **[init-session]** réussie : on présente alors le formulaire d'authentification vide ;
- **[authentifier-utilisateur-201]** : 201 est le code d'état à l'issue d'une action **[authentifier-utilisateur]** ayant échoué (identifiants non reconnus) : on présente alors le formulaire d'authentification pour qu'il soit corrigé ;

Il suffit de les réutiliser et de voir s'ils affichent bien la vue d'authentification. On montre ici deux cas :

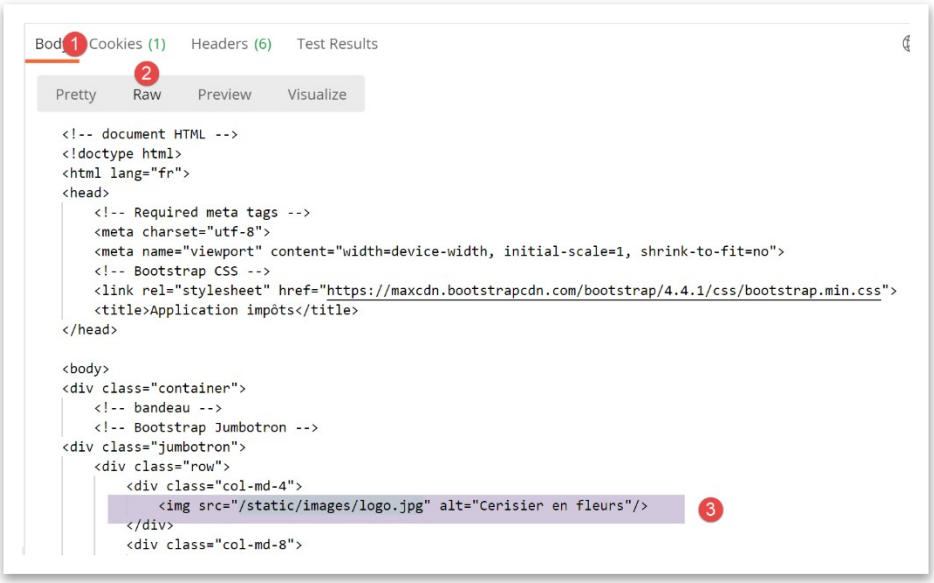
Cas 1 : [init-session-html-700], début d'une session HTML ;



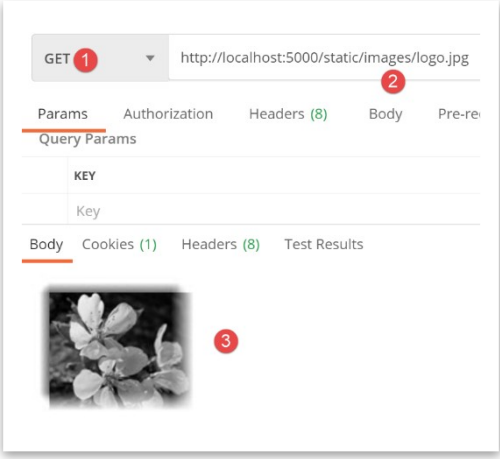
La réponse est la suivante :



- en **[5]**, le mode **[Preview]** permet de visualiser la page HTML reçue ;
- en **[6]**, on a bien le formulaire vide attendu ;
- en **[7]**, Postman n'a pas suivi le lien de l'image de la page ;
- en **[8]**, le mode **[Raw]** donne accès au HTML reçu ;

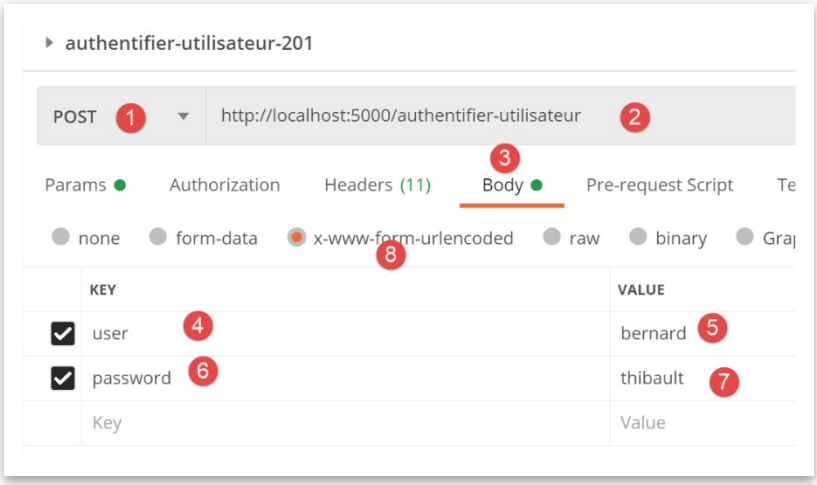


- en **[3]**, le lien que Postman n'a pas chargé. Il a affiché la valeur de l'attribut **[alt=alternative]** qui est affichée lorsque l'image ne peut être chargée. Ici c'est plutôt que Postman n'a pas voulu la charger. On peut le vérifier en demandant l'URL **[http://localhost :5000/static/images.logo.jpg]** avec Postman :



Cas 2 : **[authentifier-utilisateur-201]**, authentification erronée

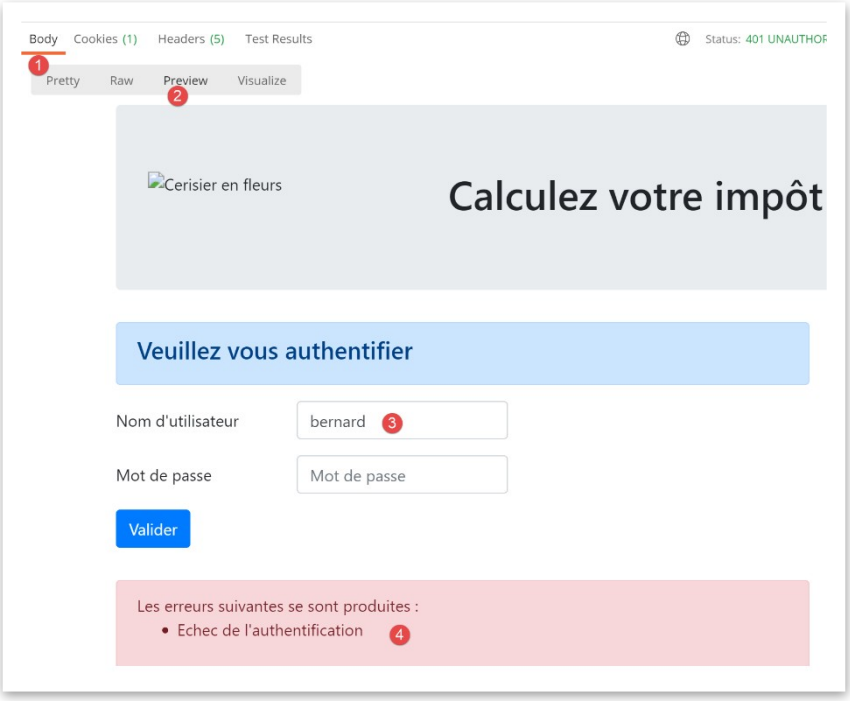
Maintenant, faisons une authentification erronée, après avoit fait une initialisation de session HTML réussie :



Ci-dessus :

- en **[4,7]** : la requête poste la chaîne **[user=bernard&password=thibault]** ;

La réponse est la suivante :



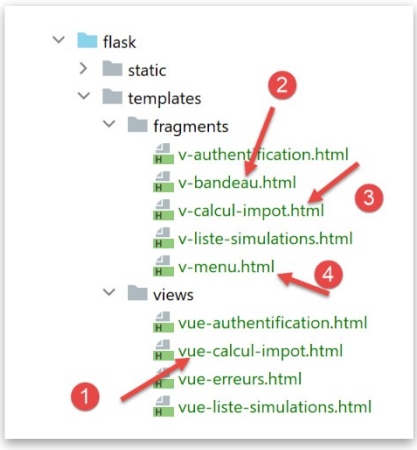
- en **[4]**, un message d'erreur est affiché ;
- en **[3]**, l'utilisateur erroné a été réaffiché ;

32.5.8 Conclusion

Nous avons pu tester la vue **[vue-authentification.html]** sans avoir écrit les autres vues. Cela a été possible parce que :

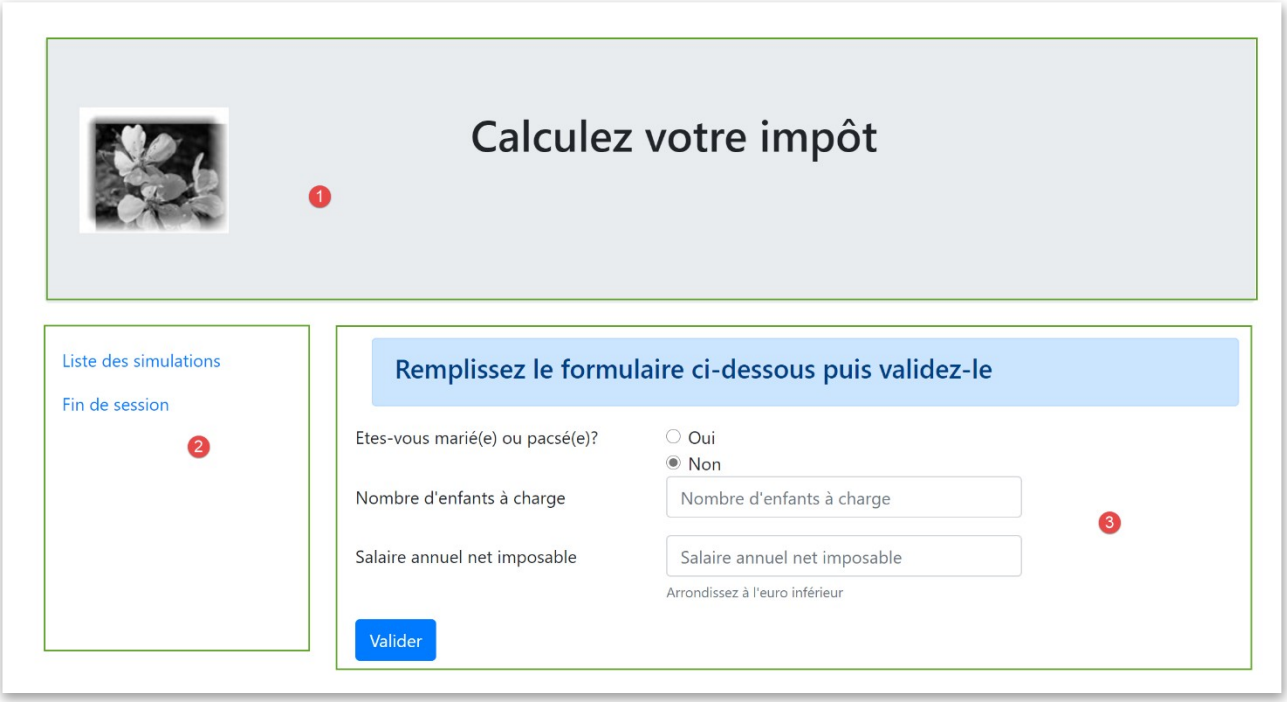
- tous les contrôleurs sont écrits ;
- **[Postman]** nous permet d'émettre des requêtes vers le serveur sans avoir besoin de toutes les vues. Lorsqu'on écrit les contrôleurs, il faut être prêt à traiter des requêtes qu'aucune vue ne permettrait. On ne doit jamais se dire a priori « cette requête est impossible » . Il faut vérifier ;

32.6 La vue de calcul de l'impôt



32.6.1 Présentation de la vue

La vue de calcul de l'impôt est la suivante :



La vue a trois parties :

- 1 : le bandeau supérieur est généré par le fragment **[v-bandeau.html]** déjà présenté ;
- 2 : le formulaire de calcul de l'impôt généré par le fragment **[v-calcul-impot.html]** ;
- 3 : un menu présentant deux liens, généré par le fragment **[v-menu.html]** ;

La vue de calcul de l'impôt est générée par le code **[vue-calcul-impot.html]** suivant :

```
<!-- document HTML -->
<!doctype html>
<html lang="fr">
<head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
    <title>Application impôts</title>
</head>

<body>
<div class="container">
    <!-- bandeau -->
    {% include "fragments/v-bandeau.html" %}
    <!-- ligne à deux colonnes -->
    <div class="row">
        <!-- le menu -->
        <div class="col-md-3">
            {% include "fragments/v-menu.html" %}
        </div>
        <!-- le formulaire de calcul -->
        <div class="col-md-9">
            {% include "fragments/v-calcul-impot.html" %}
        </div>
    </div>
    <!-- cas du succès -->

    {% if modèle.success %}
    <!-- on affiche une alerte de réussite -->
    <div class="row">
        <div class="col-md-3">

        </div>
        <div class="col-md-9">
            <div class="alert alert-success" role="alert">
```

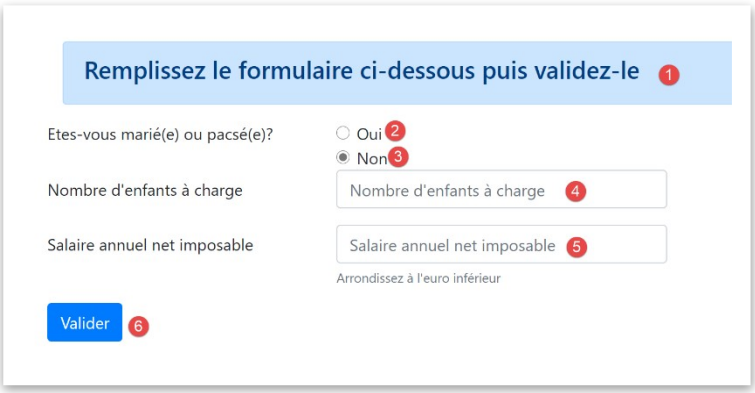
```
38.                 {{modèle.impôt}}</br>
39.                 {{modèle.décôte}}</br>
40.                 {{modèle.réduction}}</br>
41.                 {{modèle.surcôte}}</br>
42.                 {{modèle.taux}}</br>
43.             </div>
44.         </div>
45.     </div>
46.     {% endif %}
47.
48.     {% if modèle.error %}
49.     <!-- liste des erreurs sur 9 colonnes -->
50.     <div class="row">
51.         <div class="col-md-3">
52.
53.         </div>
54.         <div class="col-md-9">
55.             <div class="alert alert-danger" role="alert">
56.                 Les erreurs suivantes se sont produites :
57.                 <ul>{{modèle.erreurs | safe}}</ul>
58.             </div>
59.         </div>
60.     </div>
61.     {% endif %}
62. </div>
63. </body>
64. </html>
```

Commentaires

- nous ne commentons que des nouveautés encore non rencontrées ;
- ligne 16 : inclusion du bandeau supérieur de la vue dans la première ligne Bootstrap de la vue ;
- ligne 21 : inclusion du menu qui occupera trois colonnes de la seconde ligne Bootstrap de la vue (lignes 18, 20) ;
- ligne 25 : inclusion du formulaire de calcul d'impôt qui occupera neuf colonnes (ligne 24) de la seconde ligne Bootstrap de la vue (ligne 18) ;
- lignes 30-46 : si le calcul de l'impôt réussit **`[modèle.success=True]`**, alors le résultat du calcul de l'impôt est affiché dans un cadre vert (lignes 37-43). Ce cadre est dans la troisième ligne Bootstrap de la vue (ligne 32) et occupe neuf colonnes (ligne 36) à droite de trois colonnes vides (lignes 33-35). Ce cadre sera donc sous le formulaire de calcul de l'impôt ;
- lignes 48-61 : si le calcul de l'impôt échoue **`[modèle.error=True]`**, alors un message d'erreur est affiché dans un cadre rose (lignes 55-58). Ce cadre est dans la troisième ligne Bootstrap de la vue (ligne 50) et occupe neuf colonnes (ligne 54) à droite de trois colonnes vides (lignes 51-53). Ce cadre sera donc lui aussi sous le formulaire de calcul de l'impôt ;

32.6.2 Le fragment `[v-calcul-impot.html]`

Le fragment **`[v-calcul-impot.html]`** affiche le formulaire de calcul de l'impôt de l'application web :



Le code du fragment **`[v-calcul-impot.html]`** est le suivant :

```
1.  <!-- formulaire HTML posté -->
2.  <form method="post" action="/calculer-impot">
3.      <!-- message sur 12 colonnes sur fond bleu -->
4.      <div class="col-md-12">
5.          <div class="alert alert-primary" role="alert">
6.              <h4>Remplissez le formulaire ci-dessous puis validez-le</h4>
7.          </div>
8.      </div>
9.      <!-- éléments du formulaire -->
```

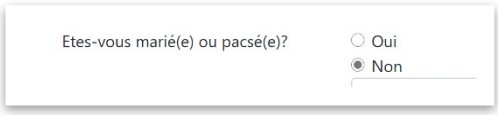
```
10.     <fieldset class="form-group">
11.         <!-- première ligne sur 9 colonnes -->
12.         <div class="row">
13.             <!-- libellé sur 4 colonnes -->
14.             <legend class="col-form-label col-md-4 pt-0">Etes-vous marié(e) ou pacsé(e)?</legend>
15.             <!-- boutons radio sur 5 colonnes-->
16.             <div class="col-md-5">
17.                 <div class="form-check">
18.                     <input class="form-check-input" type="radio" name="marié" id="gridRadios1" value="oui" {
    {modèle.checkedOui}}>
19.                     <label class="form-check-label" for="gridRadios1">
20.                         Oui
21.                     </label>
22.                 </div>
23.                 <div class="form-check">
24.                     <input class="form-check-input" type="radio" name="marié" id="gridRadios2" value="non" {
    {modèle.checkedNon}}>
25.                     <label class="form-check-label" for="gridRadios2">
26.                         Non
27.                     </label>
28.                 </div>
29.             </div>
30.         </div>
31.         <!-- deuxième ligne sur 9 colonnes -->
32.         <div class="form-group row">
33.             <!-- libellé sur 4 colonnes -->
34.             <label for="enfants" class="col-md-4 col-form-label">Nombre d'enfants à charge</label>
35.             <!-- zone de saisie numérique du nombre d'enfants sur 5 colonnes -->
36.             <div class="col-md-5">
37.                 <input type="number" min="0" step="1" class="form-control" id="enfants" name="enfants" place
    holder="Nombre d'enfants à charge" value="{{modèle.enfants}}" required>
38.             </div>
39.         </div>
40.         <!-- troisème ligne sur 9 colonnes -->
41.         <div class="form-group row">
42.             <!-- libellé sur 4 colonnes -->
43.             <label for="salaire" class="col-md-4 col-form-label">Salaire annuel net imposable</label>
44.             <!-- zone de saisie numérique pour le salaire sur 5 colonnes -->
45.             <div class="col-md-5">
46.                 <input type="number" min="0" step="1" class="form-control" id="salaire" name="salaire" place
    holder="Salaire annuel net imposable" aria-describedby="salaireHelp" value="{{modèle.salaire}}" required>
47.                 <small id="salaireHelp" class="form-text text-muted">Arrondissez à l'euro inférieur</small>
48.             </div>
49.         </div>
50.         <!-- quatrième ligne, bouton [submit] sur 5 colonnes -->
51.         <div class="form-group row">
52.             <div class="col-md-5">
53.                 <button type="submit" class="btn btn-primary">Valider</button>
54.             </div>
55.         </div>
56.     </fieldset>
57.
58. </form>
```

Commentaires

- ligne 2 : le formulaire HTML sera posté (attribut **[method]**) à l'URL **[/calculer-impot]** (attribut **[action]**). Les valeurs postées seront les valeurs des zones de saisie :
  - la valeur du bouton radio coché sous la forme :
    - **[marié=oui]** si le bouton radio **[Oui]** est coché (lignes 17-22). **[marié]** est la valeur de l'attribut **[name]** de la ligne 18, **[oui]** la valeur de l'attribut **[value]** de la ligne 18 ;
    - **[marié=non]** si le bouton radio **[Non]** est coché (lignes 23-28). **[marié]** est la valeur de l'attribut **[name]** de la ligne 24, **[non]** la valeur de l'attribut **[value]** de la ligne 24 ;
  - la valeur de la zone de saisie numérique de la ligne 37 sous la forme **[enfants=xx]** où **[enfants]** est la valeur de l'attribut **[name]** de la ligne 37, et **[xx]** la valeur saisie par l'utilisateur au clavier ;
  - la valeur de la zone de saisie numérique de la ligne 46 sous la forme **[salaire=xx]** où **[salaire]** est la valeur de l'attribut **[name]** de la ligne 46, et **[xx]** la valeur saisie par l'utilisateur au clavier ;

    Finalement, la valeur postée aura la forme **[marié=xx&enfants=yy&salaire=zz]**.

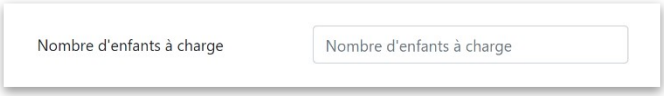
  - les valeurs saisies seront postées lorsque l'utilisateur cliquera sur le bouton de type **[submit]** de la ligne 53 ;
- lignes 16-30 : les deux boutons radio :



Les deux boutons radio font partie du même groupe de boutons radio car ils ont le même attribut **[name]** (lignes 18, 24). Le navigateur s'assure que dans un groupe de boutons radio, un seul est coché à un moment donné. Donc cliquer sur l'un désactive celui qui était coché auparavant ;

- ce sont des boutons radio à cause de l'attribut **[type="radio"]** (lignes 18, 24) ;
- à l'affichage du formulaire (avant saisie), l'un des boutons radio devra être coché : il suffit pour cela d'ajouter l'attribut **[checked='checked']** à la balise <input **type="radio"**> concernée. Cela est réalisé avec des variables dynamiques :
  - **[modèle.checkedOui]** à la ligne 18 ;
  - **[modèle->checkedNon]** à la ligne 24 ;

  Ces variables feront partie du modèle de la vue.
- ligne 37 : une zone de saisie numérique **[type="number"]** avec une valeur minimale de 0 **[min="0"]**. Dans les navigateurs récents, cela signifie que l'utilisateur ne pourra saisir qu'un nombre >=0. Sur ces mêmes navigateurs récents, la saisie peut être faite avec un variateur qu'on peut cliquer à la hausse ou à la baisse. L'attribut **[step="1"]** de la ligne 37 indique que le variateur opèrera avec des sauts de 1 unité. Cela a pour conséquence que le variateur ne prendra pour valeur que des entiers progressant de 0 à n avec un pas de 1. Pour la saisie manuelle, cela signifie que les nombres avec virgule ne seront pas acceptés ;

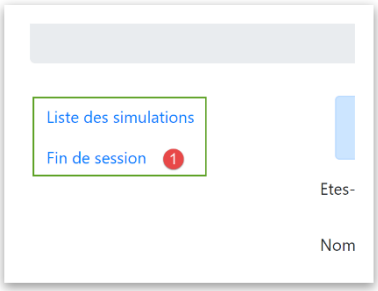


- ligne 37 : lors de certains affichages, la zone de saisie des enfants devra être préremplie avec la dernière saisie faite dans cette zone. On utilise pour cela l'attribut **[value]** qui fixe la valeur à afficher dans la zone de saisie. Cette valeur sera dynamique et générée par la variable **[modèle.enfants]** ;
- ligne 37 : l'attribut **[required]** force l'utilisateur à saisir une donnée pour que le formulaire soit validé ;
- ligne 46 : mêmes explications pour la saisie du salaire que pour celle des enfants ;
- ligne 53 : le bouton de type **[submit]** qui déclenche le POST des valeurs saisies à l'URL **[/calculer-impot]** (ligne 2) ;



32.6.3 Le fragment [v-menu.html]

Ce fragment affiche un menu à gauche du formulaire de calcul de l'impôt :



Le code de ce fragment est le suivant :

```
1.  <!-- menu Bootstrap -->
2.  <nav class="nav flex-column">
3.      <!-- affichage d'une liste de liens HTML -->
4.      {% for optionMenu in modèle.optionsMenu %}
5.        <a class="nav-link" href="{{optionMenu.url}}">{{optionMenu.text}}</a>
6.      {% endfor %}
7.  </nav>
```

Commentaires

- lignes 2-7 : la balise HTML **[nav]** encadre une portion de document HTML présentant des liens de navigation vers d'autres documents ;
- ligne 5 : la balise HTML **[a]** introduit un lien de navigation :
  - **[optionMenu.url]** : est l'URL vers laquelle on navigue lorsqu'on clique sur le lien **[optionMenu.text]**. C'est alors une opération **[GET optionMenu.url]** qui est faite par le navigateur. **[optionMenu.url]** sera une URL absolue mesurée à partir de la racine **[http://machine :port/chemin]** de l'application. Ainsi en **[1]**, on créera le lien :

```
<a href='/lister-simulations'>Liste des simulations</a>
```

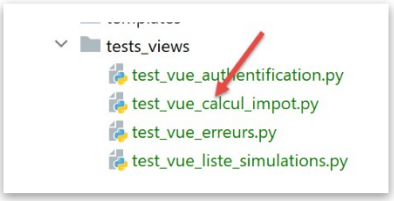
- ligne 5 : le modèle **[modèle.optionsMenu]** du fragment sera une liste de la forme :

```
['Liste des simulations':'/liste-simulations',
'Fin de session':'/fin-session']
```

- lignes 2, 7 : les classes CSS **[nav, flex-column, nav-link]** sont des classes Bootstrap qui donnent son apparence au menu ;

32.6.4 Test visuel

Nous rassemblons ces différents éléments dans le dossier **[Tests]** et nous créons un modèle de test pour la vue **[vue-calcul-impot.html]** :



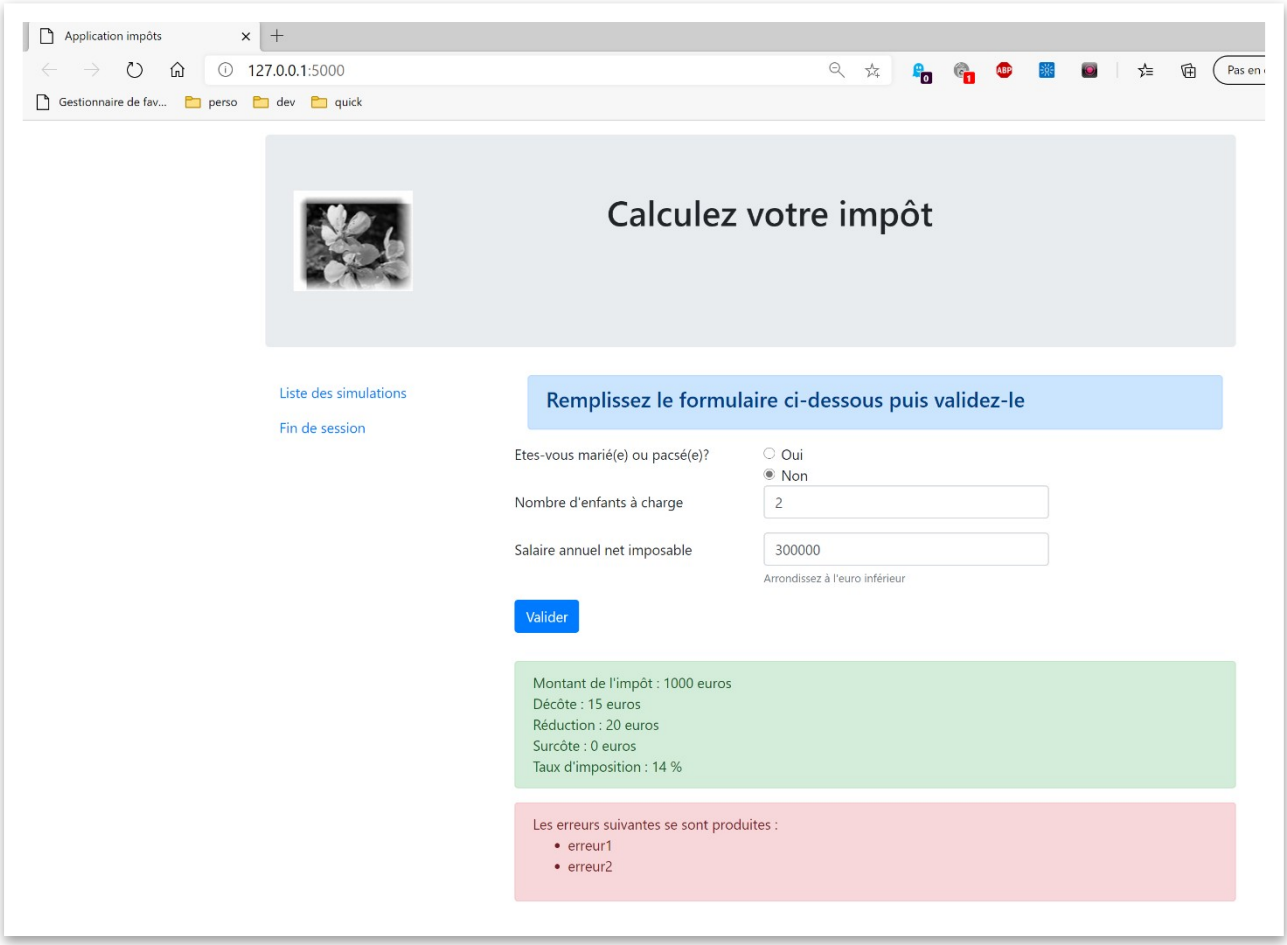
Le script de test **[test_vue_calcul_impot]** sera le suivant :

```
1.  from flask import Flask, render_template, make_response
2.
3.  # application Flask
4.  app = Flask(__name__, template_folder="../templates", static_folder="../static")
5.
6.  # Home URL
7.  @app.route('/')
8.  def index():
9.      # on encapsule les données de la pagé dans modèle
10.     modèle = {}
11.     # formulaire
12.     modèle["checkedOui"] = ""
13.     modèle["checkedNon"] = 'checked="checked"'
14.     modèle["enfants"] = 2
15.     modèle["salaire"] = 300000
16.     # message de réussite
17.     modèle["success"] = True
18.     modèle["impôt"] = "Montant de l'impôt : 1000 euros"
19.     modèle["décôte"] = "Décôte : 15 euros"
20.     modèle["réduction"] = "Réduction : 20 euros"
21.     modèle["surcôte"] = "Surcôte : 0 euros"
22.     modèle["taux"] = "Taux d'imposition : 14 %"
23.     # message d'erreur
24.     modèle["error"] = True
25.     erreurs = ["erreur1", "erreur2"]
26.     # on construit une liste HTML des erreurs
27.     content = ""
28.     for erreur in erreurs:
29.         content += f"<li>{erreur}</li>"
30.     modèle["erreurs"] = content
31.     # menu
32.     modèle["optionsMenu"] = [
33.         {"text": 'Liste des simulations', "url": '/lister-simulations'},
34.         {"text": 'Fin de session', "url": '/fin-session'}]
35.     # affichage de la page
36.     return make_response(render_template("views/vue-calcul-impot.html", modèle=modèle))
37.
38. # main
39. if __name__ == '__main__':
40.     app.config.update(ENV="development", DEBUG=True)
41.     app.run()
```

Commentaires

- lignes 9-34 : on initialise toutes les parties dynamiques de la vue **[vue-calcul-impot.html]** et des fragments **[v-calcul-impot.html]** et **[v-menu.html]** ;
- ligne 36 : on affiche la vue **[vue-calcul-impot.html]** ;

Lorsqu'on exécute le script de test **[test_vue_calcul_impot]**, on obtient le résultat suivant :



On travaille sur cette vue jusqu'à ce que le résultat obtenu visuellement nous convienne. On peut ensuite passer à l'intégration de la vue dans l'application web en cours d'écriture.

32.6.5 Calcul du modèle de la vue

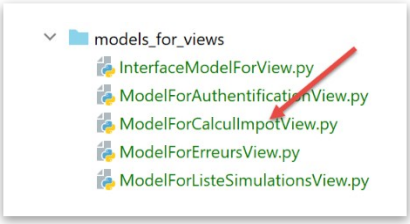
Une fois l'aspect visuel de la vue déterminé, on peut procéder au calcul du modèle de la vue en conditions réelles. Rappelons les codes d'état qui mènent à cette vue. On les trouve dans le fichier de configuration :

```
1.              {
2.                  # vue du calcul de l'impôt
3.                  "états": [
4.                      # /authentifier-utilisateur réussite
5.                      200,
6.                      # /calculer-impot réussite
7.                      300,
8.                      # /calculer-impot échec
9.                      301,
10.                     # /afficher-calcul-impot
11.                     800
12.                 ],
13.                 "view_name": "views/vue-calcul-impot.html",
14.                 "model_for_view": ModelForCalculImpotView()
15.             },
```

Ce sont donc les codes d'état **[200, 300, 301, 800]** qui font afficher la vue de calcul de l'impôt. Pour retrouver la signification de ces codes, on peut s'aider des tests **[Postman]** réalisés sur l'application jSON :

- **[authentifier-utilisateur-200]** : 200 est le code d'état à l'issue d'une action **[authentifier-utilisateur]** réussie : on présente alors le formulaire de calcul d'impôt vide ;
- **[calculer-impot-300]** : 300 est le code d'état à l'issue d'une action **[calculer-impot]** réussie. On affiche alors le formulaire de calcul avec les données qui y ont été saisies et le montant de l'impôt. L'utilisateur peut alors refaire un autre calcul ;
- le code d'état **[301]** est celui obtenu pour un calcul d'impôt erroné ;
- le code d'état **[800]** sera présenté ultérieurement. Nous ne l'avons pas encore rencontré ;

Maintenant que nous savons à quels moments doit être affiché le formulaire de calcul de l'impôt, on peut calculer son modèle dans la classe **[ModelForCalculImpotView]** :




```
1.  from flask import Request
2.  from werkzeug.local import LocalProxy
3.
4.  from InterfaceModelForView import InterfaceModelForView
5.
6.  class ModelForCalculImpotView(InterfaceModelForView):
7.
8.      def get_model_for_view(self, request: Request, session: LocalProxy, config: dict, résultat: dict) -> dic
    t:
9.          # on encapsule les données de la vue dans modèle
10.         modèle = {}
11.         # état de l'application
12.         état = résultat["état"]
13.         # le modèle dépend de l'état
14.         if état in [200, 800]:
15.             # affichage initial d'un formulaire vide
16.             modèle["success"] = False
17.             modèle["error"] = False
18.             modèle["checkedNon"] = 'checked="checked"'
19.             modèle["checkedOui"] = ""
20.             modèle["enfants"] = ""
21.             modèle["salaire"] = ""
22.         elif état == 300:
23.             # réussite du calcul - affichage du résultat
24.             modèle["success"] = True
25.             modèle["error"] = False
26.             modèle["impôt"] = f"Montant de l'impôt : {résultat['réponse']['impôt']} euros"
27.             modèle["décôte"] = f'Décôte : {résultat["réponse"]["décôte"]} euros'
28.             modèle["réduction"] = f"Réduction : {résultat['réponse']['réduction']} euros"
29.             modèle["surcôte"] = f'Surcôte : {résultat["réponse"]["surcôte"]} euros'
30.             modèle["taux"] = f"Taux d'imposition :  {résultat['réponse']['taux'] * 100} %"
31.             # formulaire rétabli avec les valeurs saisies
32.             modèle["checkedOui"] = 'checked="checked"' if request.form.get("marié") == "oui" else ""
33.             modèle["checkedNon"] = 'checked="checked"' if request.form.get("marié") == "non" else ""
34.             modèle["enfants"] = request.form.get("enfants")
35.             modèle["salaire"] = request.form.get("salaire")
36.         elif état == 301:
37.             # erreur rencontrée - formulaire rétabli avec les valeurs saisies
38.             modèle["checkedOui"] = 'checked="checked"' if request.form.get("marié") == "oui" else ""
39.             modèle["checkedNon"] = 'checked="checked"' if request.form.get("marié") == "non" else ""
40.             modèle["enfants"] = request.form.get("enfants")
41.             modèle["salaire"] = request.form.get("salaire")
42.             # erreur
43.             modèle["success"] = False
44.             modèle["error"] = True
45.             modèle["erreurs"] = ""
46.             for erreur in résultat['réponse']:
47.                 modèle['erreurs'] += f"<li>{erreur}</li>"
48.
49.         # options du menu
50.         modèle["optionsMenu"] = [
51.             {"text": 'Liste des simulations', "url": '/lister-simulations'},
52.             {"text": 'Fin de session', "url": '/fin-session'}]
53.         # on rend le modèle
```

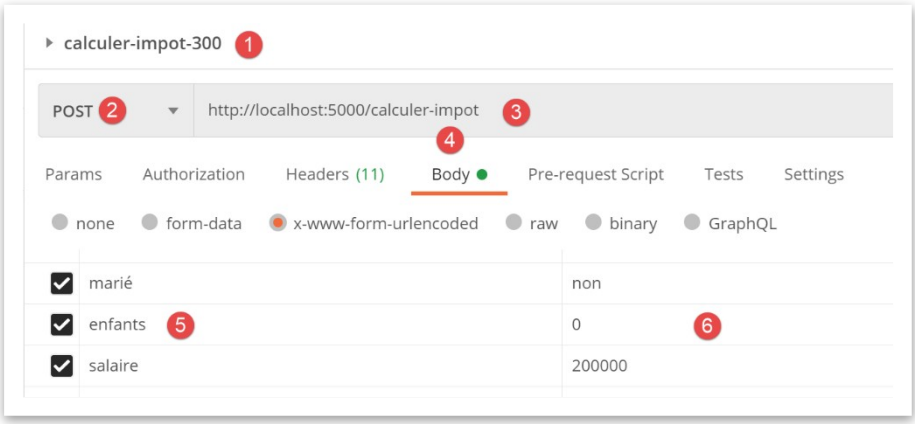
```
54.         return modèle
```

Commentaires

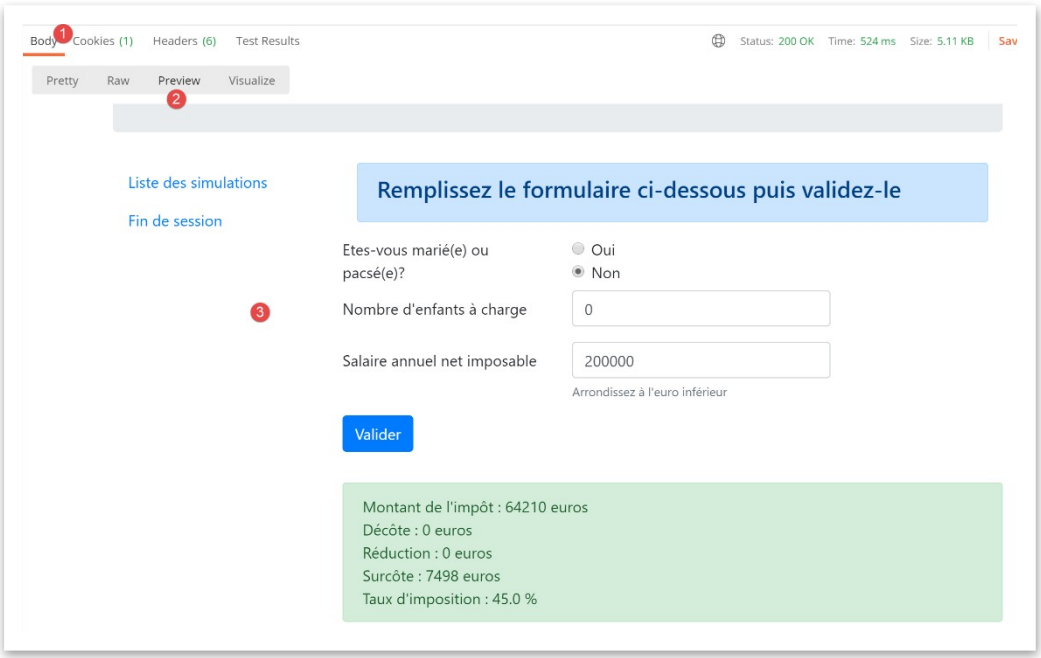
- ligne 12 : la vue à afficher dépend du code d'état rendu par le contrôleur ;
- lignes 14-21 : affichage d'un formulaire vide ;
- lignes 22-35 : cas du calcul d'impôt réussi. On réaffiche les valeurs saisies ainsi que le montant de l'impôt ;
- lignes 36-47 : cas de l'échec du calcul d'impôt ;
- lignes 49-52 : calcul des deux options du menu ;

32.6.6 Tests `[Postman]`

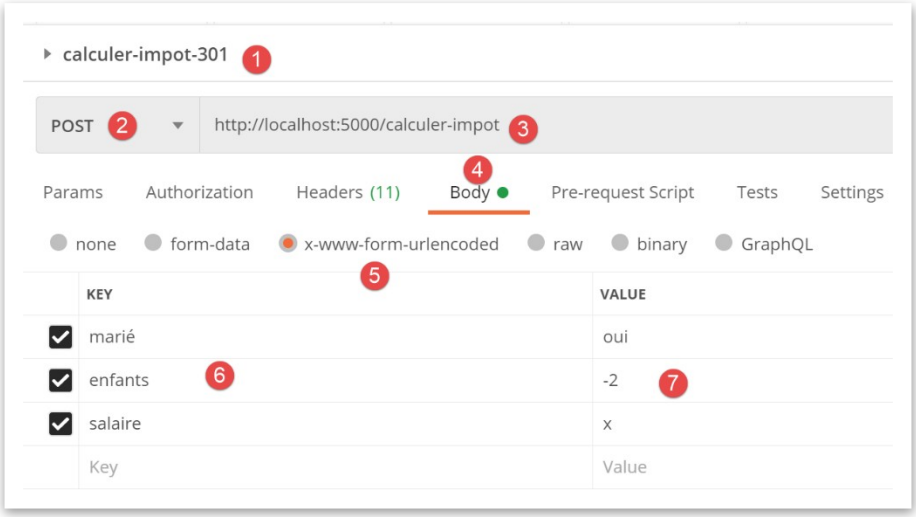
On initialise une session HTML avec la requête **`[init-session-html-700]`** puis on s'authentifie avec la requête **`[authentifier-utilisateur-200]`**. Puis nous utilisons la requête **`[calculer-impot-300]`** suivante :



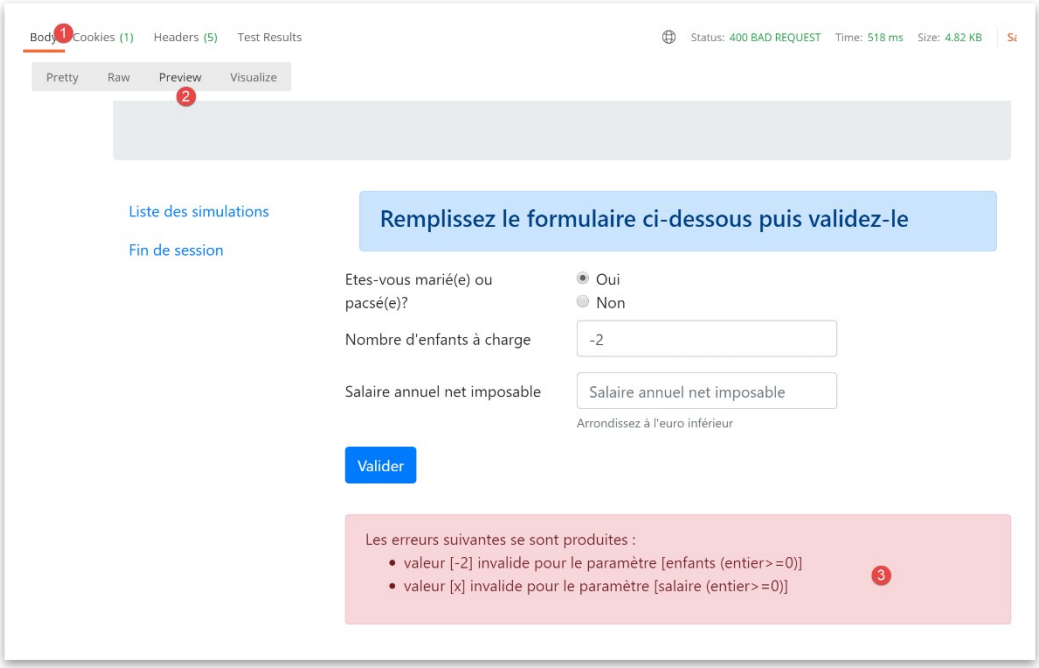
La réponse du serveur est la suivante :



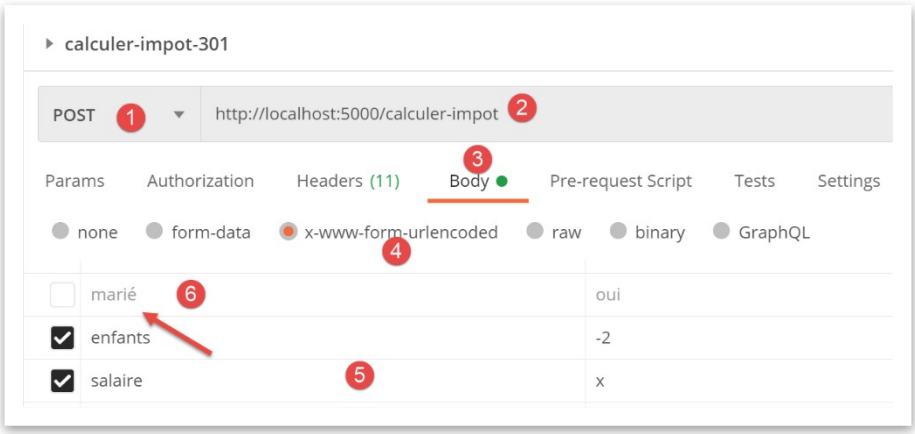
Maintenant essayons la requête **`[calculer-impot-301]`** suivante :



La réponse du serveur est la suivante :

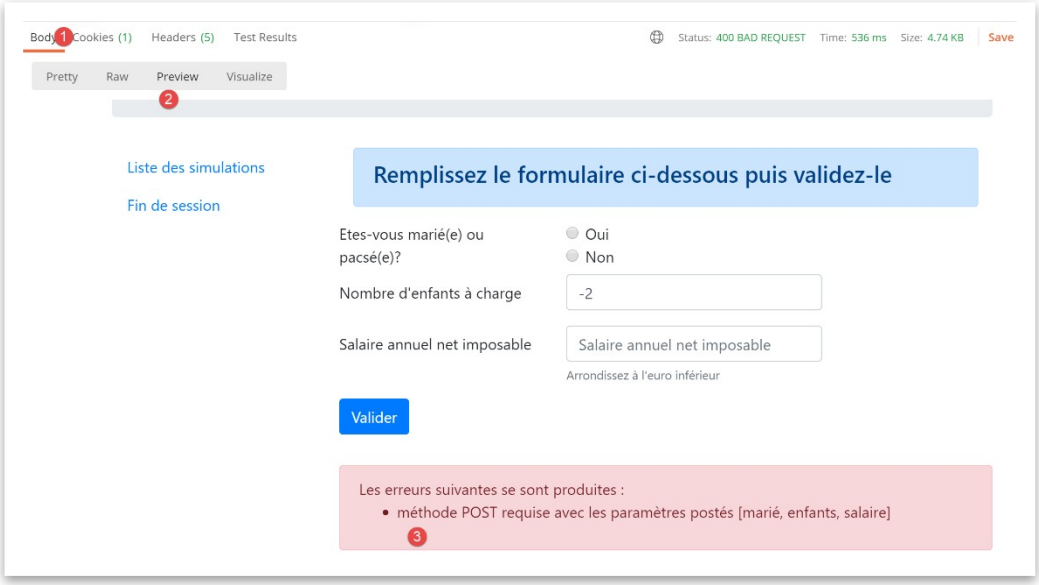


Maintenant essayons un cas inattendu, celui où il manque des paramètres au POST. Ce cas n'est pas possible dans le fonctionneent normal de l'application. Mais n'importe qui peut 'bricoler' une requête HTTP comme nous le faisons maintenant :



- en **[6]**, nous avons décoché le paramètre posté **[marié]** ;

La réponse du serveur est la suivante :



- en **[3]**, le message d'erreur du serveur ;

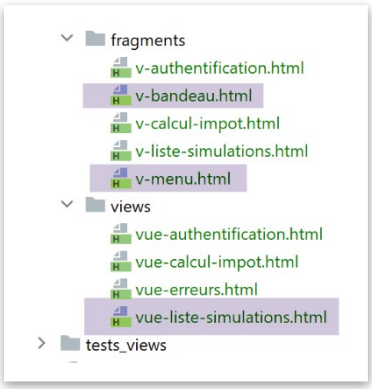
Dans cette application, nous avions le choix. Nous pouvions donner à ce cas d'erreur un code d'état qui envoie la page des erreurs inattendues. Nous avons dans cette application choisi pour chaque contrôleur deux codes d'état :

- **[xx0]** : pour une réussite ;
- **[xx1]** : pour un échec ;

Pour les cas d'échec on peut diversifier les codes d'état pour avoir une gestion plus fine des erreurs. Nous aurions pu avoir par exemple :

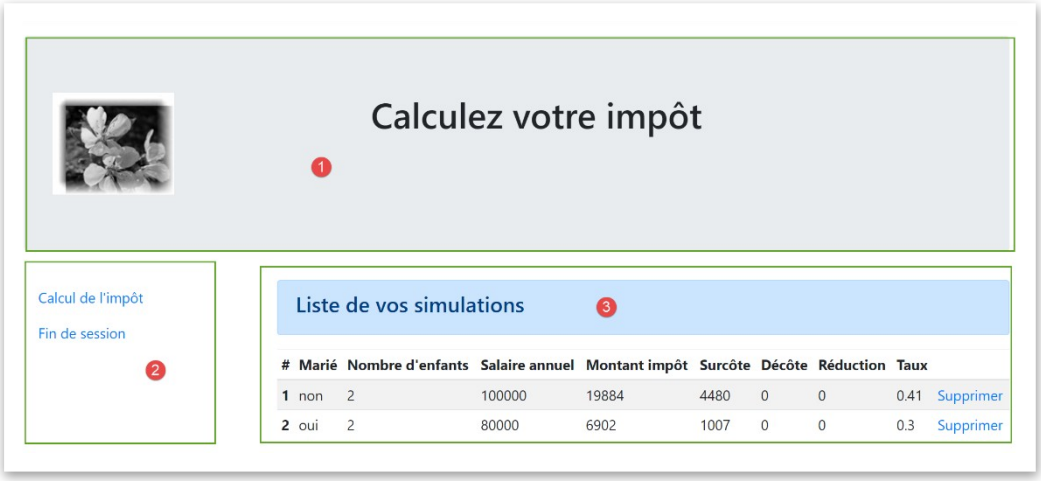
- **[xx1]** : pour des erreurs à afficher sur la page qui a provoqué l'erreur ;
- **[xx2]** : pour des erreurs inattendues dans le cadre d'une utilisation normale de l'application ;

32.7 La vue de la liste des simulations



32.7.1 Présentation de la vue

La vue qui présente la liste des simulations est la suivante :



La vue générée par le code **[vue-liste-simulations.html]** a trois parties :

- 1 : le bandeau supérieur est généré par le fragment **[v-bandeau.html]** déjà présenté ;
- 3 : le tableau des simulations généré par le fragment **[v-liste-simulations.html]** ;
- 2 : un menu présentant deux liens, généré par le fragment **[v-menu.html]** déjà présenté ;

La vue des simulations est générée par le code **[vue-liste-simulations.html]** suivant :

```
1.  <!-- document HTML -->
2.  <!doctype html>
3.  <html lang="fr">
4.  <head>
5.      <!-- Required meta tags -->
6.      <meta charset="utf-8">
7.      <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
8.      <!-- Bootstrap CSS -->
9.      <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
10.     <title>Application impôts</title>
11. </head>
12.
13. <body>
14. <div class="container">
15.     <!-- bandeau -->
16.     {% include "fragments/v-bandeau.html" %}
17.     <!-- ligne à deux colonnes -->
18.     <div class="row">
19.         <!-- menu sur trois colonnes-->
20.         <div class="col-md-3">
21.             {% include "fragments/v-menu.html" %}
22.
23.         </div>
24.         <!-- liste des simulations sur 9 colonnes-->
25.         <div class="col-md-9">
26.             {% include "fragments/v-liste-simulations.html" %}
27.         </div>
28.     </div>
29. </div>
30. </body>
31. </html>
```

Commentaires

- ligne 16 : inclusion du bandeau de l'application **[1]** ;
- ligne 21 : inclusion du menu **[2]**. Il sera affiché sur trois colonnes sous le bandeau ;
- ligne 26 : inclusion du tableau des simulations **[3]**. Il sera affiché sur neuf colonnes sous le bandeau et à droite du menu ;

Nous avons déjà commenté deux des trois fragments de cette vue :

- **[v-bandeau.html]** : au paragraphe lien ;
- **[v-menu.html]** : au paragraphe lien ;

Le fragment **[v-liste-simulations.html]** est le suivant :

```
1.  {% if modèle.simulations is undefined or modèle.simulations|length==0 %}
2.  <!-- message sur fond bleu -->
3.  <div class="alert alert-primary" role="alert">
4.      <h4>Votre liste de simulations est vide</h4>
5.  </div>
6.  {% endif %}
7.
8.  {% if modèle.simulations is defined and modèle.simulations|length!=0 %}
9.  <!-- message sur fond bleu -->
10. <div class="alert alert-primary" role="alert">
11.     <h4>Liste de vos simulations</h4>
12. </div>
13.
14. <!-- tableau des simulations -->
15. <table class="table table-sm table-hover table-striped">
16.     <!-- entêtes des six colonnes du tableau -->
17.     <thead>
18.     <tr>
19.         <th scope="col">#</th>
20.         <th scope="col">Marié</th>
21.         <th scope="col">Nombre d'enfants</th>
22.         <th scope="col">Salaire annuel</th>
23.         <th scope="col">Montant impôt</th>
24.         <th scope="col">Surcôte</th>
25.         <th scope="col">Décôte</th>
26.         <th scope="col">Réduction</th>
27.         <th scope="col">Taux</th>
28.         <th scope="col"></th>
29.     </tr>
30.     </thead>
31.     <!-- corps du tableau (données affichées) -->
32.     <tbody>
33.     <!-- on affiche chaque simulation en parcourant le tableau des simulations -->
34.     {% for simulation in modèle.simulations %}
35.
36.     <!-- affichage d'une ligne du tableau avec 6 colonnes - balise <tr> -->
37.     <!-- colonne 1 : entête ligne (n° simulation) - balise <th scope='row' -->
38.     <!-- colonne 2 : valeur paramètre [marié] - balise <td> -->
39.     <!-- colonne 3 : valeur paramètre [enfants] - balise <td> -->
40.     <!-- colonne 4 : valeur paramètre [salaire] - balise <td> -->
41.     <!-- colonne 5 : valeur paramètre [impôt] (de l'impôt) - balise <td> -->
42.     <!-- colonne 6 : valeur paramètre [surcôte] - balise <td> -->
43.     <!-- colonne 7 : valeur paramètre [décôte] - balise <td> -->
44.     <!-- colonne 8 : valeur paramètre [réduction] - balise <td> -->
45.     <!-- colonne 9 : valeur paramètre [taux] (de l'impôt) - balise <td> -->
46.     <!-- colonne 10 : lien de suppression de la simulation - balise <td> -->
47.     <tr>
48.         <th scope="row">{{simulation.id}}</th>
49.         <td>{{simulation.marié}}</td>
50.         <td>{{simulation.enfants}}</td>
51.         <td>{{simulation.salaire}}</td>
52.         <td>{{simulation.impôt}}</td>
53.         <td>{{simulation.surcôte}}</td>
54.         <td>{{simulation.décôte}}</td>
55.         <td>{{simulation.réduction}}</td>
56.         <td>{{simulation.taux}}</td>
57.         <td><a href="/supprimer-simulation/{{simulation.id}}">Supprimer</a></td>
58.     </tr>
59.     {% endfor %}
60.     </tr>
61.     </tbody>
62. </table>
63. {% endif %}
```

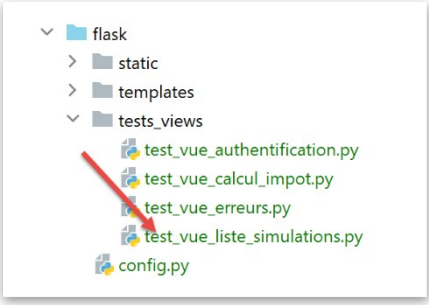
Commentaires

- un tableau HTML est réalisé avec la balise <table> (lignes 15 et 62) ;
- les entêtes des colonnes du tableau se font à l'intérieur d'une balise <thead> (*table head*, lignes 17, 30). La balise <tr> (*table row*, lignes 18 et 29) délimitent une ligne. Lignes 19-28, la balise <th> (*table header*) définit un entête de colonne. Il y en a donc dix. **[scope="col"]** indique que l'entête s'applique à la colonne. **[scope="row"]** indique que l'entête s'applique à la ligne ;
- lignes 32-61 : la balise <tbody> encadre les données affichées par le tableau ;
- lignes 47-58 : la balise <tr> encadre une ligne du tableau ;
- ligne 48 : la balise <th scope='row'> définit l'entête de la ligne. le navigateur fait ressortir cet entête ;

- lignes 49-57 : chaque balise <td> (*table data*) définit une colonne de la ligne ;
- ligne 34 : la liste des simulations sera trouvée dans le modèle **[modèle.simulations]** qui est une liste de dictionnaires ;
- ligne 57 : un lien pour supprimer la simulation. L'URL utilise le n° de la simulation affichée dans la ligne ;

32.7.2 Test visuel

Nous créons un script de test pour la vue **[vue-liste-simulations.html]** :



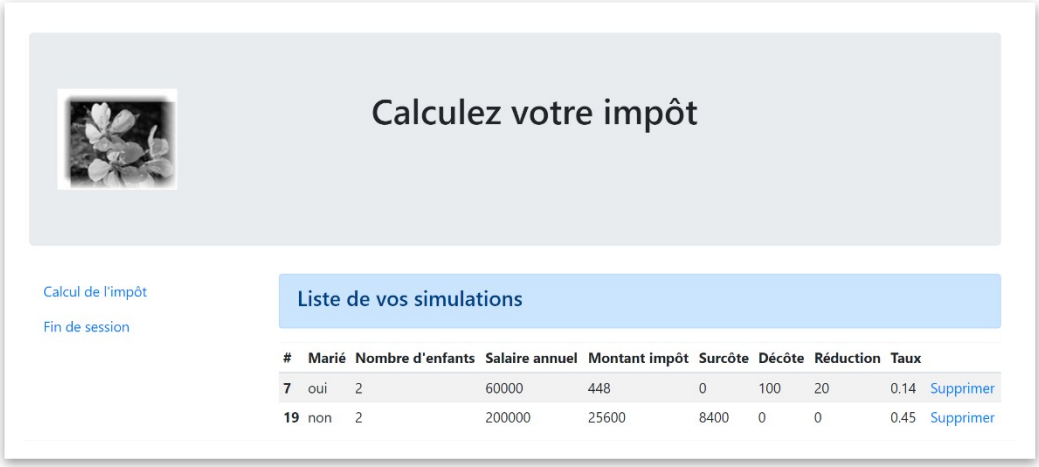
Le script **[test_vue_liste_simulations]** est le suivant :

```
1.  from flask import Flask, make_response, render_template
2.
3.  # application Flask
4.  app = Flask(__name__, template_folder="../templates", static_folder="../static")
5.
6.  # Home URL
7.  @app.route('/')
8.  def index():
9.      # on encapsule les données de la pagé dans modèle
10.     modèle = {}
11.     # on met les simulations au format attendu par la page
12.     modèle["simulations"] = [
13.         {
14.             "id": 7,
15.             "marié": "oui",
16.             "enfants": 2,
17.             "salaire": 60000,
18.             "impôt": 448,
19.             "décôte": 100,
20.             "réduction": 20,
21.             "surcôte": 0,
22.             "taux": 0.14
23.         },
24.         {
25.             "id": 19,
26.             "marié": "non",
27.             "enfants": 2,
28.             "salaire": 200000,
29.             "impôt": 25600,
30.             "décôte": 0,
31.             "réduction": 0,
32.             "surcôte": 8400,
33.             "taux": 0.45
34.         }
35.     ]
36.     # menu
37.     modèle["optionsMenu"] = [
38.         {"text": "Calcul de l'impôt", "url": '/afficher-calcul-impot'},
39.         {"text": 'Fin de session', "url": '/fin-session'}]
40.     # affichage de la page
41.     return make_response(render_template("views/vue-liste-simulations.html", modèle=modèle))
42.
43. # main
44. if __name__ == '__main__':
45.     app.config.update(ENV="development", DEBUG=True)
46.     app.run()
```

Commentaires

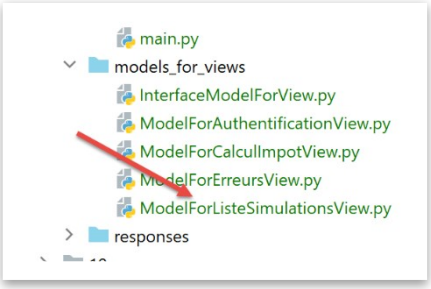
- lignes 12-35 : on met deux simulations dans le modèle
- lignes 37-39 : le tableau des options de menu ;

Affichons cette vue en exécutant ce script. On obtient le résultat suivant :



On travaille sur cette vue jusqu'à ce que le résultat obtenu visuellement nous convienne. On peut ensuite passer à l'intégration de la vue dans l'application web en cours d'écriture.

32.7.3 Calcul du modèle de la vue



Une fois l'aspect visuel de la vue déterminé, on peut procéder au calcul du modèle de la vue en conditions réelles. Rappelons les codes d'état qui mènent à cette vue. On les trouve dans le fichier de configuration :

```
{
    # vue de la liste des simulations
    "états": [
        # /lister-simulations
        500,
        # /supprimer-simulation
        600
    ],
    "view_name": "views/vue-liste-simulations.html",
    "model_for_view": ModelForListeSimulationsView()
}
```

Ce sont donc les codes d'état **[500, 600]** qui font afficher la vue des simulations. Pour retrouver la signification de ces codes, on peut s'aider des tests **[Postman]** réalisés sur l'application jSON :

- **[lister-simulations-500]** : 500 est le code d'état à l'issue d'une action **[lister-simulations]** réussie : on présente alors la liste des simulations réalisées par l'utilisateur ;
- **[supprimer-simulation-600]** : 600 est le code d'état à l'issue d'une action **[supprimer-simulation]** réussie. On présente alors la nouvelle liste des simulations obtenue après cette suppression ;

Maintenant que nous savons à quels moments doit être affichée la liste des simulations, on peut calculer son modèle dans la classe **[ModelForListeSimulationsView]** :

```
from flask import Request
from werkzeug.local import LocalProxy

from InterfaceModelForView import InterfaceModelForView

class ModelForListeSimulationsView(InterfaceModelForView):

    def get_model_for_view(self, request: Request, session: LocalProxy, config: dict, résultat: dict) -> dict:
```

```
9.          # on encapsule les données de la pagé dans modèle
10.         modèle = {}
11.         # les simulations sont trouvées dans la réponse du contrôleur qui a exécuté l'action
12.         # sous la forme d'un tableau de dictionnaires TaxPayer
13.         modèle["simulations"] = résultat["réponse"]
14.         # menu
15.         modèle["optionsMenu"] = [
16.              {"text": "Calcul de l'impôt", "url": '/afficher-calcul-impot'},
17.              {"text": 'Fin de session', "url": '/fin-session'}]
18.         # on rend le modèle
19.         return modèle
```

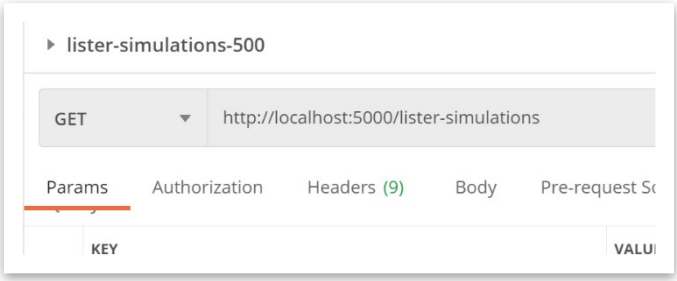
Commentaires

- lignes 13 : les simulations à afficher sont trouvées dans **[résultat["réponse"]]** ;
- lignes 15-17 : les options du menu à afficher ;

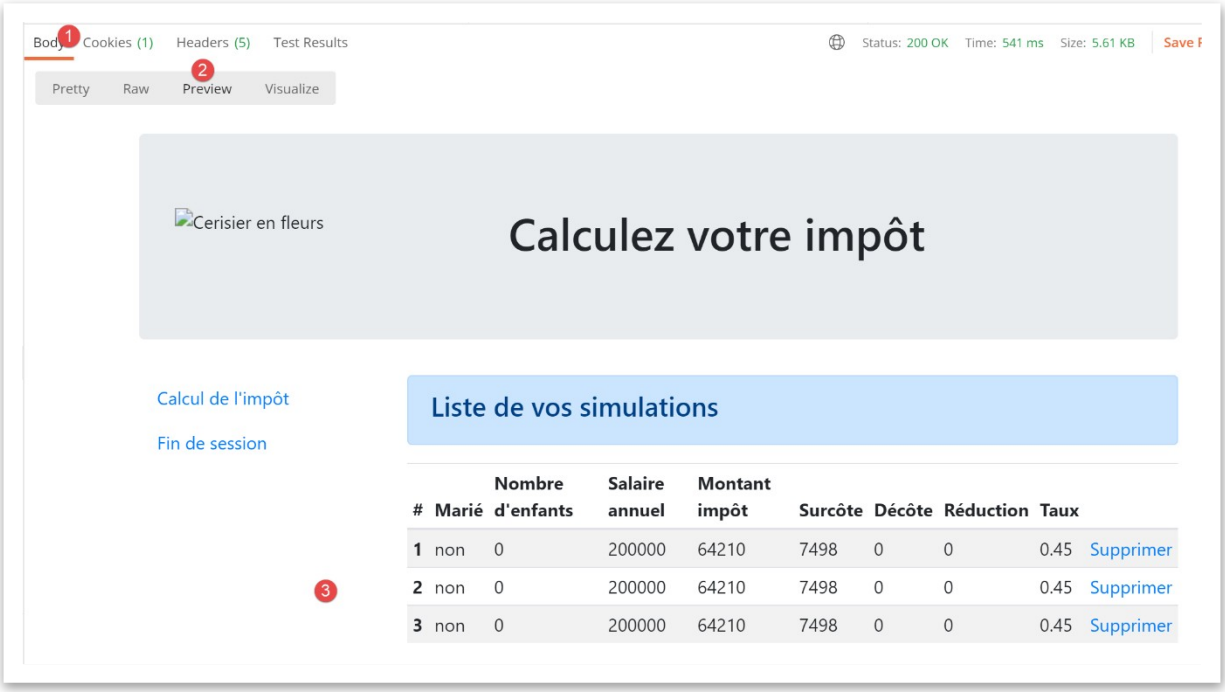
32.7.4 Tests [Postman]

On
- initialise une session HTML ;
- s'authentifie ;
- fait trois calculs d'impôt ;

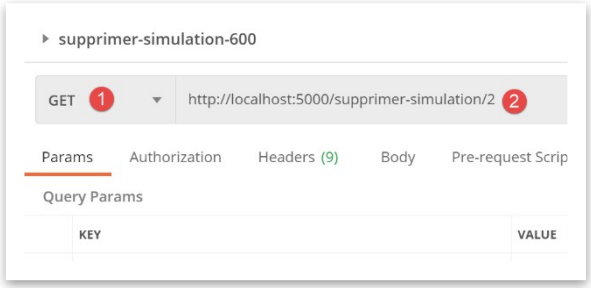
Le test **[lister-simulations-500]** nous permet d'avoir le code d'état 500. Il correspond à une demande pour voir les simulations :



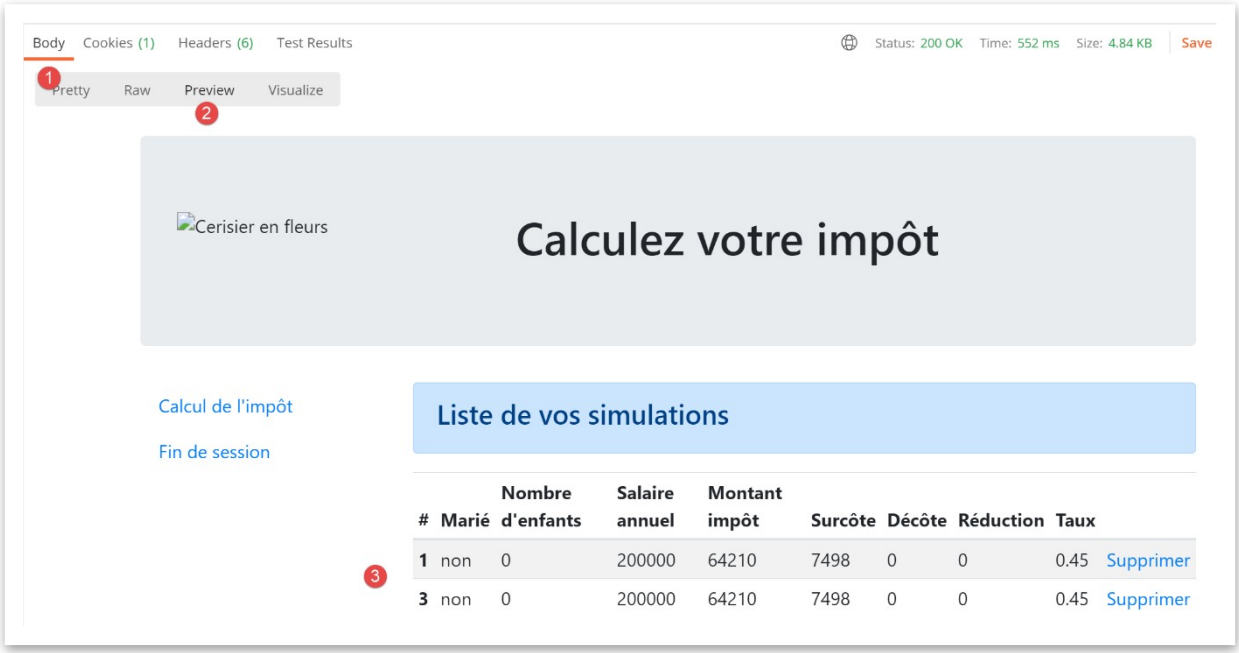
La réponse du serveur est la suivante :



Le test **[supprimer-simulation-600]** nous permet d'avoir le code d'état 600. On va ici supprimer la simulation n° 2.



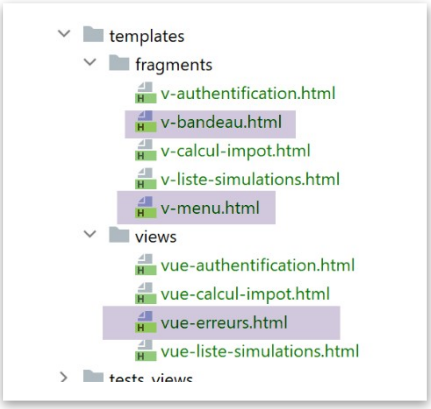
Le résultat renvoyé est une liste de simulations avec une simulation en moins :



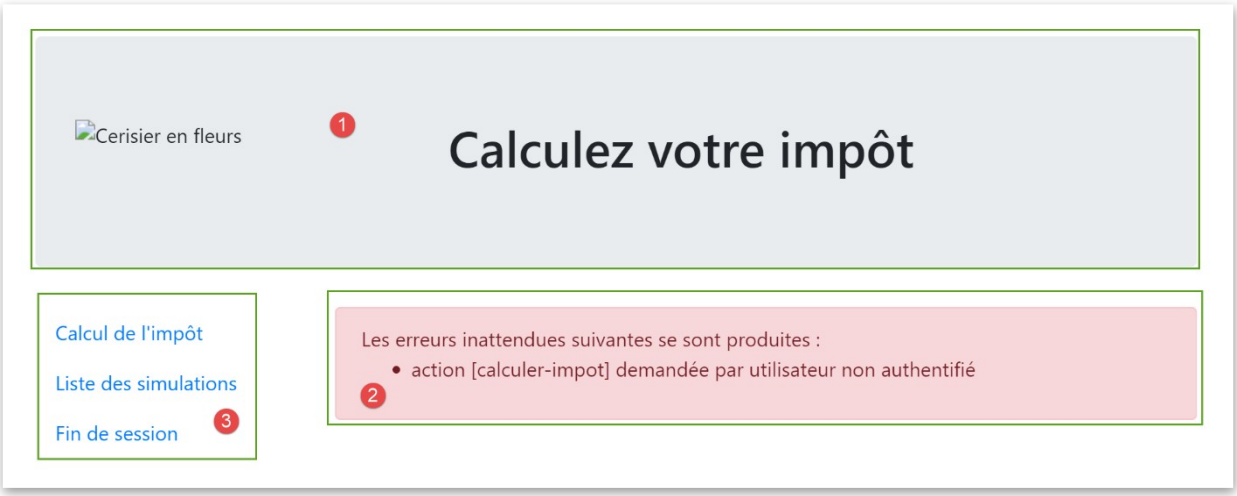
32.8 La vue des erreurs inattendues

On appelle ici, erreur inattendue, une erreur qui n'aurait pas dû se produire dans le cadre d'une utilisation normale de l'application web. Par exemple, le fait de demander un calcul d'impôt sans être authentifié. Rien n'empêche un utilisateur de taper directement l'URL **`[/calcul-impot]`** dans son navigateur. Par ailleurs, comme nous l'avons vu, il peut faire un POST sur l'URL **`[/calcul-impot]`** en n'envoyant pas les paramètres attendus. On a vu que notre application web savait répondre correctement à cette requête. On appellera, **erreur inattendue**, une erreur qui ne devrait pas se produire dans le cadre de l'application HTML. Si elle se produit, c'est que probablement quelqu'un essaie de 'hacker' l'application. Par souci de pédagogie, on a décidé d'afficher une vue d'erreurs pour ces cas. Dans la réalité, on pourrait réafficher la dernière page envoyée au client. Il suffit pour cela d'enregistrer en session, la dernière réponse HTML envoyée. En cas d'erreur inattendue, on renvoie cette réponse. Ainsi l'utilisateur aura l'impression que le serveur ne répond pas à ses erreurs puisque la page affichée ne change pas.

32.8.1 Présentation de la vue



La vue qui présente les erreurs inattendues est la suivante :



La vue générée par le code **[vue-erreurs.html]** a trois parties :

- 1 : le bandeau supérieur est généré par le fragment **[v-bandeau.html]** déjà présenté ;
- 2 : la ou les erreurs inattendues ;
- 3 : un menu présentant trois liens, généré par le fragment **[v-menu.html]** déjà présenté ;

La vue des erreurs inattendues est générée par le script **[vue-erreurs.html]** suivant :

```
1.  <!-- document HTML -->
2.  <!doctype html>
3.  <html lang="fr">
4.  <head>
5.      <!-- Required meta tags -->
6.      <meta charset="utf-8">
7.      <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
8.      <!-- Bootstrap CSS -->
9.      <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
10.     <title>Application impôts</title>
11. </head>
12.
13. <body>
14. <div class="container">
15.     <!-- bandeau sur 12 colonnes -->
16.     {% include "fragments/v-bandeau.html" %}
17.     <!-- ligne à deux sections -->
18.     <div class="row">
19.         <!-- menu sur 3 colonnes-->
20.         <div class="col-md-3">
21.             {% include "fragments/v-menu.html" %}
22.         </div>
23.         <!-- liste des erreurs sur 9 colonnes -->
24.         <div class="col-md-9">
25.             <div class="alert alert-danger" role="alert">
```

```
26.                 Les erreurs inattendues suivantes se sont produites :
27.                 <ul>{{modèle.erreurs|safe}}</ul>
28.             </div>
29.         </div>
30.     </div>
31. </div>
32. </body>
33. </html>
```

Commentaires

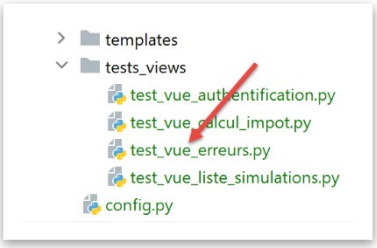
- ligne 16 : inclusion du bandeau de l'application **[1]** ;
- ligne 21 : inclusion du menu **[3]**. Il sera affiché sur trois colonnes sous le bandeau ;
- lignes 24-29 : affichage de la zone d'erreurs sur neuf colonnes ;
- ligne 25 : cet affichage se fera dans un cadre Bootstrap à fond rose ;
- ligne 26 : un texte de présentation ;
- ligne 27 : la balise <ul> encadre une liste à puces. Cette liste à puces est fournie par le modèle **[modèle.erreurs]** ;

Nous avons déjà commenté les deux fragments de cette vue :

- **[v-bandeau.html]** : au paragraphe lien ;
- **[v-menu.html]** : au paragraphe lien ;

32.8.2 Test visuel

Nous créons un script de test pour la vue **[vue-erreurs.html]** :

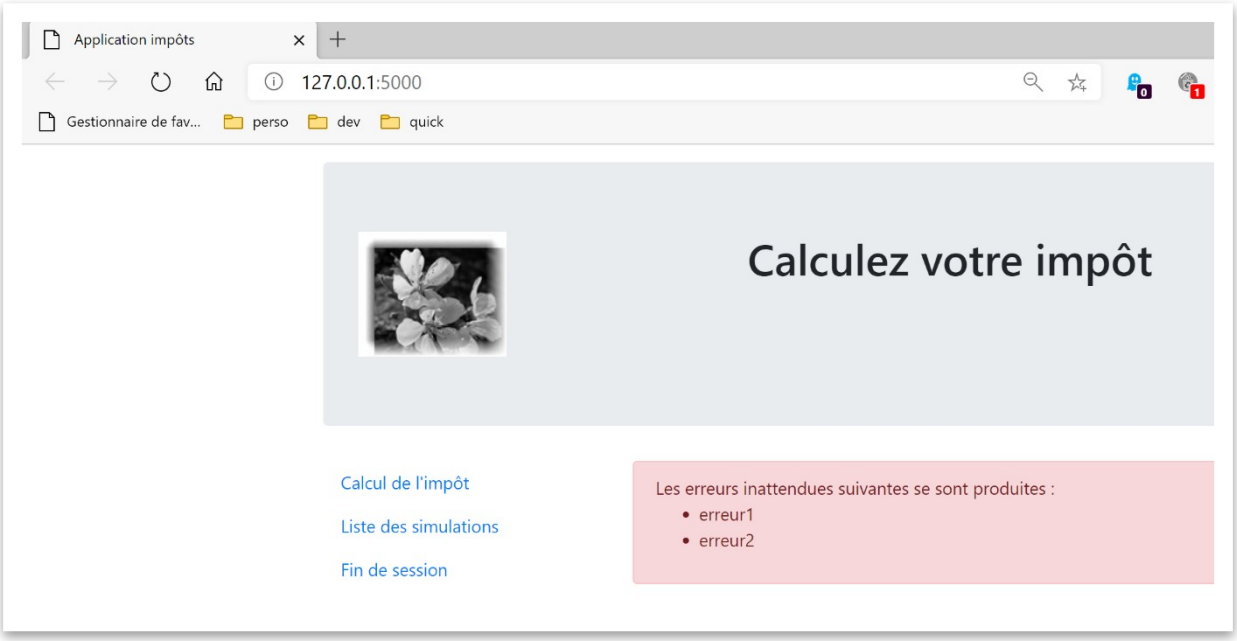



```
1.  from flask import Flask, render_template, make_response
2.
3.  # application Flask
4.  app = Flask(__name__, template_folder="../templates", static_folder="../static")
5.
6.  # Home URL
7.  @app.route('/')
8.  def index():
9.      # on encapsule les données de la pagé dans modèle
10.     modèle = {}
11.     # on construit une liste HTML des erreurs
12.     content = ""
13.     for erreur in ["erreur1", "erreur2"]:
14.         content += f"<li>{erreur}</li>"
15.     modèle["erreurs"] = content
16.     # options du menu
17.     modèle["optionsMenu"] = [
18.         {"text": "Calcul de l'impôt", "url": '/calculer-impot'},
19.         {"text": 'Liste des simulations', "url": '/lister-simulations'},
20.         {"text": 'Fin de session', "url": '/fin-session'}]
21.
22.     # affichage de la page
23.     return make_response(render_template("views/vue-erreurs.html", modèle=modèle))
24.
25. # main
26. if __name__ == '__main__':
27.     app.config.update(ENV="development", DEBUG=True)
28.     app.run()
```

Commentaires

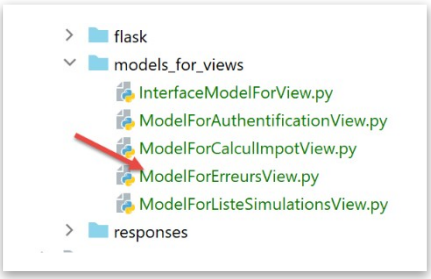
- lignes 11-15 : construction de la liste HTML des erreurs ;
- lignes 17-20 : le tableau des options de menu ;

Exécutons ce script. On obtient le résultat suivant :



On travaille sur cette vue jusqu'à ce que le résultat obtenu visuellement nous convienne. On peut ensuite passer à l'intégration de la vue dans l'application web en cours d'écriture.

32.8.3 Calcul du modèle de la vue



Une fois l'aspect visuel de la vue déterminé, on peut procéder au calcul du modèle de la vue en conditions réelles. Rappelons les codes d'état qui mènent à cette vue. On les trouve dans le fichier de configuration :

```
1.          # les vues HTML et leurs modèles dépendent de l'état rendu par le contrôleur
2.          "views": [
3.              {
4.                  # vue d'authentification
5.                  "états": [
6.                      # /init-session réussite
7.                      700,
8.                      # /fin-session
9.                      400,
10.                     # /authentifier-utilisateur échec
11.                     201
12.                 ],
13.                 "view_name": "views/vue-authentification.html",
14.                 "model_for_view": ModelForAuthentificationView()
15.             },
16.             {
17.                 # vue du calcul de l'impôt
18.                 "états": [
19.                     # /authentifier-utilisateur réussite
20.                     200,
21.                     # /calculer-impot réussite
22.                     300,
23.                     # /calculer-impot échec
24.                     301,
```

```
                    # /afficher-calcul-impot
                    800
                ],
                "view_name": "views/vue-calcul-impot.html",
                "model_for_view": ModelForCalculImpotView()
            },
            {
                # vue de la liste des simulations
                "états": [
                    # /lister-simulations
                    500,
                    # /supprimer-simulation
                    600
                ],
                "view_name": "views/vue-liste-simulations.html",
                "model_for_view": ModelForListeSimulationsView()
            }
        ],
        # vue des erreurs inattendues
        "view-erreurs": {
            "view_name": "views/vue-erreurs.html",
            "model_for_view": ModelForErreursView()
        },
```

Ce sont les codes d'état qui ne mènent pas à une vue HTML des lignes 3-41 qui font afficher la vue des erreurs inattendues.

Le calcul du modèle de la vue **[vue-erreurs.html]** est fait par la classe **[ModelForErreursView]** suivante :

```
from flask import Request
from werkzeug.local import LocalProxy

from InterfaceModelForView import InterfaceModelForView

class ModelForErreursView(InterfaceModelForView):

    def get_model_for_view(self, request: Request, session: LocalProxy, config: dict, résultat: dict) -> dict:
        # le modèle
        modèle = {}
        # les erreurs
        modèle["erreurs"] = ""
        for erreur in résultat['réponse']:
            modèle['erreurs'] += f"<li>{erreur}</li>"
        # menu
        modèle["optionsMenu"] = [
             {"text": "Calcul de l'impôt", "url": '/afficher-calcul-impot'},
             {"text": 'Liste des simulations', "url": '/lister-simulations'},
             {"text": 'Fin de session', "url": '/fin-session'}]
        # on rend le modèle
        return modèle
```

Commentaires

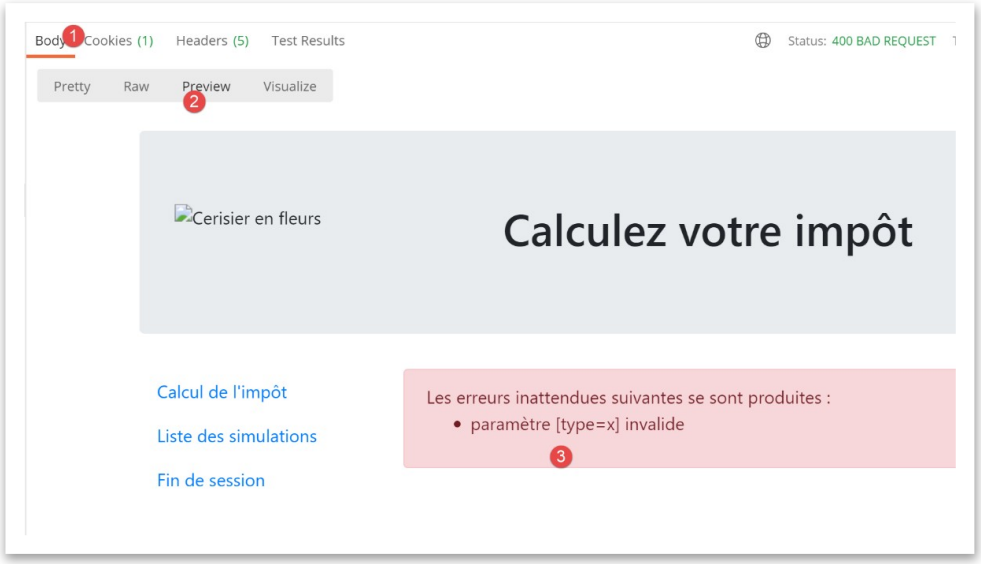
- lignes 11-14 : calcul du modèle **[modèle.erreurs]** utilisé par la vue **[vue-erreurs.html]** ;
- lignes 16-197 : calcul du modèle **[modèle.optionsMenu]** utilisé par le fragment **[v-menu.html]** ;

32.8.4 Tests [Postman]

On fait :

- l'action **[/init-session/html]** ;
- puis l'action **[/init-session/x]** ;

La réponse HTML est alors la suivante :



32.9 Implémentation des actions du menu de l'application

Nous allons ici traiter de l'implémentation des actions du menu. Rappelons la signification des liens que nous avons rencontrés

Vue	Lien	Cible	Rôle
Calcul de l'impôt	`[Liste des simulations]`	`[/lister-simulations]`	Demander la liste des simulations
	`[Fin de session]`	`[/fin-session]`	Demander la fin de la session
Liste des simulations	`[Calcul de l'impôt]`	`[/afficher-calcul-impot]`	Afficher la vue du calcul d'impôt
	`[Fin de session]`	`[/fin-session]`	Demander la fin de la session
Erreurs inattendues	`[Calcul de l'impôt]`	`[/afficher-calcul-impot]`	Afficher la vue du calcul d'impôt
	`[Liste des simulations]`	`[/lister-simulations]`	Demander la liste des simulations
	`[Fin de session]`	`[/fin-session]`	Demander la fin de la session

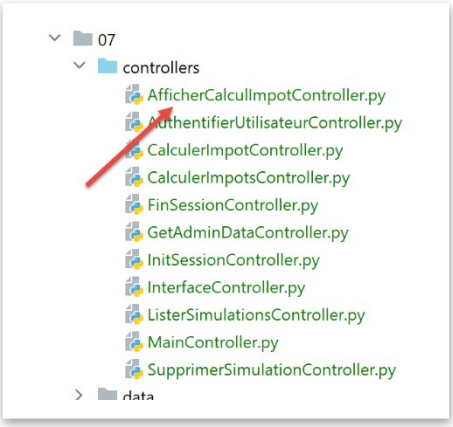
Il faut rappeler qu'un clic sur un lien provoque un GET vers la cible du lien. Les actions `[/lister-simulations, /fin-session]` ont été implémentées avec une opération GET, ce qui nous permet de les mettre comme cibles de liens. Lorsque l'action se fait par un POST, l'utilisation d'un lien n'est plus possible sauf à l'associer avec du Javascript.

32.9.1 L'action `[/afficher-calcul-impot]`

Des actions ci-dessus, il apparaît que l'action `[/afficher-calcul-impot]` n'a pas encore été implémentée. C'est une opération de navigation entre deux vues : le serveur jSON ou XML n'a aucune raison de l'implémenter car ils n'ont pas la notion de vue. C'est le serveur HTML qui introduit cette notion.

Il nous faut donc implémenter l'action `[/afficher-calcul-impot]`. Cela va nous permettre de réviser le mode opératoire de l'implémentation d'une action au sein du serveur.

Tout d'abord, il nous faut ajouter un nouveau contrôleur secondaire. Nous l'appellerons `[AfficherCalculImpotController]` :



Ce contrôleur doit être ajouté dans le fichier de configuration **[config]** :

```
1.      # les contrôleurs
2.      from AfficherCalculImpotController import AfficherCalculImpotController
3.      from AuthentifierUtilisateurController import AuthentifierUtilisateurController
4.      from CalculerImpotController import CalculerImpotController
5.      from CalculerImpotsController import CalculerImpotsController
6.      from FinSessionController import FinSessionController
7.      from GetAdminDataController import GetAdminDataController
8.      …
9.
10.
11.         # actions autorisées et leurs contrôleurs
12.         "controllers": {
13.             # initialisation d'une session de calcul
14.             "init-session": InitSessionController(),
15.             # authentification d'un utilisateur
16.             "authentifier-utilisateur": AuthentifierUtilisateurController(),
17.             # calcul de l'impôt en mode individuel
18.             "calculer-impot": CalculerImpotController(),
19.             # calcul de l'impôt en mode lots
20.             "calculer-impots": CalculerImpotsController(),
21.             # liste des simulations
22.             "lister-simulations": ListerSimulationsController(),
23.             # suppression d'une simulation
24.             "supprimer-simulation": SupprimerSimulationController(),
25.             # fin de la session de calcul
26.             "fin-session": FinSessionController(),
27.             # affichage de la vue de calcul de l'impôt
28.             "afficher-calcul-impot": AfficherCalculImpotController(),
29.             # obtention des données de l'administration fiscale
30.             "get-admindata": GetAdminDataController(),
31.             # main controller
32.             "main-controller": MainController()
33.         },
34. …
35.         # les vues HTML et leurs modèles dépendent de l'état rendu par le contrôleur
36.         "views": [
37.             {
38.                 # vue d'authentification
39.                 …
40.             },
41.             {
42.                 # vue du calcul de l'impôt
43.                 "états": [
44.                     # /authentifier-utilisateur réussite
45.                     200,
46.                     # /calculer-impot réussite
47.                     300,
48.                     # /calculer-impot échec
49.                     301,
50.                     # /afficher-calcul-impot
51.                     800
52.                 ],
53.                 "view_name": "views/vue-calcul-impot.html",
54.                 "model_for_view": ModelForCalculImpotView()
55.             },
```

```
56.             {…
57.             }
58.         ],
```

- ligne 2 : le nouveau contrôleur ;
- ligne 28 : la nouvelle action et son contrôleur ;
- ligne 51 : le nouveau contrôleur rendra le code d'état 800. Sur un changement de vues, il ne peut y avoir d'erreur. La vue affichée est la vue **[vue-calcul-impot.html]** que nous avons étudiée, expliquée et testée ;

Le contrôleur **[AfficherCalculImpotController]** sera le suivant :

```
1.  from flask_api import status
2.  from werkzeug.local import LocalProxy
3.
4.  from InterfaceController import InterfaceController
5.
6.  class AfficherCalculImpotController(InterfaceController):
7.
8.      def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
9.          # on récupère les éléments du path
10.         dummy, action = request.path.split('/')
11.
12.         # changement de vue - juste un code d'état à positionner
13.         return {"action": action, "état": 800, "réponse": ""}, status.HTTP_200_OK
```

Commentaires

- ligne 6 : comme les autres contrôleurs secondaires, le nouveau contrôleur implémente l'interface **[InterfaceController]** ;
- ligne 13 : les changements de vue sont simples à implémenter : il suffit de rendre un code d'état associé à la vue cible, ici le code 800 comme il a été vu plus haut ;

32.9.2 L'action [/fin-session]

L'action **[/fin-session]** est particulière. Elle n'amène pas directement à une vue mais à une redirection. Rappelons que les redirections sont configurées dans la configuration **[config]** de la façon suivante :

```
1.          # redirections
2.          "redirections": [
3.              {
4.                  "états": [
5.                      400,  # /fin-session réussi
6.                  ],
7.                  # redirection vers
8.                  "to": "/init-session/html",
9.              }
10.         ],
```

Il n'y a qu'une redirection dans l'application :

- lorsque le contrôleur rend le code d'état **[400]** (ligne 5), il faut rediriger le client vers l'URL **[http://machine:port/chemin/init-session/html]** (ligne 8) ;

Le code d'état **[400]** est le code rendu suite à une action **[/fin-session]** réussie. Pourquoi faut-il alors rediriger le client l'URL **[/init-session/html]** ? Parce que le code de l'action **[/fin-session]** supprime le type de la session présent dans la session web. On ne sait plus alors qu'on est dans une session html. Il faut le redire. On le fait à l'aide de l'action **[/init-session/html]**.

Les redirections HTML sont gérées par la classe **[HtmlResponse]** :

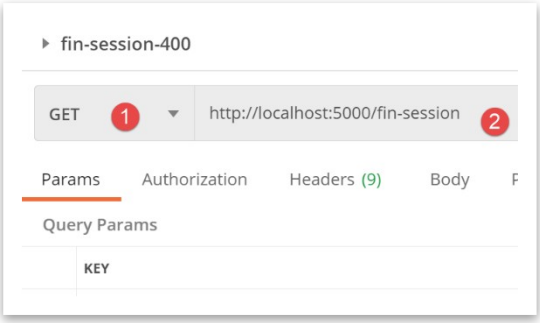
```
1.      def build_http_response(self, request: LocalProxy, session: LocalProxy, config: dict, status_code: int,
2.                              résultat: dict) -> (Response, int):
3.          # la réponse HTML dépend du code d'état rendu par le contrôleur
4.          état = résultat["état"]
5.
6.          # faut-il faire une redirection ?
7.          for redirection in config["redirections"]:
8.              # états nécessitant une redirection
9.              états = redirection["états"]
10.             if état in états:
11.                 # il faut faire une redirection
12.                 return redirect(f"{redirection['to']}"), status.HTTP_302_FOUND
13.
```

```
14.         # à un état, correspond une vue
15.         # on cherche celle-ci dans la liste des vues
16.   ..
```

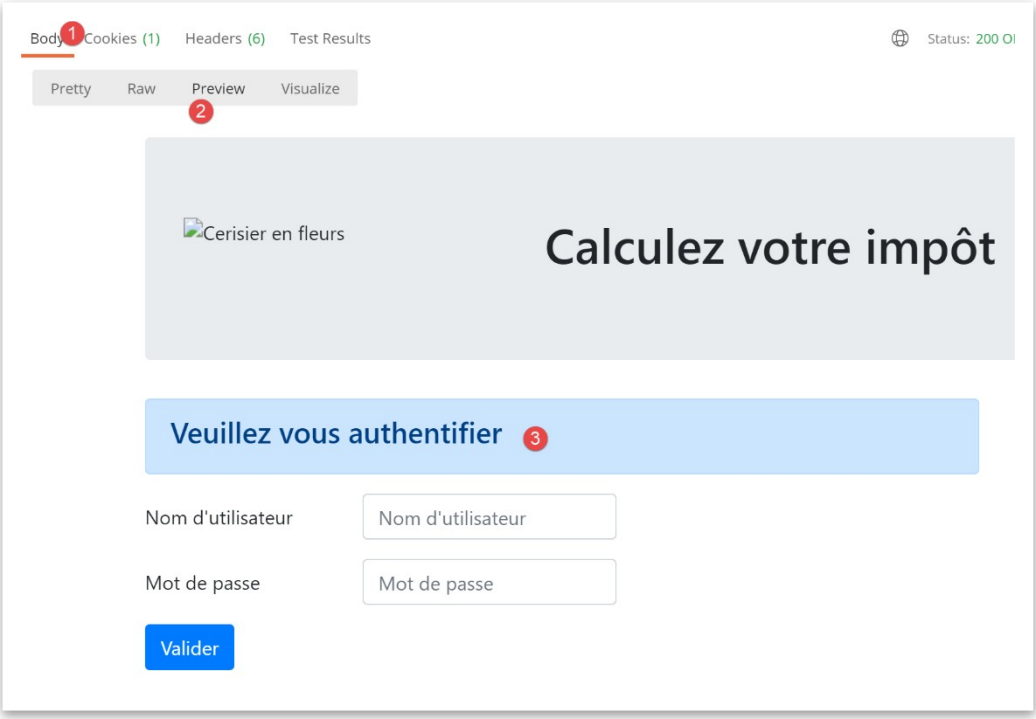
- les lignes 6-12 traitent les redirections ;
- ligne 7 : config**[‘redirections’]** est une liste de redirections. Chaque redirection est un dictionnaire avec les clés :
  - **[états]** : les états rendus par le contrôleur qui mènent à une redirection ;
  - **[to]** : l’adresse de redirection ;
- lignes 7-12 : on parcourt la liste des redirections ;
- ligne 9 : pour chaque redirection, on récupère les états qui y mènent ;
- ligne 10 : si l’état testé est dans cette liste, alors on fait la redirection, ligne 12 ;
- ligne 12 : on rappelle que la méthode **[build_http_response]** doit rendre un tuple à deux éléments :
  - **[response]** : la réponse HTTP à faire. Celle-ci est construite avec la fonction **[redirect]** dont le paramètre est l’adresse de redirection ;
  - **[status_code]** : le code de statut de la réponse HTTP, ici le code **[status.HTTP_302_FOUND]** qui indique au client qu’il doit se rediriger ;

Faisons un test **[Postman]**. On :

- initialise une session HTML **[init-session/html]** ;
- s’authentifie **[/authentifier-utilisateur]** ;
- termine la session **[/fin-session]** ;



La réponse du serveur est la suivante :



On a obtenu la vue d’authentification. C’est bien elle qu’on attendait. Maintenant, voyons comment elle a été obtenue. Passons sur la console **[Postman]** (Ctrl-Alt-C) :



- en **[1]**, l'action **[/fin-session]** ;
- en **[2-3]**, le code 302 de statut HTTP rendu par le serveur indique au client qu'il dit se rediriger ;
- en **[4]**, le client **[Postman]** suit la redirection ;

32.10 Tests de l'application HTML en conditions réelles

Le code a été écrit et chaque action testée avec **[Postman]**. Il nous reste à tester l'enchaînement des vues en situation réelle. Il nous faut un moyen d'initialiser la session HTML. On sait qu'il faut envoyer au serveur la requête **[/init-session/html]**. Ce n'est pas une URL très pratique. On préfèrerait démarrer avec l'URL **[/]**.

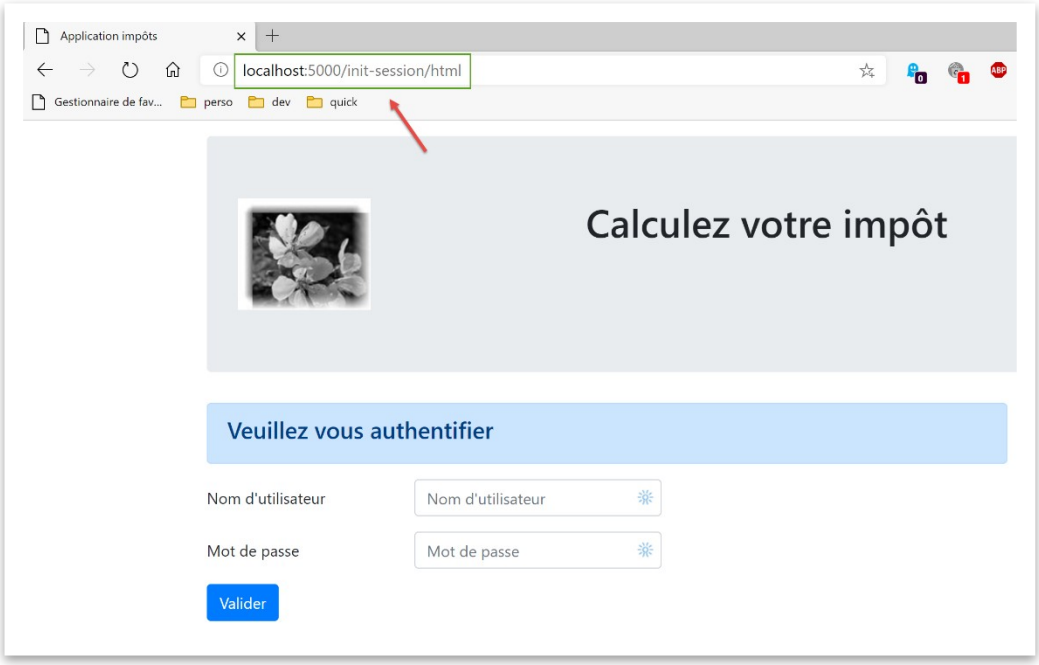
Nous avons écrit dans le script principal **[main]** la route suivante :

```
1.  from flask import request, Flask, session, url_for, redirect
2.  …
3.  …
4.  @app.route('/', methods=['GET'])
5.  def index() -> tuple:
6.      # redirection vers /init-session/html
7.      return redirect(url_for("init_session", type_response="html"), status.HTTP_302_FOUND)
8.  …
9.  # init-session
10. @app.route('/init-session/<string:type_response>', methods=['GET'])
11. def init_session(type_response: str) -> tuple:
12.     # on exécute le contrôleur associé à l'action
13.     return front_controller()
```

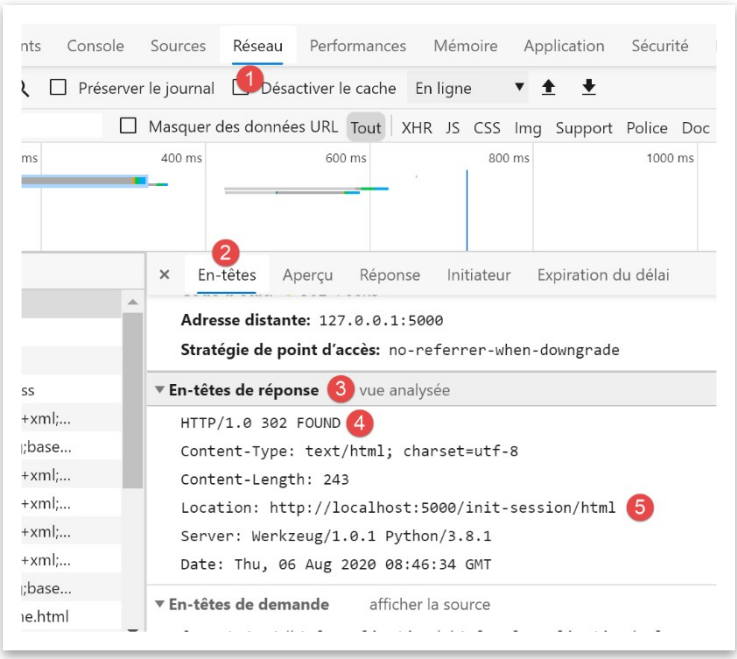
- lignes 4-7 : gestion de la route **[/]**. Le point d'entrée de l'application web sera l'URL**[/init-session/html]** (ligne 10). Aussi ligne 7, nous redirigeons le client vers cette URL :
  - la fonction **[url_for]** est importée ligne 1. Elle a ici deux paramètres (ligne 7) :
    - le 1er paramètre est le nom d'une des fonctions de routage, ici celle de la ligne 11. On voit que cette fonction attend un paramètre **[type_response]** qui est le type (json, xml, html) de réponse souhaité par le client ;
    - le 2ième paramètre reprend le nom du paramètre de la ligne 11, **[type_response]**, et lui donne une valeur. S'il y avait d'autres paramètres, on répéterait l'opération pour chacun d'eux ;
    - elle rend l'URL associée à la fonction désignée par les deux paramètres qui lui ont été donnés. Ici cela donnera l'URL de la ligne 10 où le paramètre est remplacé par sa valeur **[/init-session/html]** ;
  - la fonction **[redirect]** a été importée ligne 1. Elle a pour rôle d'envoyer un entête HTTP de redirection au client :
    - le 1er paramètre est l'URL vers laquelle le client doit être redirigé ;
    - le 2ième paramètre est le code de statut de la réponse HTTP faite au client. Le code **[status.HTTP_302_FOUND]** correspond à une redirection HTTP ;

Nous sommes prêts. Nous présentons maintenant quelques enchaînements de vues.

Dans notre navigateur, nous activons le suivi des requêtes (F12 sur Chrome, Firefox, Edge) et nous demandons l'URL de démarrage **[http://localhost:5000/]**. La réponse du serveur est la suivante :

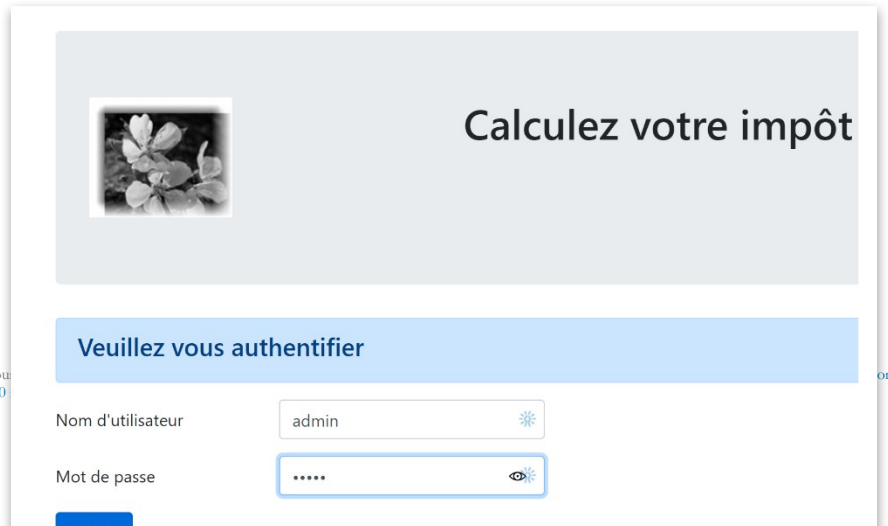


Si on regarde les échanges réseau qui ont eu lieu entre le client et le serveur :

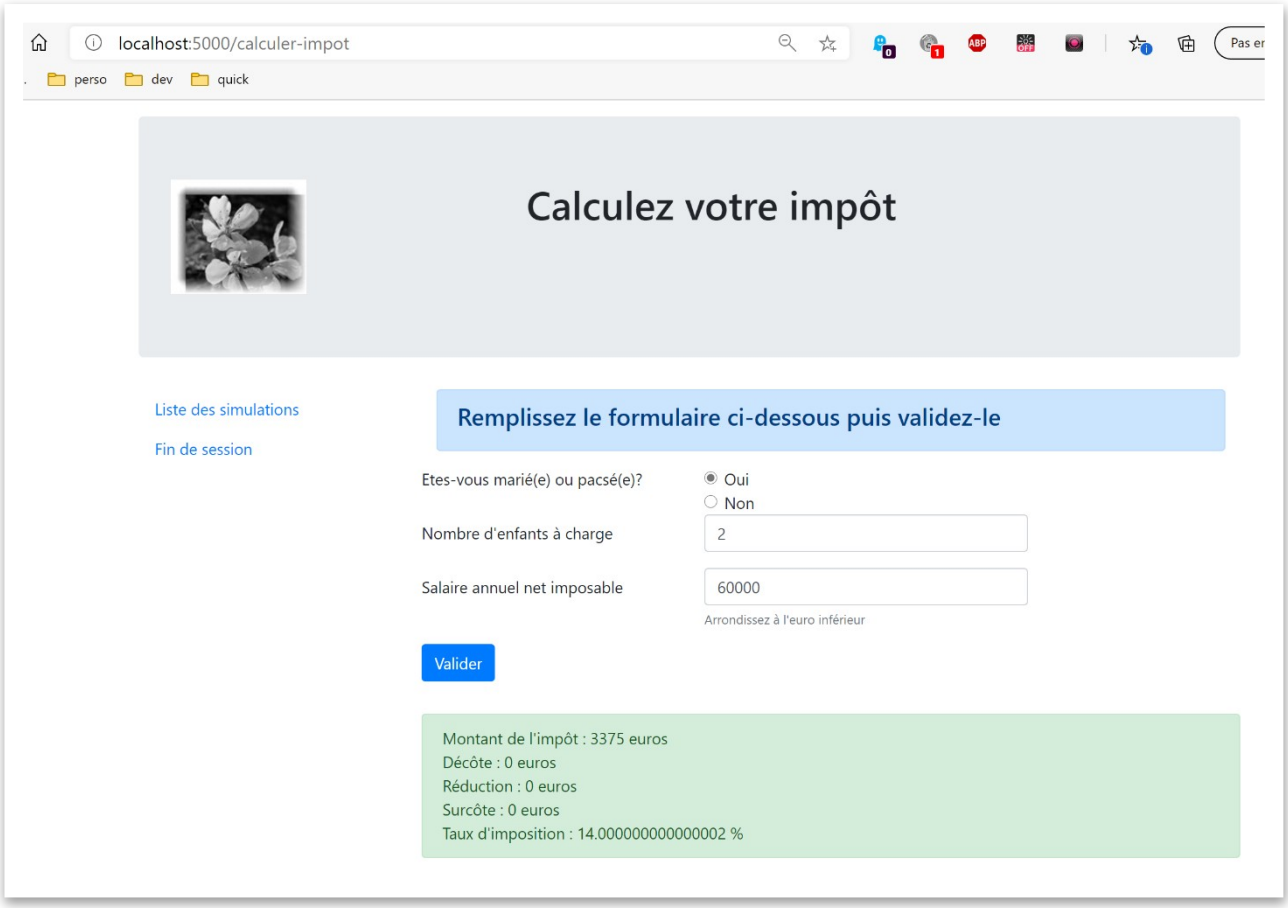


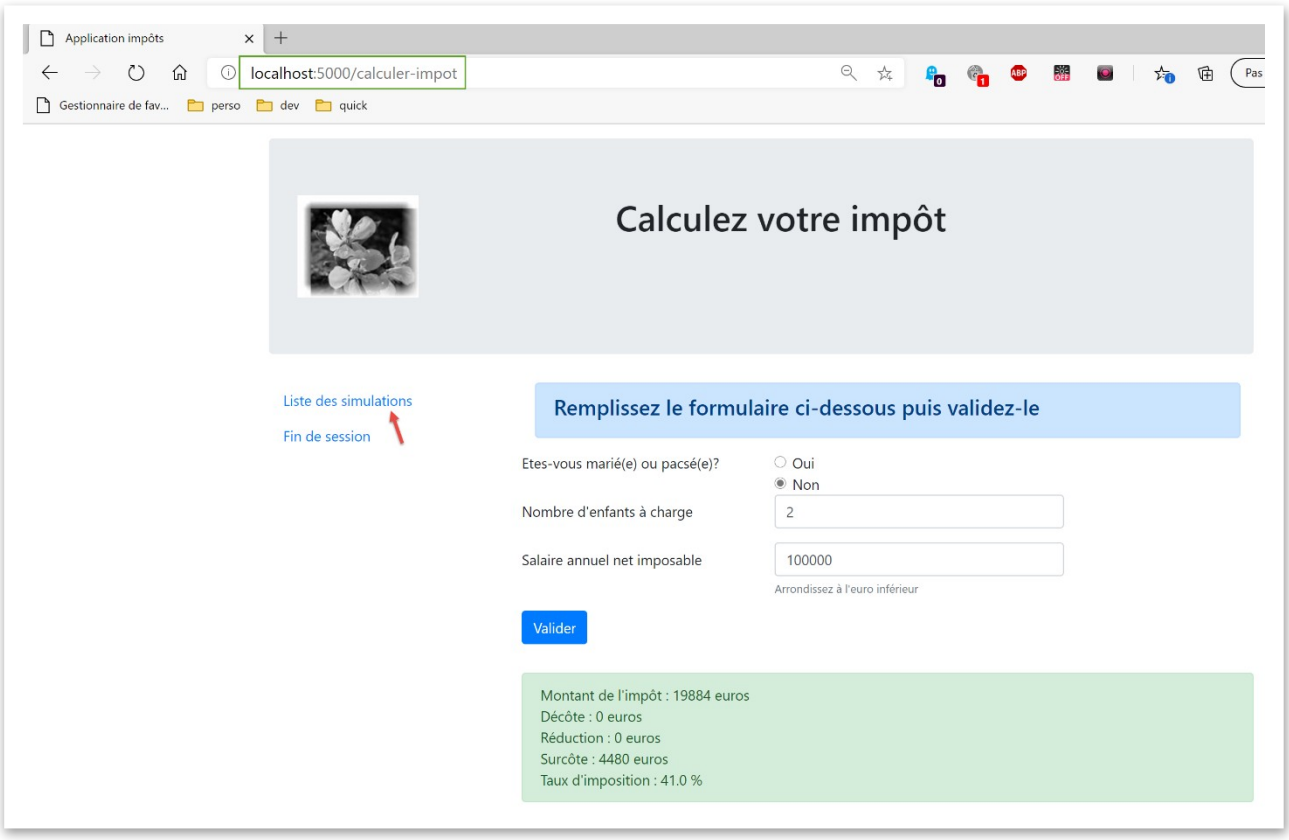
- on voit qu'en **`[4, 5]`**, le navigateur a reçu une demande de redirection vers l'URL **`[/init-session/html]`** ;

Remplissons le formulaire que nous avons reçu ;

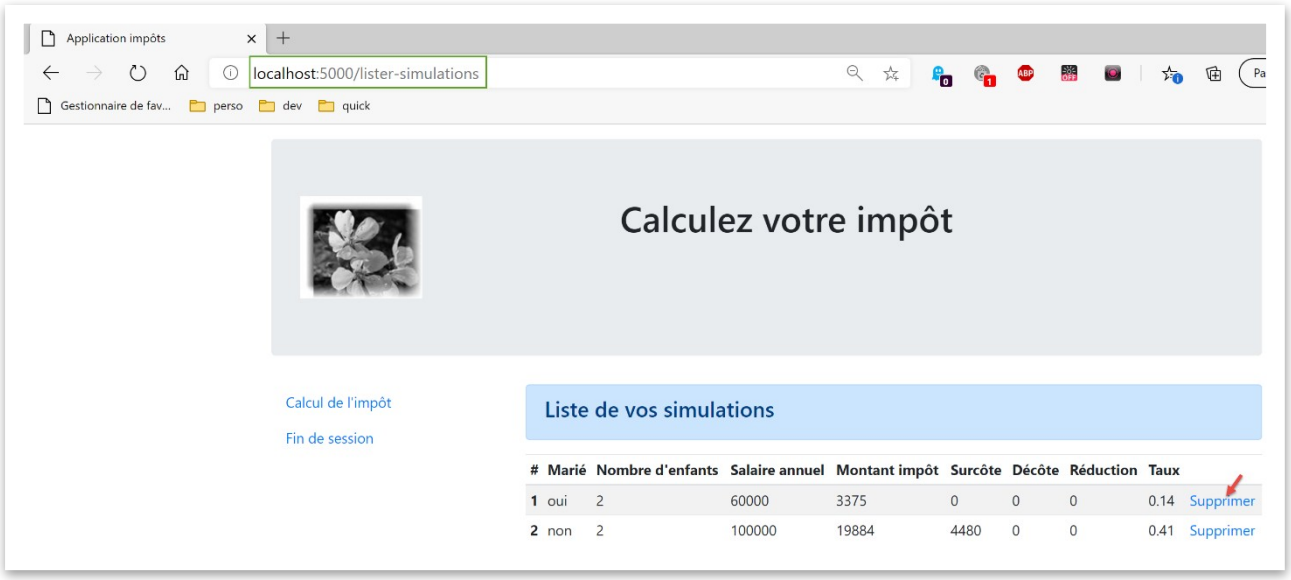


Puis faisons quelques simulations :

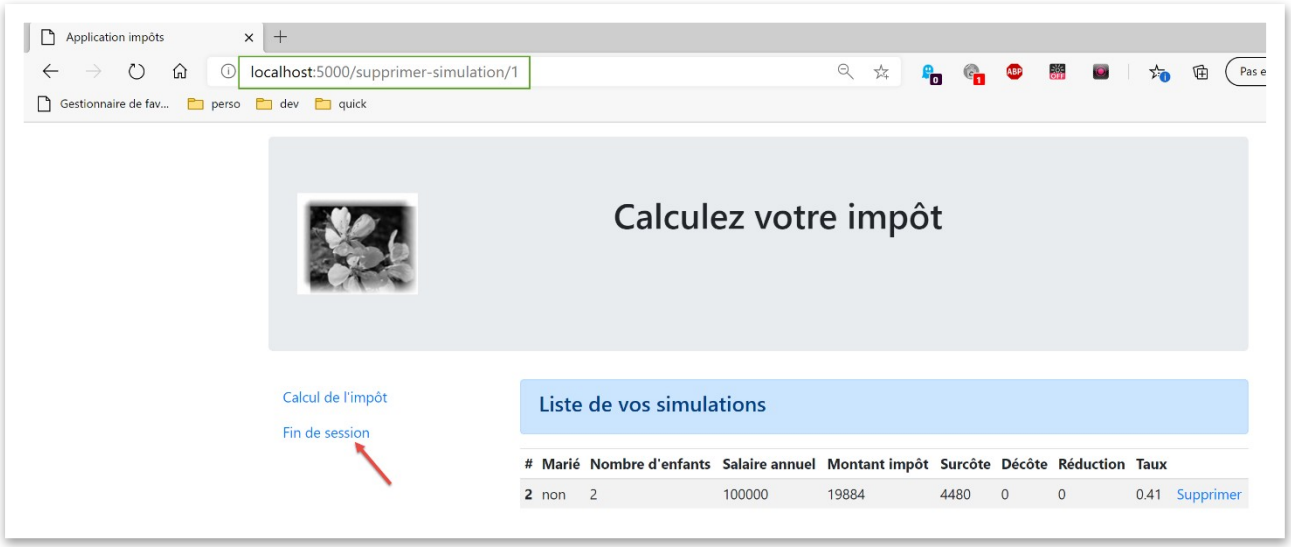




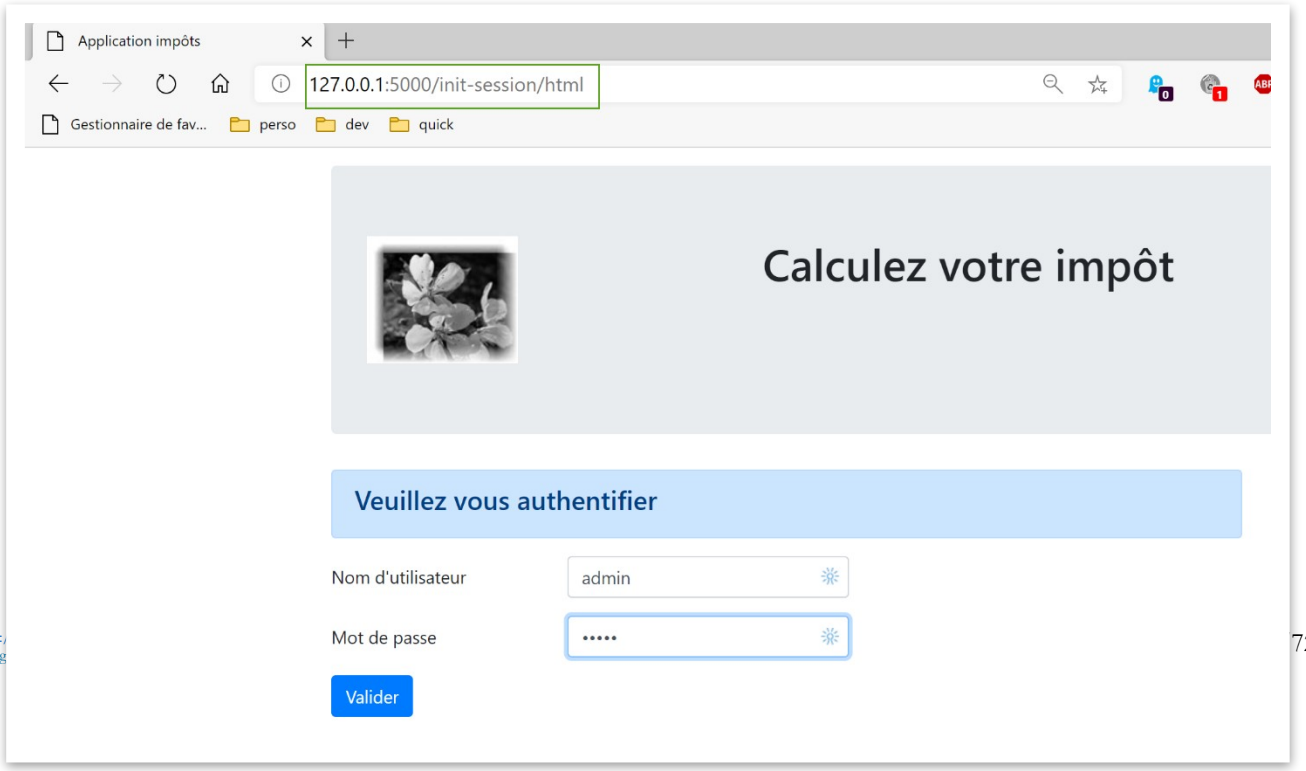
Demandons la liste des simulations :



Supprimons la 1re simulation :



Terminons la session :



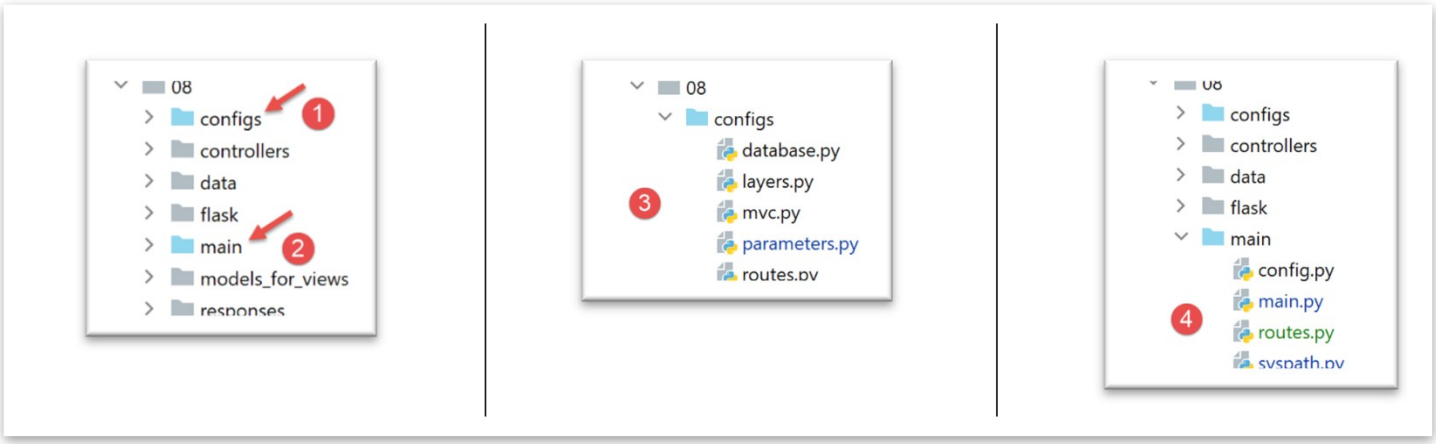
Le lecteur est invité à faire d'autres tests.

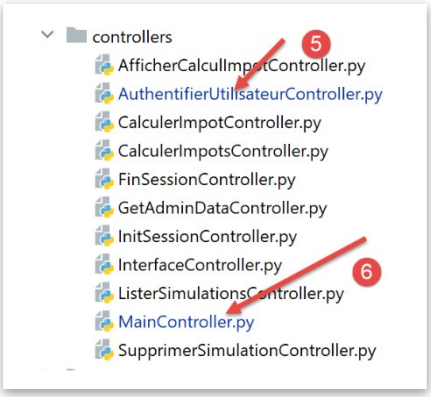
33 Exercice d'application : version 13

La version 13 modifie la version 12 sur les points suivants :

- on restructure certaines parties du code (refactoring) ;
- on gère la session différemment à l'aide du module `[flask_session]` ;
- on utilise des mots de passe cryptés dans le fichier de configuration ;

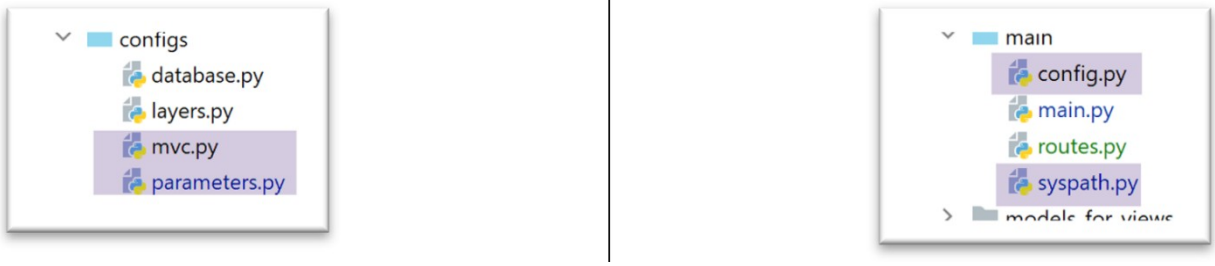
La version 13 est obtenue initialement par recopie de la version 12 :





- en `[1]`, la configuration va être refactorisée. On la sort notamment du dossier `[flask]` ;
- en `[2]`, le script principal va être refactorisé. On le sort également du dossier `[flask]` ;
- en `[3]`, la configuration va être éclatée sur plusieurs fichiers ;
- en `[4]` : on va simplifier le script principal `[main]` en déportant du code sur d'autres fichiers ;
- en `[5]`, le contrôleur d'authentification va être modifié puisque désormais les mots de passe des utilisateurs seront cryptés ;
- en `[6]`, le contrôleur principal va intégrer du code précédemment présent dans le script principal `[main]` ;

33.1 Refactorisation de la configuration de l'application



Trois nouveaux fichiers de configuration apparaissent :

- **[mvc]** : pour paramétrer l'architecture MVC de l'application ;
- **[parameters]** : qui va rassembler toutes les constantes de l'application ;
- **[syspath]** : qui configure le Python Path de l'application ;

Le fichier **[syspath.py]** est le suivant :

```
1.  def configure(config: dict) -> dict:
2.      import os
3.
4.      # dossier de ce fichier
5.      script_dir = os.path.dirname(os.path.abspath(__file__))
6.
7.      # chemin racine
8.      root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
9.
10.     # dépendances
11.     absolute_dependencies = [
12.         # dossiers du projet
13.         # BaseEntity, MyException
14.         f"{root_dir}/classes/02/entities",
15.         # InterfaceImpôtsDao, InterfaceImpôtsMétier, InterfaceImpôtsUi
16.         f"{root_dir}/impots/v04/interfaces",
17.         # AbstractImpôtsdao, ImpôtsConsole, ImpôtsMétier
18.         f"{root_dir}/impots/v04/services",
19.         # ImpotsDaoWithAdminDataInDatabase
20.         f"{root_dir}/impots/v05/services",
21.         # AdminData, ImpôtsError, TaxPayer
22.         f"{root_dir}/impots/v04/entities",
23.         # Constantes, tranches
24.         f"{root_dir}/impots/v05/entities",
25.         # Logger, SendAdminMail
26.         f"{root_dir}/impots/http-servers/02/utilities",
27.         # dossier du script principal
28.         script_dir,
29.         # configs [database, layers, parameters, controllers, views]
30.         f"{script_dir}/../configs",
31.         # contrôleurs
32.         f"{script_dir}/../controllers",
33.         # réponses HTTP
34.         f"{script_dir}/../responses",
35.         # modèles des vues
36.         f"{script_dir}/../models_for_views",
37.     ]
38.
39.     # on fixe le syspath
40.     from myutils import set_syspath
41.     set_syspath(absolute_dependencies)
42.
43.     # on rend la configuration
44.     return {
45.         "root_dir": root_dir,
46.         "script_dir": script_dir
47.     }
```

- le script **[syspath]** sert à configurer le Python Path de l'application (lignes 40-41) ;
- il rend deux informations utiles aux autres scripts de configuration (lignes 45-46) ;

Le script **[mvc]** configure l'architecture MVC de l'application web jSON / XML / HTML :

```
1.  def configure(config: dict) -> dict:
2.      # configuration de l'application MVC
3.
4.      # les contrôleurs
5.      from AfficherCalculImpotController import AfficherCalculImpotController
6.      from AuthentifierUtilisateurController import AuthentifierUtilisateurController
7.      from CalculerImpotController import CalculerImpotController
8.      from CalculerImpotsController import CalculerImpotsController
9.      from FinSessionController import FinSessionController
10.     from GetAdminDataController import GetAdminDataController
11.     from InitSessionController import InitSessionController
12.     from ListerSimulationsController import ListerSimulationsController
13.     from MainController import MainController
14.     from SupprimerSimulationController import SupprimerSimulationController
```

```
    # les réponses HTTP
    from HtmlResponse import HtmlResponse
    from JsonResponse import JsonResponse
    from XmlResponse import XmlResponse
    # les modèles des vues
    from ModelForAuthentificationView import ModelForAuthentificationView
    from ModelForCalculImpotView import ModelForCalculImpotView
    from ModelForErreursView import ModelForErreursView
    from ModelForListeSimulationsView import ModelForListeSimulationsView

    # actions autorisées et leurs contrôleurs
    controllers = {
        # initialisation d'une session de calcul
        "init-session": InitSessionController(),
        # authentification d'un utilisateur
        "authentifier-utilisateur": AuthentifierUtilisateurController(),
        # calcul de l'impôt en mode individuel
        "calculer-impot": CalculerImpotController(),
        # calcul de l'impôt en mode lots
        "calculer-impots": CalculerImpotsController(),
        # liste des simulations
        "lister-simulations": ListerSimulationsController(),
        # suppression d'une simulation
        "supprimer-simulation": SupprimerSimulationController(),
        # fin de la session de calcul
        "fin-session": FinSessionController(),
        # affichage de la vue de calcul de l'impôt
        "afficher-calcul-impot": AfficherCalculImpotController(),
        # obtention des données de l'administration fiscale
        "get-admindata": GetAdminDataController(),
        # main controller
        "main-controller": MainController()
    }
    # les différents types de réponse (json, xml, html)
    responses = {
        "json": JsonResponse(),
        "html": HtmlResponse(),
        "xml": XmlResponse()
    }
    # les vues HTML et leurs modèles dépendent de l'état rendu par le contrôleur
    views = [
        {
            # vue d'authentification
            "états": [
                700,    # /init-session - succès
                201,    # /authentifier-utilisateur échec
            ],
            "view_name": "views/vue-authentification.html",
            "model_for_view": ModelForAuthentificationView()
        },
        {
            # vue du calcul de l'impôt
            "états": [
                200,    # /authentifier-utilisateur réussite
                300,    # /calculer-impot réussite
                301,    # /calculer-impot échec
                800,    # /afficher-calcul-impot réussite
            ],
            "view_name": "views/vue-calcul-impot.html",
            "model_for_view": ModelForCalculImpotView()
        },
        {
            # vue de la liste des simulations
            "états": [
                500,    # /lister-simulations réussite
                600,    # /supprimer-simulation réussite
            ],
            "view_name": "views/vue-liste-simulations.html",
            "model_for_view": ModelForListeSimulationsView()
        },

    ]
    # vue des erreurs inattendues
    view_erreurs = {
        "view_name": "views/vue-erreurs.html",
        "model_for_view": ModelForErreursView()
    }
```

```
92.     # redirections
93.     redirections = [
94.         {
95.             "états": [
96.                 400,  # /fin-session réussite
97.             ],
98.             # redirection vers l'URL
99.             "to": "/init-session/html",
100.        }
101.    ]
102.
103.
104.    # on rend la configuration MVC
105.    return {
106.        # contrôleurs
107.        "controllers": controllers,
108.        # réponses HTTP
109.        "responses": responses,
110.        # vues et modèles
111.        "views": views,
112.        # liste des redirections
113.        "redirections": redirections,
114.        # vue des erreurs inattendues
115.        "view_erreurs": view_erreurs
116.    }
```

- lignes 1-101 : ce code est connu ;
- lignes 105-116 : on rend la configuration MVC de l'application ;

Le script **[parameters]** rassemble les constantes de l'application :

```
1.  def configure(config: dict) -> dict:
2.      # paramétrage de l'application
3.
4.      # script_dir
5.      script_dir = config['syspath']['script_dir']
6.
7.      # configuration de l'application
8.      parameters = {
9.          # utilisateurs autorisés à utiliser l'application
10.         "users": [
11.             {
12.                 "login": "admin",
13.                 "password": "$pbkdf2-sha256$29000$mPM.h3COkTIGYOzde68VIg$7LH5Q7rN/
    1hW.Xa.6rcmR6h52PntvVqd5.na7EtgQNw"
14.             }
15.          ],
16.         # fichier de logs
17.         "logsFilename": f"{script_dir}/../data/logs/logs.txt",
18.         # config serveur SMTP
19.         "adminMail": {
20.             # serveur SMTP
21.             "smtp-server": "localhost",
22.             # port du serveur SMTP
23.             "smtp-port": "25",
24.             # administrateur
25.             "from": "guest@localhost.com",
26.             "to": "guest@localhost.com",
27.             # sujet du mail
28.             "subject": "plantage du serveur de calcul d'impôts",
29.             # tls à True si le serveur SMTP requiert une autorisation, à False sinon
30.             "tls": False
31.         },
32.         # durée pause thread en secondes
33.         "sleep_time": 0,
34.         # serveur Redis
35.         "redis": {
36.             "host": "127.0.0.1",
37.             "port": 6379
38.         },
39.     }
40.
41.     # on rend le paramétrage de l'application
42.     return parameters
```

- ligne 13 : désormais les mots de passe des utilisateurs seront cryptés ;
- lignes 34-38 : la configuration d'un serveur **[Redis]** sur lequel nous reviendrons ;

Avec ces nouveaux fichiers de configuration, le script **[config]** devient le suivant :

```python
def configure(config: dict) -> dict:
    # configuration du syspath
    import syspath
    config['syspath'] = syspath.configure(config)

    # paramétrage de l'application
    import parameters
    config['parameters'] = parameters.configure(config)

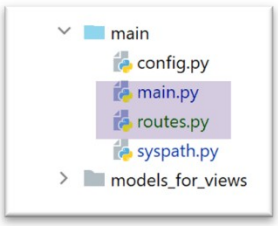
    # configuration de la base de données
    import database
    config["database"] = database.configure(config)

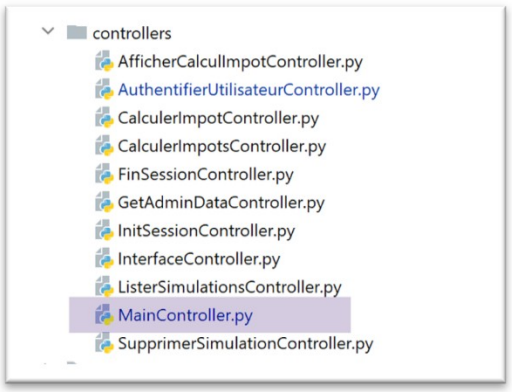
    # instanciation des couches de l'application
    import layers
    config['layers'] = layers.configure(config)

    # configuration MVC de la couche [web]
    import mvc
    config['mvc'] = mvc.configure(config)

    # on rend la configuration
    return config
```

33.2 Refactorisation du script principal [main]





Le script principal **[main.py]** se contente de mettre en route le serveur :

```python
# on attend un paramètre mysql ou pgres
import sys

syntaxe = f"{sys.argv[0]} mysql / pgres"
erreur = len(sys.argv) != 2
if not erreur:
    sgbd = sys.argv[1].lower()
    erreur = sgbd != "mysql" and sgbd != "pgres"
if erreur:
    print(f"syntaxe : {syntaxe}")
    sys.exit()

# on configure l'application
import config
config = config.configure({'sgbd': sgbd})

# dépendances
from SendAdminMail import SendAdminMail
from Logger import Logger
from ImpôtsError import ImpôtsError
import redis

# envoi d'un mail à l'administrateur
def send_adminmail(config: dict, message: str):
    # on envoie un mail à l'administrateur de l'application
    config_mail = config['parameters']['adminMail']
```

```
27.     config_mail["logger"] = config['logger']
28.     SendAdminMail.send(config_mail, message)
29.
30. # vérification du fichier de logs
31. logger = None
32. erreur = False
33. message_erreur = None
34. try:
35.     # logueur
36.     logger = Logger(config['parameters']['logsFilename'])
37. except BaseException as exception:
38.     # log console
39.     print(f"L'erreur suivante s'est produite : {exception}")
40.     # on note l'erreur
41.     erreur = True
42.     message_erreur = f"{exception}"
43. # on mémorise le logueur dans la config
44. config['logger'] = logger
45. # gestion de l'erreur
46. if erreur:
47.     # mail à l'administrateur
48.     send_adminmail(config, message_erreur)
49.     # fin de l'application
50.     sys.exit(1)
51.
52. # log de démarrage
53. log = "[serveur] démarrage du serveur"
54. logger.write(f"{log}\n")
55.
56. # on vérifie la disponibilité du serveur Redis
57. redis_client = redis.Redis(host=config["parameters"]["redis"]["host"],
58.                            port=config["parameters"]["redis"]["port"])
59. # on ping le serveur Redis
60. try:
61.     redis_client.ping()
62. except BaseException as exception:
63.     # Redis pas disponible
64.     log = f"[serveur] Le serveur Redis n'est pas disponible : {exception}"
65.     # console
66.     print(log)
67.     # log
68.     logger.write(f"{log}\n")
69.     # fin
70.     sys.exit(1)
71.
72. # on mémorise le client [redis] dans la config
73. config['redis_client'] = redis_client
74.
75. # récupération des données de l'administration fiscale
76. erreur = False
77. try:
78.     # admindata sera une donnée de portée application en lecture seule
79.     config["admindata"] = config["layers"]["dao"].get_admindata()
80.     # log de réussite
81.     logger.write("[serveur] connexion à la base de données réussie\n")
82. except ImpôtsError as ex:
83.     # on note l'erreur
84.     erreur = True
85.     # log d'erreur
86.     log = f"L'erreur suivante s'est produite : {ex}"
87.     # console
88.     print(log)
89.     # fichier de logs
90.     logger.write(f"{log}\n")
91.     # mail à l'administrateur
92.     send_adminmail(config, log)
93.
94. # le thread principal n'a plus besoin du logger
95. logger.close()
96.
97. # s'il y a eu erreur on s'arrête
98. if erreur:
99.     sys.exit(2)
100.
101.# import des routes de l'application web
102.import routes
103.routes.config=config
```

```
104.routes.execute(__name__)
```

- lignes 56-73 : on introduit un serveur Redis et son client ;
- lignes 102-104 : les routes de l'application ont été externalisées dans le script **[routes]** ;

33.2.1 Les modules [flask_session], et [redis]

Le serveur **[Redis]** va être utilisé pour mémoriser les sessions des utilisateurs. Nous allons utiliser le module **[flask_session]** pour gérer ces sessions. Ce module peut mémoriser les sessions des utilisateurs à plusieurs endroits. Redis est l'un de ceux-là et nous allons l'utiliser.

Le module **[flask_session]** doit être installé dans un terminal PyCharm :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\packages>pip install flask-session
2.  Collecting flask-session
3.    …
```

Pour dialoguer avec le serveur Redis, il nous faut un client Redis. Celui nous sera fourni par le module **[redis]** que nous installons également :

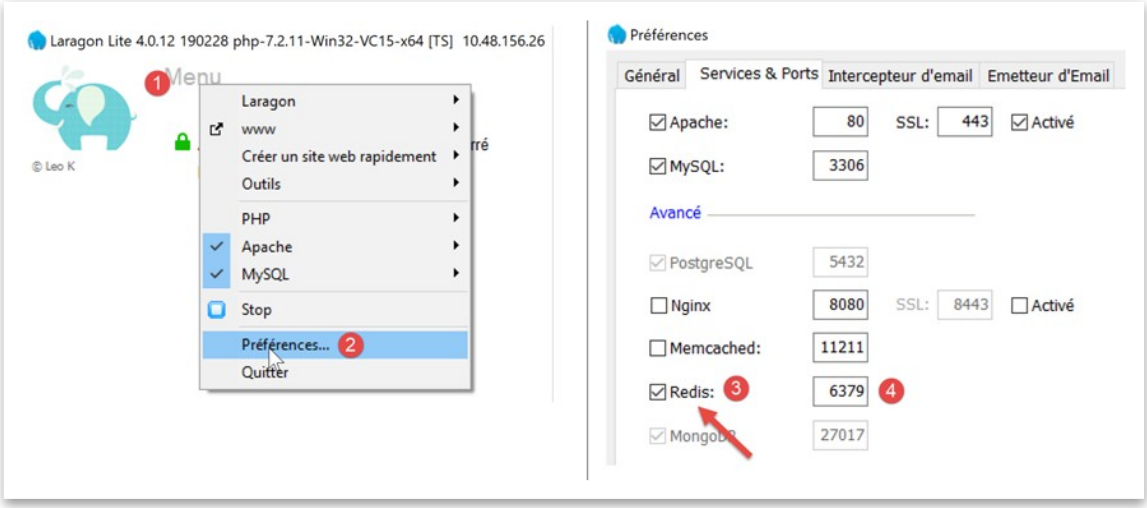
```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\packages>pip install redis
2.  Collecting redis
3.  …
```

33.2.2 Le serveur Redis

Le serveur Redis va servir à stocker les sessions des utilisateurs. Le module **[flask_session]** fonctionne de la façon suivante :

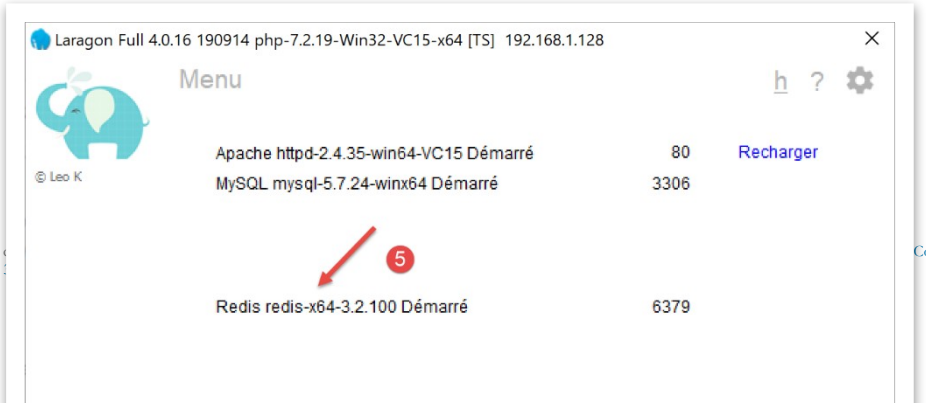
- chaque utilisateur a un identifiant de session et c'est ça qui est envoyé au client et uniquement ça. Le client ne reçoit de cookie de session qu'une seule fois, à l'issue de sa première requête. Ce cookie contient l'identifiant de session de l'utilisateur qui ne changera plus au fil des requêtes du client. C'est pourquoi le serveur n'a pas à renvoyer de nouveau cookie de session ;
- précédemment le contenu de la session était envoyé au client. Ce ne sera donc plus le cas. Le contenu de la session de l'utilisateur sera stocké sur le serveur Redis ;

Laragon vient avec un serveur Redis non activé par défaut. Il faut donc commencer par l'activer :

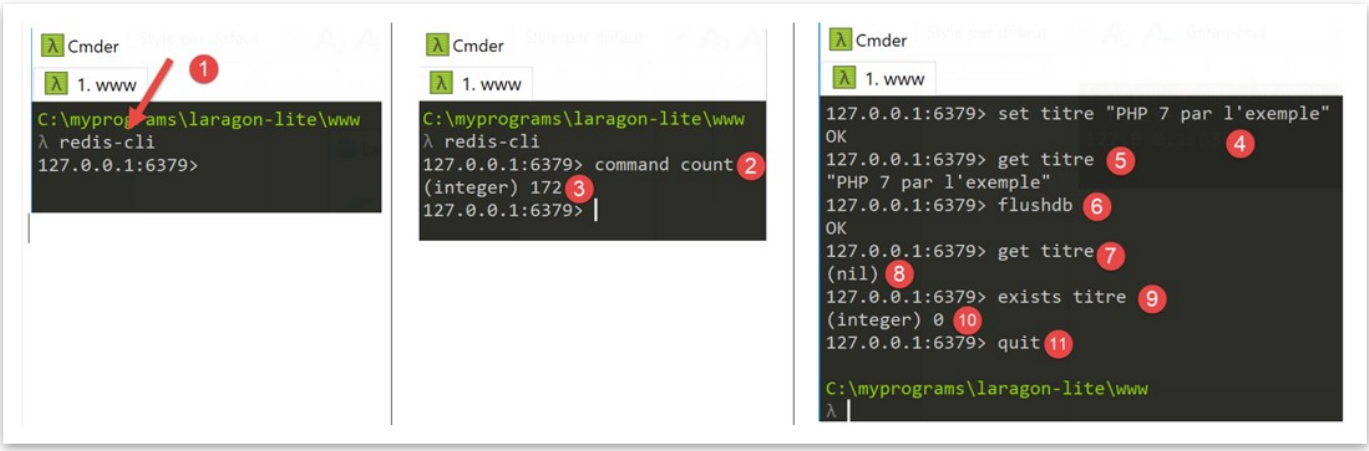


- en **[3]**, activer le serveur **[Redis]** ;
- en **[4]**, laisser le port **[6379]** que les clients Redis utilisent par défaut ;

Les services Laragon sont automatiquement relancés après activation de Redis :



Le serveur Redis peut être interrogé en mode commande. On ouvre un terminal Laragon **[6]** :



- en **[1]**, la commande **[redis-cli]** lance le client en mode commande du serveur Redis ;

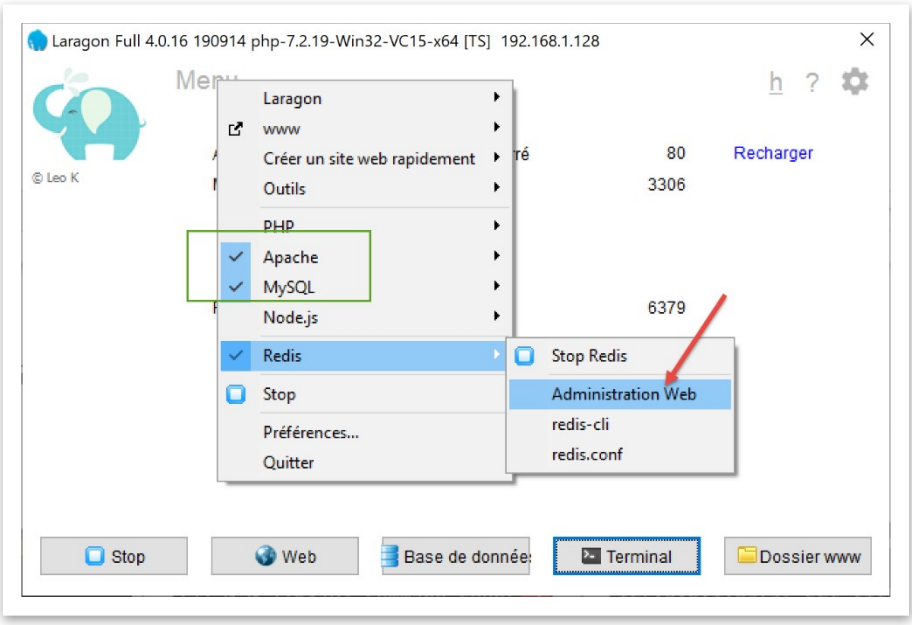
En juillet 2019, le client Redis peut utiliser 172 commandes pour dialoguer avec le serveur **[https://redis.io/commands#list]**. L'une d'elles **[command count] [2]**, affiche ce nombre **[3]**.

L'écriture dans **[Redis]** se fait avec la commande Redis **[set attribut valeur] [4]**. La valeur peut ensuite être récupérée avec la commande **[get attribut] [5]**.

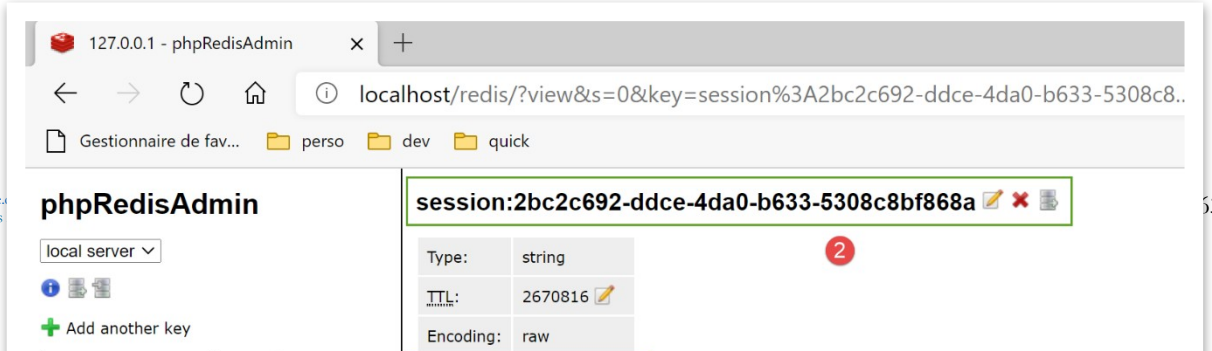
Il peut être nécessaire de vider la mémoire de Redis. Cela se fait avec la commande **[flushdb] [6]**. Ensuite si on demande la valeur de l'attribut **[titre] [7]**, on obtient une référence **[nil] [8]** indiquant que l'attribut n'a pas été trouvé. On peut également utiliser la commande **[exists] [9-10]** pour vérifier l'existence d'un attribut.

Pour quitter le client Redis, taper la commande **[quit] [11]**.

On peut également utiliser une interface web pour gérer les clés présentes dans le serveur Redis. Pour cela, il faut que le serveur Apache de Laragon soit lancé :



On obtient l'interface suivante :



- en **[1-4]**, l'une des sessions stockées sur le serveur Redis ;

33.2.3 Gestion du serveur Redis dans le script principal [main]

Le script **[main]** vérifie la présence du serveur Redis de la façon suivante :

```
1.  import redis
2.  …
3.  # on vérifie la disponibilité du serveur Redis
4.  redis_client = redis.Redis(host=config["parameters"]["redis"]["host"],
5.                             port=config["parameters"]["redis"]["port"])
6.  # on ping le serveur Redis
7.  try:
8.      redis_client.ping()
9.  except BaseException as exception:
10.     # Redis pas disponible
11.     log = f"[serveur] Le serveur Redis n'est pas disponible : {exception}"
12.     # console
13.     print(log)
14.     # log
15.     logger.write(f"{log}\n")
16.     # fin
17.     sys.exit(1)
18.
19. # on mémorise le client [redis] dans la config
20. config['redis_client'] = redis_client
```

- ligne 4 : le constructeur de la classe **[redis.Redis]** construit un client du serveur Redis. Les caractéristiques de celui-ci (adresse, port) sont trouvées dans le script **[parameters]** ;
- ligne 8 : la méthode **[ping]** permet de vérifier la présence du serveur Redis ;
- lignes 9-17 : si le ping n'aboutit pas, alors on logue l'erreur et on arrête le serveur ;
- ligne 20 : la référence du client Redis est mise dans la configuration ;

33.2.4 Gestion des routes dans le script principal [main]

La gestion des routes dans **[main]** se limite aux lignes suivantes :

```
1.  # import des routes de l'application web
2.  import routes
3.  routes.config=config
4.  routes.execute(__name__)
```

- ligne 1 : les routes ont été externalisées dans le module **[routes]** ;
- ligne 3 : les routes ont besoin de connaître la configuration de l'exécution ;
- ligne 4 : on lance l'application Flask en lui passant le nom du script exécuté (__main__) ;

Le script des routes est le suivant :

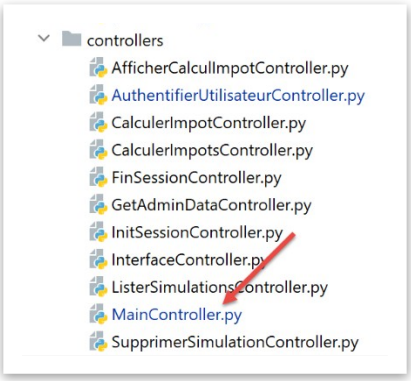
```
1.  # dépendances
2.  import os
3.
4.  from flask import Flask, redirect, request, session, url_for
5.  from flask_api import status
6.  from flask_session import Session
7.
8.  # application Flask
9.  app = Flask(__name__, template_folder="../flask/templates", static_folder="../flask/static")
10.
11. # configuration application
12. config = {}
13.
14. # le front controller
15. def front_controller() -> tuple:
16.     # on fait suivre la requête au contrôleur principal
17.     main_controller = config['mvc']['controllers']['main-controller']
18.     return main_controller.execute(request, session, config)
19.
20. @app.route('/', methods=['GET'])
21. def index() -> tuple:
22.     # redirection vers /init-session/html
23.     return redirect(url_for("init_session", type_response="html"), status.HTTP_302_FOUND)
```

```
24.
25. # init-session
26. @app.route('/init-session/<string:type_response>', methods=['GET'])
27. def init_session(type_response: str) -> tuple:
28.     # on exécute le contrôleur associé à l'action
29.     return front_controller()
30.
31. # authentifier-utilisateur
32. @app.route('/authentifier-utilisateur', methods=['POST'])
33. def authentifier_utilisateur() -> tuple:
34.     # on exécute le contrôleur associé à l'action
35.     return front_controller()
36.
37. # calculer-impot
38. @app.route('/calculer-impot', methods=['POST'])
39. def calculer_impot() -> tuple:
40.     # on exécute le contrôleur associé à l'action
41.     return front_controller()
42.
43. # calcul de l'impôt par lots
44. @app.route('/calculer-impots', methods=['POST'])
45. def calculer_impots():
46.     # on exécute le contrôleur associé à l'action
47.     return front_controller()
48.
49. # lister-simulations
50. @app.route('/lister-simulations', methods=['GET'])
51. def lister_simulations() -> tuple:
52.     # on exécute le contrôleur associé à l'action
53.     return front_controller()
54.
55. # supprimer-simulation
56. @app.route('/supprimer-simulation/<int:numero>', methods=['GET'])
57. def supprimer_simulation(numero: int) -> tuple:
58.     # on exécute le contrôleur associé à l'action
59.     return front_controller()
60.
61. # fin-session
62. @app.route('/fin-session', methods=['GET'])
63. def fin_session() -> tuple:
64.     # on exécute le contrôleur associé à l'action
65.     return front_controller()
66.
67. # afficher-calcul-impot
68. @app.route('/afficher-calcul-impot', methods=['GET'])
69. def afficher_calcul_impot() -> tuple:
70.     # on exécute le contrôleur associé à l'action
71.     return front_controller()
72.
73. # get-admindata
74. @app.route('/get-admindata', methods=['GET'])
75. def get_admindata() -> tuple:
76.     # on exécute le contrôleur associé à l'action
77.     return front_controller()
78.
79. def execute(name: str):
80.     # clé secrète de la session
81.     app.secret_key = os.urandom(12).hex()
82.     # Flask-Session
83.     app.config.update(SESSION_TYPE='redis',
84.                       SESSION_REDIS=config['redis_client'])
85.     Session(app)
86.     # cas où on lance l'application Flask via un script console
87.     if name == '__main__':
88.         app.config.update(ENV="development", DEBUG=True)
89.         app.run(threaded=True)
```

- ligne 9 : l'application Flask est instanciée ;
- ligne 12 : la configuration de l'application n'est pas encore connue à l'écriture du script. Elle n'est connue qu'au moment de son exécution ;
- lignes 20-77 : les routes de l'application telles qu'elles étaient définies dans la version précédente. Ca ne change pas ;
- lignes 14-18 : toutes les routes se contentent d'appeler la fonction **[front_controller]**. Nous avons dépouillé celle-ci de son code initial. Elle se contente maintenant d'appeler le contrôleur principal de l'application web ;
- lignes 79-89 : **[execute]** est la fonction appelée par le script **[main]** pour lancer l'application web ;
- ligne 81 : le module **[flask_session]** utilise la clé secrète de Flask ;

- lignes 82-84 : configuration du module **[flask_session]**. Celle-ci consiste à ajouter les clés **[SESSION_TYPE, SESSION_REDIS]** à la configuration **[app.config]** de l'application Flask **[app]** :
  - **[SESSION_TYPE]** : le type de la session. Il en existe plusieurs. Le type **[redis]** indique que **[flask_session]** utilise un serveur **[redis]** pour mémoriser les sessions des utilisateurs. A cause de cela, on doit définir la clé **[SESSION_REDIS]** qui doit être la référence d'un client Redis ;
- ligne 85 : la session **[Flask-Session]** est associée à l'application Flask ;
- lignes 86-89 : si le paramètre **[name]** de ligne 79 est la chaîne **[__main__]**, alors l'application Flask est lancée ;

33.3 Refactorisation du contrôleur principal



Le code autrefois dans la fonction **[front_controller]** du script **[main]** a été déplacé dans le contrôleur principal :

```
1.  # import des dépendances
2.  import threading
3.  import time
4.  from random import randint
5.
6.  from flask_api import status
7.  from werkzeug.local import LocalProxy
8.
9.  from InterfaceController import InterfaceController
10. from Logger import Logger
11. from SendAdminMail import SendAdminMail
12.
13. def send_adminmail(config: dict, message: str):
14.     # on envoie un mail à l'administrateur de l'application
15.     config_mail = config['parameters']['adminMail']
16.     config_mail["logger"] = config['logger']
17.     SendAdminMail.send(config_mail, message)
18.
19. # contrôleur principal de l'application
20. class MainController(InterfaceController):
21.     def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
22.         # on traite la requête
23.         logger = None
24.         action = None
25.         type_response1 = None
26.         try:
27.             # on récupère les éléments du path
28.             params = request.path.split('/')
29.
30.             # l'action est le 1er élément
31.             action = params[1]
32.
33.             # pas d'erreurs au départ
34.             erreur = False
35.
36.             # le type de session doit être connu avant certaines actions
37.             type_response1 = session.get('typeResponse')
38.             if type_response1 is None and action != "init-session":
39.                 # on note l'erreur
40.                 résultat = {"action": action, "état": 101,
41.                             "réponse": ["pas de session en cours. Commencer par action [init-session]"]}
42.                 erreur = True
43.
44.             # logger
45.             logger = Logger(config['parameters']['logsFilename'])
46.
47.             # on le mémorise dans une config associée au thread
48.             thread_config = {"logger": logger}
49.             thread_name = threading.current_thread().name
50.             config[thread_name] = {"config": thread_config}
51.
```

```
            # on logue la requête
            logger.write(f"[MainController] requête : {request}\n")

            # on interrompt le thread si cela a été demandé
            sleep_time = config['parameters']['sleep_time']
            if sleep_time != 0:
                # la pause est aléatoire pour que certains threads soient interrompus et d'autres pas
                aléa = randint(0, 1)
                if aléa == 1:
                    # log avant pause
                    logger.write(f"[front_controller] mis en pause du thread pendant {sleep_time} seconde(s)\n")
                    # pause
                    time.sleep(sleep_time)

            # pour certaines actions on doit être authentifié
            user = session.get('user')
            if not erreur and user is None and action not in ["init-session", "authentifier-utilisateur"]:
                # on note l'erreur
                résultat = {"action": action, "état": 101,
                            "réponse": [f"action [{action}] demandée par utilisateur non authentifié"]}
                erreur = True

            # y-a-t-il des erreurs ?
            if erreur:
                # la requête est invalide
                status_code = status.HTTP_400_BAD_REQUEST
            else:
                # on exécute le contrôleur associé à l'action
                controller = config['mvc']['controllers'][action]
                résultat, status_code = controller.execute(request, session, config)

        except BaseException as exception:
            # autres exceptions (inattendues)
            résultat = {"action": action, "état": 131, "réponse": [f"{exception}"]}
            status_code = status.HTTP_400_BAD_REQUEST

        finally:
            pass

        # on logue le résultat envoyé au client
        log = f"[MainController] {résultat}\n"
        logger.write(log)

        # y-a-t-il eu une erreur fatale ?
        if status_code == status.HTTP_500_INTERNAL_SERVER_ERROR:
            # on envoie un mail à l'administrateur de l'application
            send_adminmail(config, log)

        # on détermine le type souhaité pour la réponse
        type_response2 = session.get('typeResponse')
        if type_response2 is None and type_response1 is None:
            # le type de session n'a pas encore été établi - ce sera du jSON
            type_response = 'json'
        elif type_response2 is not None:
            # le type de la réponse est connu et dans la session
            type_response = type_response2
        else:
            # sinon on continue à utiliser type_response1
            type_response = type_response1

        # on construit la réponse à envoyer
        response_builder = config['mvc']['responses'][type_response]
        response, status_code = response_builder \
            .build_http_response(request, session, config, status_code, résultat)

        # on ferme le fichier de logs s'il a été ouvert
        if logger:
            logger.close()

        # on envoie la réponse HTTP
        return response, status_code
```

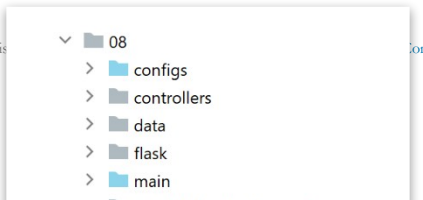
Tout ce code a été vu à un moment ou à un autre.

33.4 Gestion des mots de passe cryptés

Pour gérer des mots de passe cryptés nous allons utiliser le module **[passlib]** qu'on installe à partir d'un terminal Pycharm :

```
(venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\packages>pip install passlib
Collecting passlib
…
```

Voici un exemple de script cryptant le mot de passe qu'on lui passe en paramètre :



Le script **[create_hashed_password]** est le suivant (https://passlib.readthedocs.io/en/stable/) :

```
1.  import sys
2.
3.  # fonction de cryptage
4.  from passlib.hash import pbkdf2_sha256
5.
6.  # on attend le mot de passe à cryter
7.  syntaxe = f"{sys.argv[0]} password"
8.  erreur = len(sys.argv) != 2
9.  if erreur:
10.     print(f"syntaxe : {syntaxe}")
11.     sys.exit()
12. else:
13.     password = sys.argv[1]
14.
15. # on crypte le mot de passe
16. hash = pbkdf2_sha256.hash(password)
17. print(f"version cryptée de [{password}] = {hash}")
18.
19. # vérification
20. correct = pbkdf2_sha256.verify(password, hash)
21. print(correct)
```

- ligne 16 : on crypte le mot de passe passé en paramètre ;
- ligne 20 : on compare le mot de passe **[password]** passé en paramètre à sa version cryptée **[hash]**. La fonction **[verify]** crypte le mot de passe **[password]** et compare la chaîne cryptée obtenue à **[hash]**. Rend True si les deux chaînes sont égales ;

Le script ci-dessus nous permet d'avoir la version cryptée du mot de passe **[admin]** :

```
1.  C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\venv\Scripts\python.exe
    C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/http-servers/08/passlib/
    create_hashed_password.py admin
2.  version cryptée de [admin] = $pbkdf2-
    sha256$29000$fU9pTendO6c0ZoyR8r5Xqg$5ZXywIUnbMfN2hPnBaefiuqWjEbmAY.Lu06i4dwcnek
3.  True
```

Ligne 2, la valeur que nous mettons dans le script **[parameters]** :

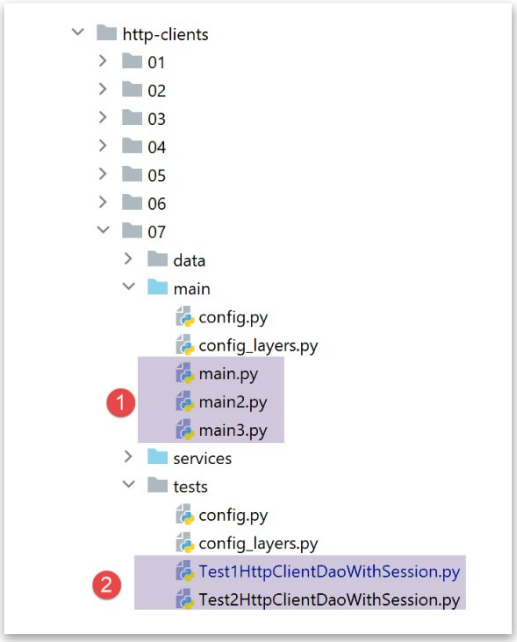
```
1.          "users": [
2.               {
3.                   "login": "admin",
4.                   "password": "$pbkdf2-
    sha256$29000$fU9pTendO6c0ZoyR8r5Xqg$5ZXywIUnbMfN2hPnBaefiuqWjEbmAY.Lu06i4dwcnek"
5.               }
6.           ],
```

Le contrôleur d'authentification **[AuthentifierUtilisateurController]** évolue de la façon suivante :

```
1.  from passlib.handlers.pbkdf2 import pbkdf2_sha256
2.  …
3.              # on vérifie la validité du couple (user, password)
4.              users = config['parameters']['users']
5.              i = 0
6.              nbusers = len(users)
7.              trouvé = False
8.              while not trouvé and i < nbusers:
9.                  trouvé = user == users[i]["login"] and pbkdf2_sha256.verify(password, users[i]["password"])
10.                 i += 1
11.             # trouvé ?
12.             if not trouvé:
13. …
```

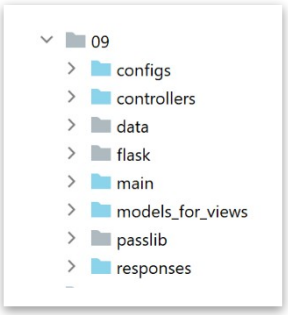
33.5 Tests

Outre les tests avec un navigateur, on peut également utiliser les clients du dossier **[http-servers/07]** écrits pour la version 12. Ils doivent 'marcher' également pour la version 13 :



- en **[1]**, les trois clients doivent fonctionner ;
- en **[2]**, les deux tests doivent fonctionner ;

34 Exercice d'application : version 14



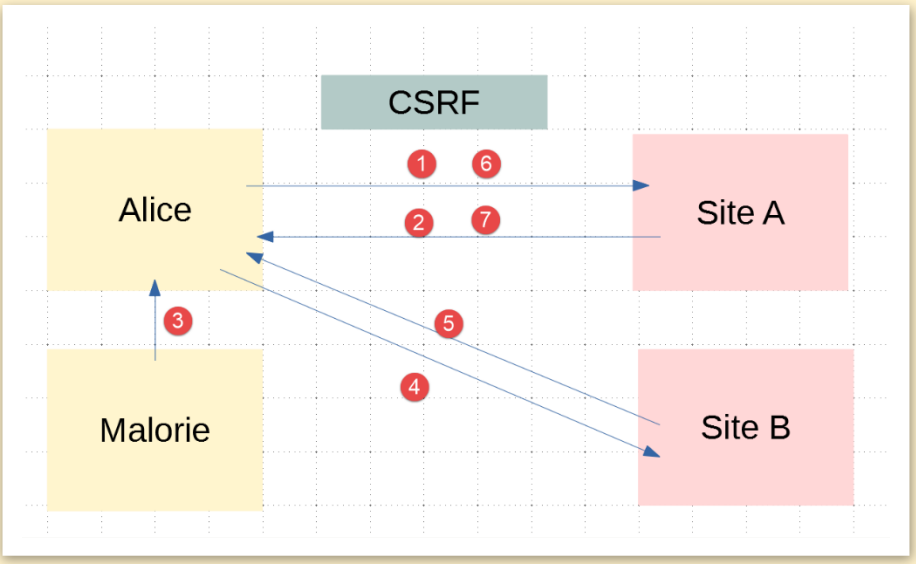
Le dossier **[http-servers/09]** de la version 14 est obtenu par recopie du dossier **[http-servers/08]** de la version 13.

34.1 Introduction

CSRF (Cross Site Request Forgery) est une technique de vol de session. Elle est expliquée ainsi dans Wikipedia (https://fr.wikipedia.org/wiki/Cross-site_request_forgery):

Supposons qu'Alice soit l'administratrice d'un forum et qu'elle soit connectée à celui-ci par un système de sessions. Malorie est un membre de ce même forum, elle veut supprimer un des messages du forum. Comme elle n'a pas les droits nécessaires avec son compte, elle utilise celui d'Alice grâce à une attaque de type CSRF.

1. Malorie arrive à connaitre le lien qui permet de supprimer le message en question.
2. Malorie envoie un message à Alice contenant une pseudo-image à afficher (qui est en fait un script). L'URL de l'image est le lien vers le script permettant de supprimer le message désiré.
3. Alice doit avoir une session ouverte dans son navigateur pour le site visé par Malorie. C'est une condition requise pour que l'attaque réussisse de façon silencieuse sans requérir une demande d'authentification qui alerterait Alice. Cette session doit disposer des droits requis pour exécuter la requête destructrice de Malorie. Il n'est pas nécessaire qu'un onglet du navigateur soit ouvert sur le site cible ni même que le navigateur soit démarré. Il suffit que la session soit active.
4. Alice lit le message de Malorie, son navigateur utilise la session ouverte d'Alice et ne demande pas d'authentification interactive. Il tente de récupérer le contenu de l'image. En faisant cela, le navigateur actionne le lien et supprime le message, il récupère une page web texte comme contenu pour l'image. Ne reconnaissant pas le type d'image associé, il n'affiche pas



d'image et Alice ne sait pas que Malorie vient de lui faire supprimer un message contre son gré.

Même expliqué comme ça, la technique du CSRF est difficile à comprendre. Faisons un schéma :

- en **[1-2]**, Alice communique avec le forum (Site A). Ce forum maintient une session pour chaque utilisateur. Le navigateur d'Alice stocke localement ce cookie de session et le renvoie à chaque fois qu'il fait une nouvelle requête au site A ;
- en **[3]**, Malorie envoie un message à Alice. Celle-ci le lit avec son navigateur. Le message lu est au format HTML et contient un lien vers une image du site B. En fait ce lien est un lien vers un script Javascript qui s'exécute une fois arrivé sur le navigateur d'Alice ;

- ce script Javascript réalise alors une requête vers le site A. Le navigateur d'Alice envoie alors automatiquement la requête avec le cookie de session stocké localement. C'est ici que se produit l'attaque : Malorie a réussi à interroger le site A avec les droits (session) de Alice. Ensuite, peu importe ce qui se passe, l'attaque a eu lieu ;

Pour contrer ce type d'attaque, le site A peut procéder de la façon suivante :

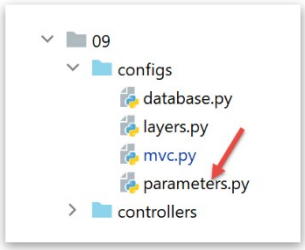
- à chaque échange **[1-2]** avec Alice, le site A envoie une clé, appelée par la suite token (jeton) CSRF, qu'Alice doit lui renvoyer lors de la requête suivante. Ainsi Alice doit envoyer à chaque requête deux informations :
  - le cookie de session ;
  - le token CSRF reçu lors de la réponse à sa dernière requête au site A ;

  La protection est là : si le navigateur renvoie automatiquement au site A le cookie de session, il ne le fait pas pour le token CSRF. Pour cette raison, l'échange 6-7 fait par le script d'attaque sera refusé car la requête 6 n'aura pas envoyé le jeton CSRF ;

Le site A peut envoyer à Alice le jeton CSRF de diverses façons pour une application HTML :

- il peut à chaque requête envoyer une page HTML où tous les liens auront le jeton CSRF, par exemple **[http://siteA/chemin/csrf_token]**. Lors de la requête suivante, Alice cliquant sur l'un de ces liens, le site A n'aura qu'à récupérer le jeton CSRF dans l'URL de la requête et vérifier qu'il est correct. C'est ce qui sera fait ici ;
- il peut, pour les pages HTML contenant un formulaire, envoyer celui-ci avec un champ caché **[input type='hidden']** contenant le jeton CSRF. Celui-ci sera alors posté automatiquement avec le formulaire lorsqu'Alice validera la page. Le site A récupèrera le jeton CSRF dans le corps (body) de la requête ;
- d'autres techniques sont envisageables ;

34.2 Configuration



Nous introduisons dans la configuration **[parameters]** de l'application deux booléens :

- **[with_redissession]** : à True, l'application utilise une session Redis. A False, l'application utilise une session Flask normale ;
- **[with_csrftoken]** : à True, les URL de l'application contiennent un jeton CSRF ;

```
        # durée pause thread en secondes
        "sleep_time": 0,
        # serveur Redis
        "with_redissession": True,
        "redis": {
            "host": "127.0.0.1",
            "port": 6379
        },
        # token csrf
        "with_csrftoken": False,
```

34.3 Implémentation CSRF

Nous allons faire en sorte que lorsque :

```
config['parameters']['with_csrftoken']
```

vaut **[True]**, l'application envoie au navigateur client des pages web dont les liens contiendront un jeton CSRF.

34.3.1 Le module [flask_wtf]

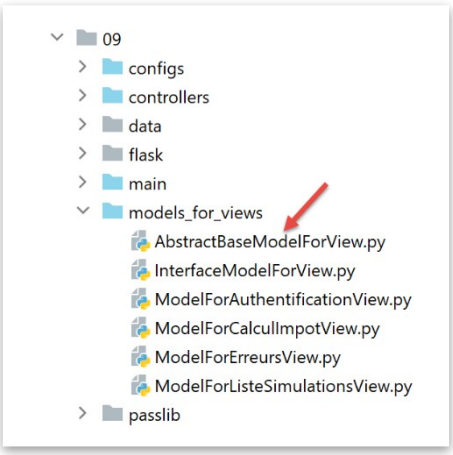
L'implémentation du jeton CSRF sera faite avec le module **[flask_wtf]** que nous installons dans un terminal PyCharm :

```
(venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\packages>pip install flask_wtf
```

```
2.  Collecting flask_wtf
3.  …
```

34.3.2 Les modèles des vues

Nous introduisons une nouvelle classe dans les modèles :



La classe **[AbstractBaseModelForView]** est la suivante :

```
1.  from abc import abstractmethod
2.
3.  from flask import Request
4.  from flask_wtf.csrf import generate_csrf
5.  from werkzeug.local import LocalProxy
6.
7.  from InterfaceModelForView import InterfaceModelForView
8.
9.  class AbstractBaseModelForView(InterfaceModelForView):
10.
11.     @abstractmethod
12.     def get_model_for_view(self, request: Request, session: LocalProxy, config: dict, résultat: dict) -> dict:
13.         pass
14.
15.     def get_csrftoken(self, config: dict):
16.         # csrf_token
17.         if config['parameters']['with_csrftoken']:
18.             return f"/{generate_csrf()}"
19.         else:
20.             return ""
```

- ligne 9 : la classe **[AbstractBaseModelForView]** implémente l'interface **[InterfaceModelForView]** implémentée par les classes des modèles ;
- lignes 11-13 : la méthode **[get_model_for_view]** n'est pas implémentée ;
- lignes 15-20 : la méthode **[get_csrftoken]** génère le jeton CSRF si l'application a été configurée pour les utiliser. Selon les cas, la fonction rend un jeton précédé du signe / sinon une chaîne vide. La fonction **[generate_csrf]** a la particularité de toujours générer la même valeur pour une requête client donnée. Le traitement d'une requête implique l'exécution de différentes fonctions. Utiliser **[generate_csrf]** dans celles-ci génère toujours la même valeur. Lors de la requête suivante, en revanche, un nouveau jeton CSRF est généré ;

Tous les modèles M de vue V incluront le jeton CSRF de la façon suivante :

```
1.  class ModelForAuthentificationView(AbstractBaseModelForView):
2.
3.      def get_model_for_view(self, request: Request, session: LocalProxy, config: dict, résultat: dict) -> dict:
4.          # on encapsule les données de la pagé dans modèle
5.          modèle = {}
6.          …
7.
8.          # jeton csrf
9.          modèle['csrf_token'] = super().get_csrftoken(config)
10.
11.         # on rend le modèle
12.         return modèle
```

- chaque classe de modèle étend la classe de base **[AbstractBaseModelForView]** ;
- ligne 8 : le jeton CSRF est demandé à la classe parent. On obtient soit la chaîne vide, soit une chaîne du genre **[/Ijk4NjQ2ZDdjZjI0ZDJiYTVjZTZjYmFhZGNjMjE3Y2U5M2I3ODI0NzYi.Xy5Okg.n-kSR_nslkndfT7AFVy2UDtdb8c]** ;

34.3.3 Les vues

De ce qu'on vient de voir, toutes les vues V auront dans leur modèle M le jeton CSRF. Elles pourront donc l'utiliser dans les liens qu'elles contiennent. Prenons quelques exemples :

Le fragment d'authentification `[v_authentification.html]`

```
1.  <!-- formulaire HTML - on poste ses valeurs avec l'action [authentifier-utilisateur] -->
2.  <form method="post" action="/authentifier-utilisateur{{modèle.csrf_token}}">
3.
4.      <!-- titre -->
5.      <div class="alert alert-primary" role="alert">
6.          <h4>Veuillez vous authentifier</h4>
7.      </div>
8.  …
9.
10. </form>
```

- ligne 2 : d'après ce qui vient d'être vu, l'URL de l'attribut **[action]** sera :

```
[/authentifier-utilisateur/Ijk4NjQ2ZDdjZjI0ZDJiYTVjZTZjYmFhZGNjMjE3Y2U5M2I3ODI0NzYi.Xy5Okg.n-
kSR_nslkndfT7AFVy2UDtdb8c]
```

ou

```
[/authentifier-utilisateur]
```

selon que l'application a été configurée pour utiliser ou non des jetons CSRF ;

Le fragment de calcul de l'impôt `[v-calcul-impot.html]`

```
1.  <!-- formulaire HTML posté -->
2.  <form method="post" action="/calculer-impot{{modèle.csrf_token}}">
3.      <!-- message sur 12 colonnes sur fond bleu -->
4.      <div class="col-md-12">
5.          <div class="alert alert-primary" role="alert">
6.              <h4>Remplissez le formulaire ci-dessous puis validez-le</h4>
7.          </div>
8.      </div>
9.      …
10. </form>
```

Le fragment des simulations `[v-liste-simulations.html]`

```
1.  {% if modèle.simulations is undefined or modèle.simulations|length==0 %}
2.  <!-- message sur fond bleu -->
3.  <div class="alert alert-primary" role="alert">
4.      <h4>Votre liste de simulations est vide</h4>
5.  </div>
6.  {% endif %}
7.
8.  {% if modèle.simulations is defined and modèle.simulations|length!=0 %}
9.  <!-- message sur fond bleu -->
10. <div class="alert alert-primary" role="alert">
11.     <h4>Liste de vos simulations</h4>
12. </div>
13.
14. <!-- tableau des simulations -->
15. <table class="table table-sm table-hover table-striped">
16.     …
17.     <!-- corps du tableau (données affichées) -->
18.     <tbody>
19.     <!-- on affiche chaque simulation en parcourant le tableau des simulations -->
20.     {% for simulation in modèle.simulations %}
21.
22.     <!-- affichage d'une ligne du tableau avec 6 colonnes - balise <tr> -->
23.     <!-- colonne 1 : entête ligne (n° simulation) - balise <th scope='row' -->
24.     <!-- colonne 2 : valeur paramètre [marié] - balise <td> -->
25.     <!-- colonne 3 : valeur paramètre [enfants] - balise <td> -->
26.     <!-- colonne 4 : valeur paramètre [salaire] - balise <td> -->
27.     <!-- colonne 5 : valeur paramètre [impôt] (de l'impôt) - balise <td> -->
28.     <!-- colonne 6 : valeur paramètre [surcôte] - balise <td> -->
29.     <!-- colonne 7 : valeur paramètre [décôte] - balise <td> -->
```

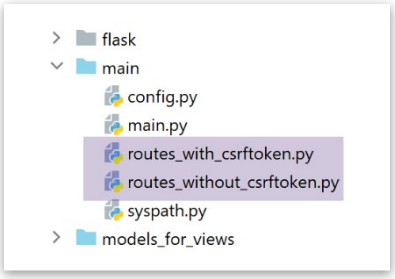
```
    <!-- colonne 8 : valeur paramètre [réduction] - balise <td> -->
    <!-- colonne 9 : valeur paramètre [taux] (de l'impôt) - balise <td> -->
    <!-- colonne 10 : lien de suppression de la simulation - balise <td> -->
    <tr>
        <th scope="row">{{simulation.id}}</th>
        <td>{{simulation.marié}}</td>
        <td>{{simulation.enfants}}</td>
        <td>{{simulation.salaire}}</td>
        <td>{{simulation.impôt}}</td>
        <td>{{simulation.surcôte}}</td>
        <td>{{simulation.décôte}}</td>
        <td>{{simulation.réduction}}</td>
        <td>{{simulation.taux}}</td>
        <td><a href="/supprimer-simulation/{{simulation.id}}{{modèle.csrf_token}}">Supprimer</a></td>
    </tr>
    {% endfor %}
    </tr>
    </tbody>
</table>
{% endif %}
```

Le fragment du menu `[v-menu.html]`

```
<!-- menu Bootstrap -->
<nav class="nav flex-column">
    <!-- affichage d'une liste de liens HTML -->
    {% for optionMenu in modèle.optionsMenu %}
    <a class="nav-link" href="{{optionMenu.url}}{{modèle.csrf_token}}">{{optionMenu.text}}</a>
    {% endfor %}
</nav>
```

34.3.4 Les routes

Il y a désormais deux types de routes, selon que celles-ci utilisent ou non un jeton CSRF :



- **[routes_without_csrftoken]** sont les routes sans jeton CSRF. Ce sont les routes de la version précédente ;
- **[routes_with_csrftoken]** sont les routes avec jeton CSRF.

Dans **[routes_with_csrftoken]**, les routes ont désormais un paramètre supplémentaire, le jeton CSRF :

```
# le front controller
def front_controller() -> tuple:
    # on fait suivre la requête au contrôleur principal
    main_controller = config['mvc']['controllers']['main-controller']
    return main_controller.execute(request, session, config)

@app.route('/', methods=['GET'])
def index() -> tuple:
    # redirection vers /init-session/html
    return redirect(url_for("init_session", type_response="html", csrf_token=generate_csrf()), status.HTTP_302_FOUND)

# init-session
@app.route('/init-session/<string:type_response>/<string:csrf_token>', methods=['GET'])
def init_session(type_response: str, csrf_token: str) -> tuple:
    # on exécute le contrôleur associé à l'action
    return front_controller()

# authentifier-utilisateur
@app.route('/authentifier-utilisateur/<string:csrf_token>', methods=['POST'])
def authentifier_utilisateur(csrf_token: str) -> tuple:
    # on exécute le contrôleur associé à l'action
    return front_controller()

# calculer-impot
@app.route('/calculer-impot/<string:csrf_token>', methods=['POST'])
```

```
26.  def calculer_impot(csrf_token: str) -> tuple:
27.      # on exécute le contrôleur associé à l'action
28.      return front_controller()
29.
30.  # calcul de l'impôt par lots
31.  @app.route('/calculer-impots/<string:csrf_token>', methods=['POST'])
32.  def calculer_impots(csrf_token: str):
33.      # on exécute le contrôleur associé à l'action
34.      return front_controller()
35.
36.  # lister-simulations
37.  @app.route('/lister-simulations/<string:csrf_token>', methods=['GET'])
38.  def lister_simulations(csrf_token: str) -> tuple:
39.      # on exécute le contrôleur associé à l'action
40.      return front_controller()
41.
42.  # supprimer-simulation
43.  @app.route('/supprimer-simulation/<int:numero>/<string:csrf_token>', methods=['GET'])
44.  def supprimer_simulation(numero: int, csrf_token: str) -> tuple:
45.      # on exécute le contrôleur associé à l'action
46.      return front_controller()
47.
48.  # fin-session
49.  @app.route('/fin-session/<string:csrf_token>', methods=['GET'])
50.  def fin_session(csrf_token: str) -> tuple:
51.      # on exécute le contrôleur associé à l'action
52.      return front_controller()
53.
54.  # afficher-calcul-impot
55.  @app.route('/afficher-calcul-impot/<string:csrf_token>', methods=['GET'])
56.  def afficher_calcul_impot(csrf_token: str) -> tuple:
57.      # on exécute le contrôleur associé à l'action
58.      return front_controller()
59.
60.  # get-admindata
61.  @app.route('/get-admindata/<string:csrf_token>', methods=['GET'])
62.  def get_admindata(csrf_token: str) -> tuple:
63.      # on exécute le contrôleur associé à l'action
64.      return front_controller()
```

Toutes les routes ont désormais le jeton CSRF dans leurs paramètres, même la route **[/init-session]**. Cela signifie que le client ne peut pas démarrer l'application en tapant directement l'URL **[/init-session/html]** car il y manquera le jeton CSRF. Il doit désormais obligatoirement passer par l'URL **[/]** des lignes 7-10.

Le choix des routes est fait dans le script principal **[main]** :

```
1.  …
2.  # le thread principal n'a plus besoin du logger
3.  logger.close()
4.
5.  # s'il y a eu erreur on s'arrête
6.  if erreur:
7.      sys.exit(2)
8.
9.  # import des routes de l'application web
10. if config['parameters']['with_csrftoken']:
11.     import routes_with_csrftoken as routes
12. else:
13.     import routes_without_csrftoken as routes
14.
15. # configuration des routes
16. routes.config = config
17.
18. # démarrage application Flask
19. routes.execute(__name__)
```

- lignes 9-13 : choix des routes selon que l'application utilise ou non des jetons CSRF ;

34.3.5 Le contrôleur [MainController]

A chaque requête, le serveur doit vérifier la présence du jeton CSRF. Nous ferons cela dans le contrôleur principal **[MainController]** qui voit passer toutes les requêtes :

```
1.  from flask_wtf.csrf import generate_csrf, validate_csrf
2.  …
3.          # on traite la requête
4.          try:
5.              # logger
```

```
6.              logger = Logger(config['parameters']['logsFilename'])
7.
8.              …
9.
10.             # on récupère les éléments du path
11.             params = request.path.split('/')
12.
13.             # l'action est le 1er élément
14.             action = params[1]
15.
16.             …
17.
18.             if config['parameters']['with_csrftoken']:
19.                 # le csrf_token est le dernier élément du path
20.                 csrf_token = params.pop()
21.                 # on vérifie la validité du token
22.                 # une exception sera lancée si le csrf_token n'est pas celui attendu
23.                 validate_csrf(csrf_token)
24.
25.             …
26.
27.         except ValidationError as exception:
28.             # csrf token invalide
29.             résultat = {"action": action, "état": 121, "réponse": [f"{exception}"]}
30.             status_code = status.HTTP_400_BAD_REQUEST
31.
32.         except BaseException as exception:
33.             # autres exceptions (inattendues)
34.             résultat = {"action": action, "état": 131, "réponse": [f"{exception}"]}
35.             status_code = status.HTTP_400_BAD_REQUEST
36.
37.         finally:
38.             pass
39.
40.         # on ajoute le csrf_token au résultat
41.         résultat['csrf_token'] = generate_csrf()
42.
43.         # on logue le résultat envoyé au client
44.         log = f"[MainController] {résultat}\n"
45.         logger.write(log)
```

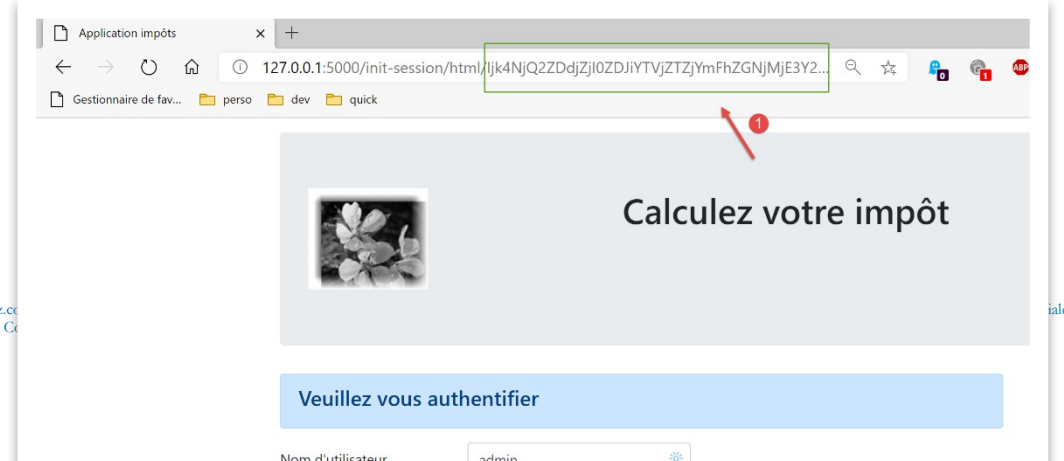
- ligne 20 : on récupère le jeton CSRF dans l'URL de la requête du type **[http://machine :port/chemin/action/param1/param2/…/csrf_token]**. Le jeton de session est toujours le dernier élément de l'URL ;
- ligne 23 : la validité du jeton CSRF récupéré dans l'URL avec le jeton CSRF de la session est vérifiée. S'il n'est pas valide, la fonction **[validate_csrf]** lance une exception de type **[ValidationError]** (ligne 27) ;
- ligne 41 : le jeton CSRF est mis dans le résultat envoyé au client. Les clients jSON et XML en auront besoin. En effet, ces clients ne reçoivent pas de pages HTML avec le jeton CSRF dans les liens contenus dans les pages. Ils le recevront donc dans le résultat jSON ou XML envoyé par le serveur ;

Note : la fonction **[validate_csrf]** de la ligne 23 ne vérifie pas une concordance stricte. Le jeton CSRF est mémorisé dans la session avec la clé **[csrf_token]**. Les test semblent montrer qu'un jeton CSRF est valide s'il a été généré au cours de la session. Ainsi si manuellement, dans l'URL affichée dans le navigateur, par exemple (/lister-simulations/xyz), vous remplacez le jeton **[xyz]** CSRF par un autre **[abc]** déjà reçu lors d'une précédente action, l'action **[/lister-simulations]** va réussir ;

34.4 Tests avec un navigateur

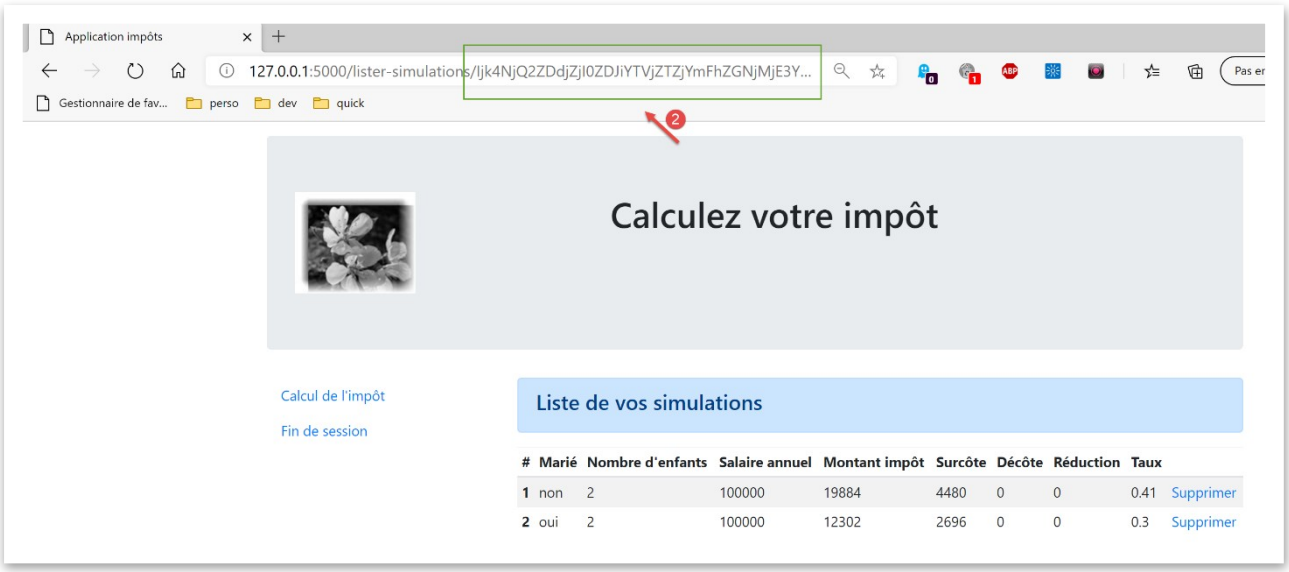
On :

- lance le serveur avec le paramètre **[with_csrftoken]** à **[True]** ;
- demande l'URL **[http://localhost:5000]** avec un navigateur ;

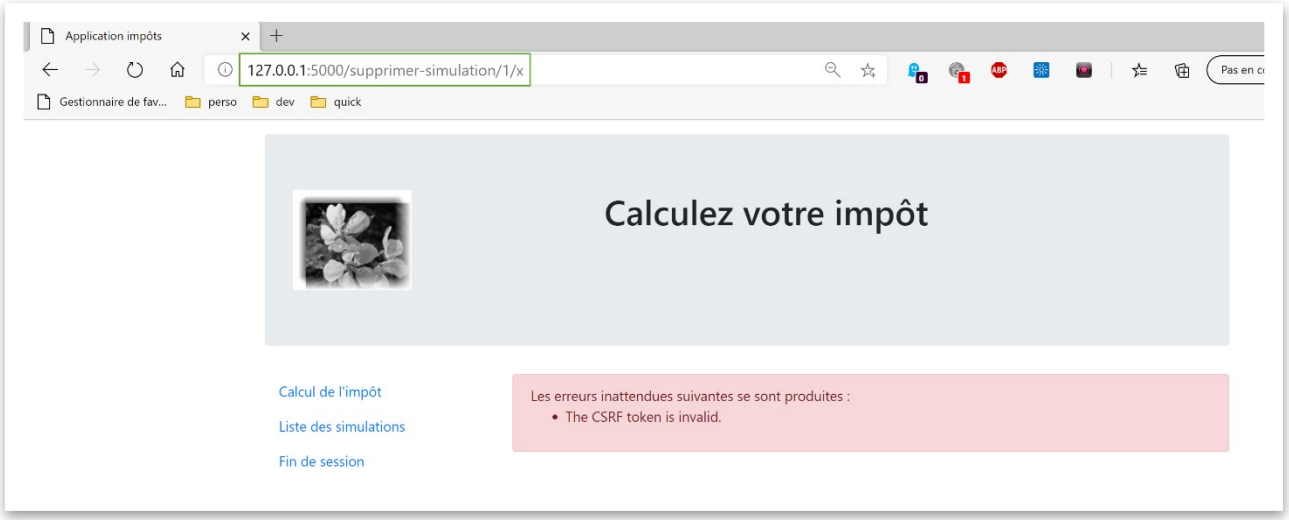


- en **[1]**, le jeton CSRF ;

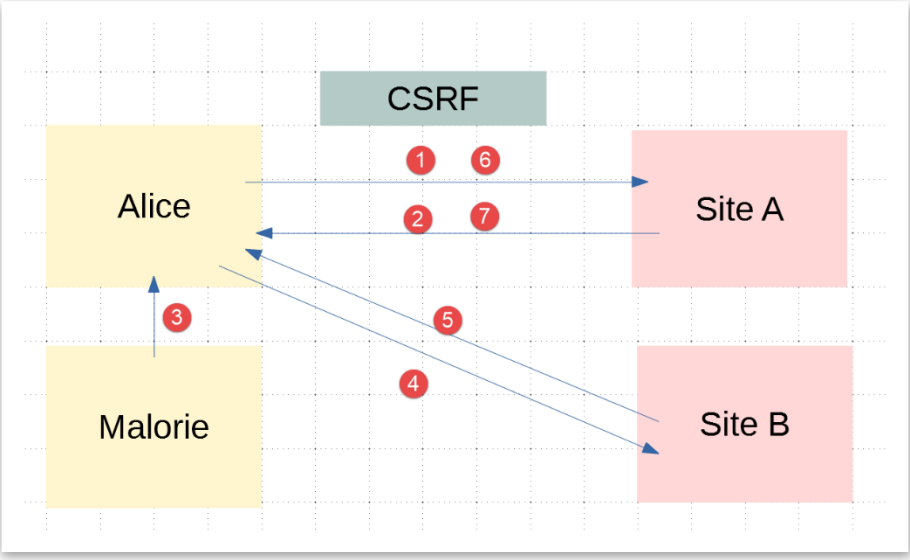
Faisons des manipulations jusqu'à avoir une liste de simulations :



Maintenant, tapons à la main, l'URL **[http://localhost:5000/supprimer-simulation/1/x]** pour supprimer la simulation d'id=1. On met volontairement un jeton CSRF incorrect pour voir ce qui se passe. La réponse du serveur est la suivante :



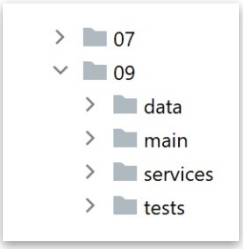
Note 1 : il n'est pas sûr que la méthode utilisée ici soit toujours suffisante pour contrer les attaques CSRF. Revenons au schéma de l'attaque :



Si le script Javascript téléchargé en **[5]** est capable de lire l'historique du navigateur utilisé par Alice, il sera capable de récupérer les URL exécutées par le navigateur, des URL telles que **[/cible/csrf_token]**. Il pourra alors récupérer le jeton de session **[csrf_token]** et faire son attaque en **[6-7]**. Néanmoins, le navigateur autorise uniquement l'exploitation de l'historique de la fenêtre du navigateur dans laquelle s'exécute le script. Si donc Alice n'utilise pas la même fenêtre pour travailler avec le site A **[1-2]** et lire le message de Malorie **[3]**, l'attaque CSRF ne sera pas possible.

34.5 Clients console

Une autre façon de tester la version 14 de l'application est de reprendre les tests de la version 12 et de les adapter au nouveau



serveur.
Le dossier **[impots/http-clients/09]** est obtenu initialement par recopie du dossier **[impots/http-clients/07]**. Il est ensuite modifié.

Revenons aux routes qui initialisent une session :

```
1.  # racine de l'application
2.  @app.route('/', methods=['GET'])
3.  def index() -> tuple:
4.      # redirection vers /init-session/html
5.      return redirect(url_for("init_session", type_response="html", csrf_token=generate_csrf()), status.HTTP_302_FOUND)
6.
7.  # init-session-with-csrf-token
8.  @app.route('/init-session/<string:type_response>/<string:csrf_token>', methods=['GET'])
9.  def init_session(type_response: str, csrf_token: str) -> tuple:
10.     # on exécute le contrôleur associé à l'action
11.     return front_controller()
```

Aucune de ces routes ne convient pour initialiser une session jSON ou XML :

- lignes 2-5 : la route **[/]** initialise une session HTML ;
- lignes 8-11 : la route **[/init-session]** nécessite un jeton CSRF qu'on ne connaît pas ;

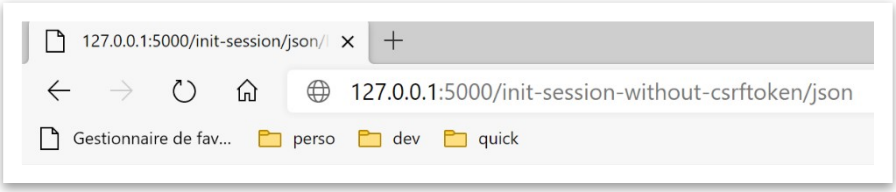
Nous décidons d'ajouter une nouvelle route au serveur :

```
1.  # init-session-without-csrftoken
2.  @app.route('/init-session-without-csrftoken/<string:type_response>', methods=['GET'])
3.  def init_session_without_csrftoken(type_response: str) -> tuple:
```

```
4.      # redirection vers /init-session/type_response
5.      return redirect(url_for("init_session", type_response=type_response, csrf_token=generate_csrf()), status.HTTP_302_FOUND
    )
```

- ligne 2 : la nouvelle route. Elle n'attend pas de jeton CSRF. On est ainsi revenu à la route **[/init-session]** de la version précédente ;
- lignes 4-5 : on redirige le client (jSON, XML, HTML) vers la route **[/init-session]** ayant le jeton CSRF dans ses paramètres ;

On peut essayer cette nouvelle route avec un navigateur :

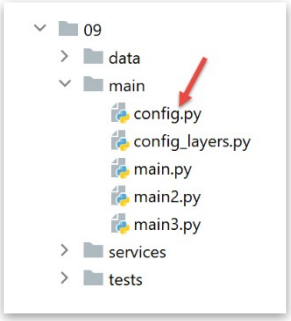


La réponse du serveur (configuré avec **[with_csrftoken=True]**) est la suivante :



- en **[1]**, le serveur a été redirigé vers la route **[/init-session]** avec jeton CSRF dans l'URL ;
- en **[2]**, le jeton CSRF est dans le dictionnaire jSON envoyé par le serveur associé à la clé **[csrf_token]** ;

Revenons au code du client :



Nous modifions la configuration **[config]** de la façon suivante :

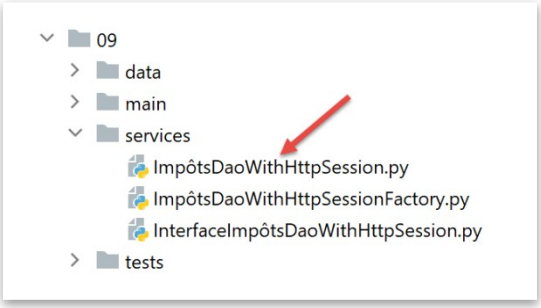
```
1.      config.update({
2.          # fichier des contribuables
3.          "taxpayersFilename": f"{script_dir}/../data/input/taxpayersdata.txt",
4.          # fichier des résultats
5.          "resultsFilename": f"{script_dir}/../data/output/résultats.json",
6.          # fichier des erreurs
7.          "errorsFilename": f"{script_dir}/../data/output/errors.txt",
8.          # fichier de logs
9.          "logsFilename": f"{script_dir}/../data/logs/logs.txt",
10.         # le serveur de calcul de l'impôt
11.         "server": {
12.             "urlServer": "http://127.0.0.1:5000",
13.             "user": {
14.                 "login": "admin",
15.                 "password": "admin"
16.             },
17.             "url_services": {
18.                 "calculate-tax": "/calculer-impot",
19.                 "get-admindata": "/get-admindata",
20.                 "calculate-tax-in-bulk-mode": "/calculer-impots",
21.                 "init-session": "/init-session-without-csrftoken",
22.                 "end-session": "/fin-session",
```

```
23.                 "authenticate-user": "/authentifier-utilisateur",
24.                 "get-simulations": "/lister-simulations",
25.                 "delete-simulation": "/supprimer-simulation",
26.             }
27.         },
28.         # mode debug
29.         "debug": True,
30.         # csrf_token
31.         "with_csrftoken": True,
32.     }
33.     )
34. …
35.     # route init-session
36.     url_services = config['server']['url_services']
37.     if config['with_csrftoken']:
38.         url_services['init-session'] = '/init-session-without-csrftoken'
39.     else:
40.         url_services['init-session'] = '/init-session'
```

- ligne 31 : un booléen indiquera au client si le serveur auquel il s'adresse travaille ou non avec des jetons CSRF ;
- lignes 37-40 : on fixe l'URL de service de l'action **[init-session]** :
  - si le serveur utilise des jetons CSRF alors l'URL de service est **[/init-session-without-csrftoken]** ;
  - sinon l'URL de service est **[/init-session]** ;

La route **[/init-session-without-csrftoken]** a été présentée. Elle permet à un client jSON / XML de démarrer une session avec le serveur sans posséder de jeton CSRF. Il trouvera ce jeton dans la réponse du serveur.

Nous modifions ensuite la classe **[ImpôtsDaoWithHttpSession]** implémentant la couche **[dao]** du client :




```
1.  # imports
2.  import json
3.
4.  import requests
5.  import xmltodict
6.  from flask_api import status
7.
8.  from AbstractImpôtsDao import AbstractImpôtsDao
9.  from AdminData import AdminData
10. from ImpôtsError import ImpôtsError
11. from InterfaceImpôtsDaoWithHttpSession import InterfaceImpôtsDaoWithHttpSession
12. from TaxPayer import TaxPayer
13.
14. class ImpôtsDaoWithHttpSession(InterfaceImpôtsDaoWithHttpSession):
15.
16.     # constructeur
17.     def __init__(self, config: dict):
18.         # initialisation parent
19.         AbstractImpôtsDao.__init__(self, config)
20.         # mémorisation éléments de la configuration
21.         # config générale
22.         self.__config = config
23.         # serveur
24.         self.__config_server = config["server"]
25.         # services
26.         self.__config_services = config["server"]['url_services']
27.         # mode debug
28.         self.__debug = config["debug"]
29.         # logger
30.         self.__logger = None
31.         # cookies
32.         self.__cookies = None
33.         # type de session (json, xml)
34.         self.__session_type = None
35.         # jeton CSRF
36.         self.__csrf_token = None
```

```
37.
38.      # étape request / response
39.      def get_response(self, method: str, url_service: str, data_value: dict = None, json_value=None):
40.          # [method] : méthode HTTP GET ou POST
41.          # [url_service] : URL de service
42.          # [data] : paramètres du POST en x-www-form-urlencoded
43.          # [json] : paramètres du POST en json
44.          # [cookies]: cookies à inclure dans la requête
45.
46.          # on doit avoir une session XML ou JSON, sinon on ne pourra pas gérer la réponse
47.          if self.__session_type not in ['json', 'xml']:
48.              raise ImpôtsError(73, "il n'y a pas de session valide en cours")
49.
50.          # on ajoute le jeton CSRF à l'URL de service
51.          if self.__csrf_token:
52.              url_service = f"{url_service}/{self.__csrf_token}"
53.
54.          # exécution de la requête
55.          response = requests.request(method,
56.                                      url_service,
57.                                      data=data_value,
58.                                      json=json_value,
59.                                      cookies=self.__cookies,
60.                                      allow_redirects=True)
61.
62.          # mode debug ?
63.          if self.__debug:
64.              # logueur
65.              if not self.__logger:
66.                  self.__logger = self.__config['logger']
67.              # on logue
68.              self.__logger.write(f"{response.text}\n")
69.
70.          # résultat
71.          if self.__session_type == "json":
72.              résultat = json.loads(response.text)
73.          else:  # xml
74.              résultat = xmltodict.parse(response.text[39:])['root']
75.
76.          # on récupère les cookies de la réponse s'il y en a
77.          if response.cookies:
78.              self.__cookies = response.cookies
79.
80.          # on récupère le jeton CSRF
81.          if self.__config['with_csrftoken']:
82.              self.__csrf_token = résultat.get('csrf_token', None)
83.
84.          # code de statut
85.          status_code = response.status_code
86.
87.          # si code de statut différent de 200 OK
88.          if status_code != status.HTTP_200_OK:
89.              raise ImpôtsError(35, résultat['réponse'])
90.
91.          # on rend le résultat
92.          return résultat['réponse']
93.
94.
95.      def init_session(self, session_type: str):
96.          # on note le type de la session
97.          self.__session_type = session_type
98.
99.          # on supprime le jeton CSRF des précédents appels
100.         self.__csrf_token = None
101.
102.         # on demande l'URL de l'action init-session
103.         url_service = f"{self.__config_server['urlServer']}{self.__config_services['init-session']}/{session_type}"
104.
105.         # exécution requête
106.         self.get_response("GET", url_service)
107. …
```

- lignes 38-92 : la gestion du jeton CSRF se passe principalement dans la méthode **[get_response]** ;
- ligne 60 : le point important est le paramètre **[allow_redirects=True]**. C'est sa valeur par défaut mais on a tenu à le mettre en relief ;

  Lorsqu'on est en mode **[with_csrftoken=True]** :

  - les clients commencent leur dialogue avec le serveur par l'appel à la route **[/init-session_without_csftoken/type_response]** ;
  - le serveur répond à cette requête par une redirection vers la route **[/init-session/type_response/csrf_token]** ;
  - à cause du paramètre **[allow_redirects=True]**, cette redirection va être suivie par le client **[requests]** ;

- le jeton CSRF sera trouvé dans le résultat récupéré aux lignes 72 et 74 associé à la clé **[csrf_token]** ;

Lorsqu'on est en mode **[with_csrftoken=False]** :

- les clients commencent leur dialogue avec le serveur par l'appel à la route **[/init-session /type_response]** ;
- le serveur répond à cette requête par une redirection vers la route **[/init-session/type_response]** ;
- à cause du paramètre **[allow_redirects=True]**, cette redirection va être suivie par le client **[requests]** ;
- il n'y a pas de jeton CSRF à récupérer aux lignes 81-82. La propriété **[self.__csrf_token]** reste alors toujours à None (ligne 36) ;

- lignes 51-52 : pour toutes les requêtes suivantes, le jeton CSRF, s'il existe, est ajouté à la route initiale ;
- lignes 81-82 : le nouveau jeton que génère le serveur à chaque nouvelle requête du client est mémorisé localement pour être renvoyé ligne 52 à la requête suivante ;

Par ailleurs, la méthode **[init_session]** évolue un peu :

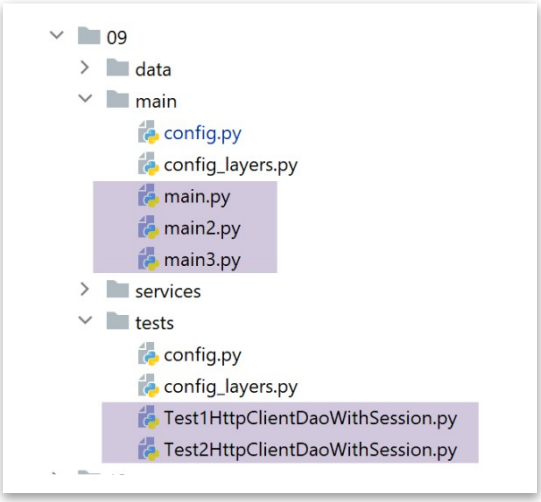
```
1.      def init_session(self, session_type: str):
2.          # on note le type de la session
3.          self.__session_type = session_type
4.
5.          # on supprime le jeton CSRF des précédents appels
6.          self.__csrf_token = None
7.
8.          # on demande l'URL de l'action init-session
9.          url_service = f"{self.__config_server['urlServer']}{self.__config_services['init-session']}/{session_type}"
10.
11.         # exécution requête
12.         self.get_response("GET", url_service)
```

Il faut se rappeler ici qu'on a créé une route **[/init-session-without-csrftoken/<type-response>]** pour initialiser le dialogue client / serveur sans jeton CSRF. Or nous avons vu que la méthode **[get_response]** appelée ligne 12 du code ajoute systématiquement à la fin de l'URL de service le jeton CSRF mémorisé dans **[self.__csrf_token]**. C'est pourquoi ligne 6 du code on supprime ce jeton CSRF s'il existait.

C'est tout. Pour les tests, on exécutera :

- les clients console **[main, main2, main3]** ;
- les classes de test **[Test1HttpClientDaoWithSession]** et **[Test2HttpClientDaoWithSession]** ;

en mettant successivement à True puis False, le paramètre de configuration **[with_csrftoken]**.



Voici comme exemple les logs obtenus à l'exécution du client **[main json]** avec **[with_csrftoken=True]** :

```
1.  2020-08-08 16:33:23.317903, MainThread : début du calcul de l'impôt des contribuables
2.  2020-08-08 16:33:23.317903, Thread-1 : début du calcul de l'impôt des 4 contribuables
3.  2020-08-08 16:33:23.317903, Thread-2 : début du calcul de l'impôt des 2 contribuables
4.  2020-08-08 16:33:23.317903, Thread-3 : début du calcul de l'impôt des 4 contribuables
5.  2020-08-08 16:33:23.317903, Thread-4 : début du calcul de l'impôt des 1 contribuables
```

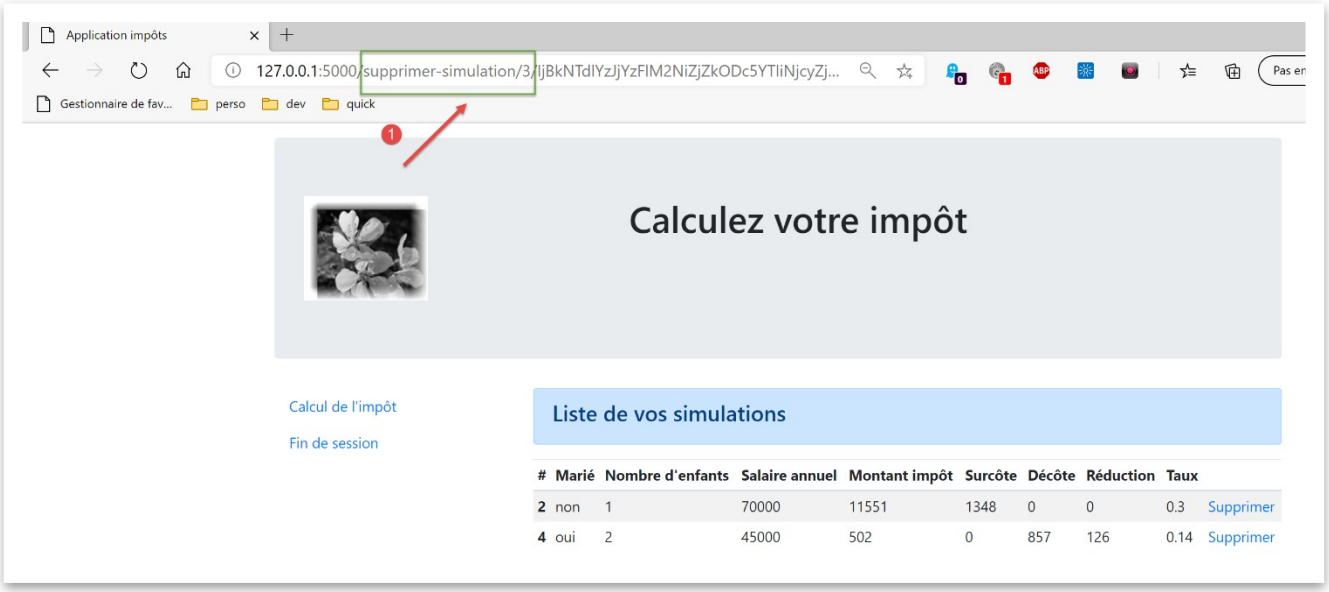
```
6.  2020-08-08 16:33:23.379221, Thread-2 : {"action": "init-session", "état": 700, "réponse": ["session
    démarrée avec le type de réponse json"], "csrf_token":
    "ImFiZmZkYjZmMzFkZDc2YWRjNWYwOGM0NTBmMGM4ODJjYzViOWI4NGEi.Xy63sw.H5L0--yWsvfaWvggrGw78z5VnN0"}
7.  2020-08-08 16:33:23.381073, Thread-4 : {"action": "init-session", "état": 700, "réponse": ["session
    démarrée avec le type de réponse json"], "csrf_token":
    "ImY5YzQyMjlkYzcyYmM4YmZiMGI0NWY5MjE4MzIzNDExZjc0MGQ3MWQi.Xy63sw.q6olg7IP_g2ro_RBFRCX1BX90g8"}
8.  2020-08-08 16:33:23.386982, Thread-3 : {"action": "init-session", "état": 700, "réponse": ["session
    démarrée avec le type de réponse json"], "csrf_token":
    "IjkxZGNlN2YyMmUxMjQ0M2Y0MTdjNDQ4ZmQ1MDMxZjkwNjBhNzAzZjMi.Xy63sw.-6buL11No3UJBlElpW4tX4B-lp0"}
9.  2020-08-08 16:33:23.390269, Thread-1 : {"action": "init-session", "état": 700, "réponse": ["session
    démarrée avec le type de réponse json"], "csrf_token":
    "IjIxNmU4MDQyZDFmZmIyZDlmZjE4MzNlNDUzYzFjMGYxMWYxYzEwNGYi.Xy63sw.fgs6Cm2owsJf4NjTm7gKrVESabI"}
10. 2020-08-08 16:33:23.413206, Thread-2 : {"action": "authentifier-utilisateur", "état": 200, "réponse":
    "Authentification réussie", "csrf_token":
    "ImFiZmZkYjZmMzFkZDc2YWRjNWYwOGM0NTBmMGM4ODJjYzViOWI4NGEi.Xy63sw.H5L0--yWsvfaWvggrGw78z5VnN0"}
11. 2020-08-08 16:33:23.422877, Thread-2 : {"action": "calculer-impots", "état": 1500, "réponse": [{"marié":
    "non", "enfants": 3, "salaire": 100000, "impôt": 16782, "surcôte": 7176, "taux": 0.41, "décôte": 0,
    "réduction": 0, "id": 1}, {"marié": "oui", "enfants": 3, "salaire": 100000, "impôt": 9200, "surcôte": 2180,
    "taux": 0.3, "décôte": 0, "réduction": 0, "id": 2}], "csrf_token":
    "ImFiZmZkYjZmMzFkZDc2YWRjNWYwOGM0NTBmMGM4ODJjYzViOWI4NGEi.Xy63sw.H5L0--yWsvfaWvggrGw78z5VnN0"}
12. 2020-08-08 16:33:23.428622, Thread-4 : {"action": "authentifier-utilisateur", "état": 200, "réponse":
    "Authentification réussie", "csrf_token":
    "ImY5YzQyMjlkYzcyYmM4YmZiMGI0NWY5MjE4MzIzNDExZjc0MGQ3MWQi.Xy63sw.q6olg7IP_g2ro_RBFRCX1BX90g8"}
13. 2020-08-08 16:33:23.429127, Thread-3 : {"action": "authentifier-utilisateur", "état": 200, "réponse":
    "Authentification réussie", "csrf_token":
    "IjkxZGNlN2YyMmUxMjQ0M2Y0MTdjNDQ4ZmQ1MDMxZjkwNjBhNzAzZjMi.Xy63sw.-6buL11No3UJBlElpW4tX4B-lp0"}
14. 2020-08-08 16:33:23.429127, Thread-1 : {"action": "authentifier-utilisateur", "état": 200, "réponse":
    "Authentification réussie", "csrf_token":
    "IjIxNmU4MDQyZDFmZmIyZDlmZjE4MzNlNDUzYzFjMGYxMWYxYzEwNGYi.Xy63sw.fgs6Cm2owsJf4NjTm7gKrVESabI"}
15. 2020-08-08 16:33:23.429127, Thread-2 : {"action": "fin-session", "état": 400, "réponse": "session
    réinitialisée", "csrf_token":
    "IjU1YjlmZDA0OWRhNTJlODFmYjgyYjlhM2ExYWNhZmUzNTk2NjA5NGIi.Xy63sw.nyNSvkcG6iG0oIMBjtYPo8ySgdw"}
16. 2020-08-08 16:33:23.438519, Thread-2 : fin du calcul de l'impôt des 2 contribuables
17. 2020-08-08 16:33:23.443033, Thread-4 : {"action": "calculer-impots", "état": 1500, "réponse": [{"marié":
    "oui", "enfants": 3, "salaire": 200000, "impôt": 42842, "surcôte": 17283, "taux": 0.41, "décôte": 0,
    "réduction": 0, "id": 1}], "csrf_token":
    "ImY5YzQyMjlkYzcyYmM4YmZiMGI0NWY5MjE4MzIzNDExZjc0MGQ3MWQi.Xy63sw.q6olg7IP_g2ro_RBFRCX1BX90g8"}
18. 2020-08-08 16:33:23.446510, Thread-3 : {"action": "calculer-impots", "état": 1500, "réponse": [{"marié":
    "oui", "enfants": 5, "salaire": 100000, "impôt": 4230, "surcôte": 0, "taux": 0.14, "décôte": 0,
    "réduction": 0, "id": 1}, {"marié": "non", "enfants": 0, "salaire": 100000, "impôt": 22986, "surcôte": 0,
    "taux": 0.41, "décôte": 0, "réduction": 0, "id": 2}, {"marié": "oui", "enfants": 2, "salaire": 30000,
    "impôt": 0, "surcôte": 0, "taux": 0.0, "décôte": 0, "réduction": 0, "id": 3}, {"marié": "non", "enfants":
    0, "salaire": 200000, "impôt": 64210, "surcôte": 7498, "taux": 0.45, "décôte": 0, "réduction": 0, "id":
    4}], "csrf_token":
    "IjkxZGNlN2YyMmUxMjQ0M2Y0MTdjNDQ4ZmQ1MDMxZjkwNjBhNzAzZjMi.Xy63sw.-6buL11No3UJBlElpW4tX4B-lp0"}
19. 2020-08-08 16:33:23.453477, Thread-1 : {"action": "calculer-impots", "état": 1500, "réponse": [{"marié":
    "oui", "enfants": 2, "salaire": 55555, "impôt": 2814, "surcôte": 0, "taux": 0.14, "décôte": 0, "réduction":
    0, "id": 1}, {"marié": "oui", "enfants": 2, "salaire": 50000, "impôt": 1384, "surcôte": 0, "taux": 0.14,
    "décôte": 384, "réduction": 347, "id": 2}, {"marié": "oui", "enfants": 3, "salaire": 50000, "impôt": 0,
    "surcôte": 0, "taux": 0.14, "décôte": 720, "réduction": 0, "id": 3}, {"marié": "non", "enfants": 2,
    "salaire": 100000, "impôt": 19884, "surcôte": 4480, "taux": 0.41, "décôte": 0, "réduction": 0, "id": 4}],
    "csrf_token":
    "IjIxNmU4MDQyZDFmZmIyZDlmZjE4MzNlNDUzYzFjMGYxMWYxYzEwNGYi.Xy63sw.fgs6Cm2owsJf4NjTm7gKrVESabI"}
20. 2020-08-08 16:33:23.457912, Thread-4 : {"action": "fin-session", "état": 400, "réponse": "session
    réinitialisée", "csrf_token": "IjQ0ZDQxODgzN2M5NjRiYWI0NjA2MTk5YWFkNGFhMzY1M2IxNWMyNDIi.Xy63sw.mOa5MKXvJ-
    EXf_qEok-OqC5j_mg"}
21. 2020-08-08 16:33:23.458442, Thread-4 : fin du calcul de l'impôt des 1 contribuables
22. 2020-08-08 16:33:23.459045, Thread-3 : {"action": "fin-session", "état": 400, "réponse": "session
    réinitialisée", "csrf_token":
    "ImQ0NDZlYmViYjY1ZDUxYzJhMTNmM2JiZTRkMjBjZGJkYzE0OGVkYzMi.Xy63sw.fviTJz4zFDqVLlVlkrosT_JRPww"}
23. 2020-08-08 16:33:23.459700, Thread-3 : fin du calcul de l'impôt des 4 contribuables
24. 2020-08-08 16:33:23.460492, Thread-1 : {"action": "fin-session", "état": 400, "réponse": "session
    réinitialisée", "csrf_token":
    "Ijg3MjQ1NGUyYTUyOGEyNTdmZmNmYWZkMmU2OTgyMzUwNjI1YTlhZjIi.Xy63sw.I0xBl9Q8DzsuXPSgOdeARc_VKBA"}
25. 2020-08-08 16:33:23.460492, Thread-1 : fin du calcul de l'impôt des 4 contribuables
26. 2020-08-08 16:33:23.460492, MainThread : fin du calcul de l'impôt des contribuables
```

Si on regarde les jetons CSRF reçus successivement on voit qu'ils sont tous différents.

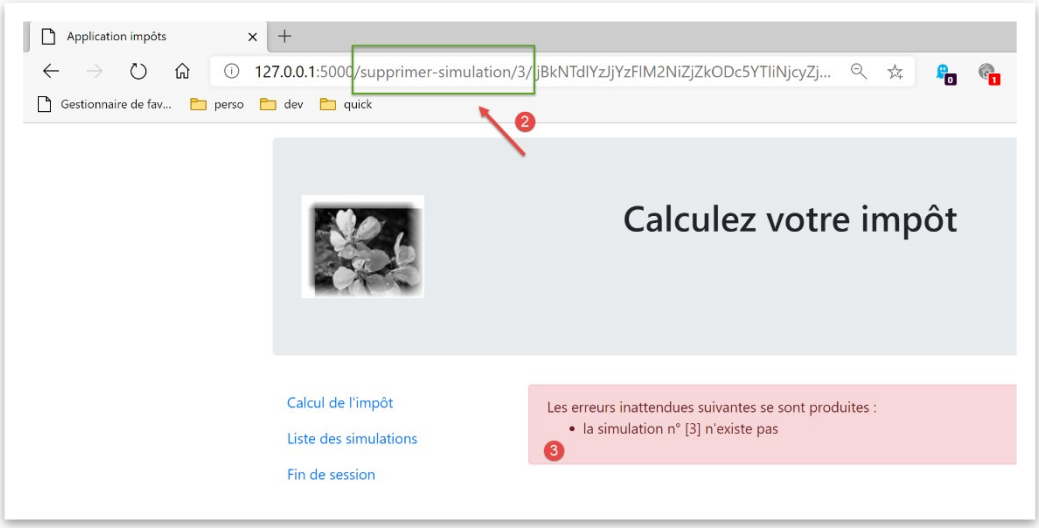
35 Exercice d'application : version 15

35.1 Introduction

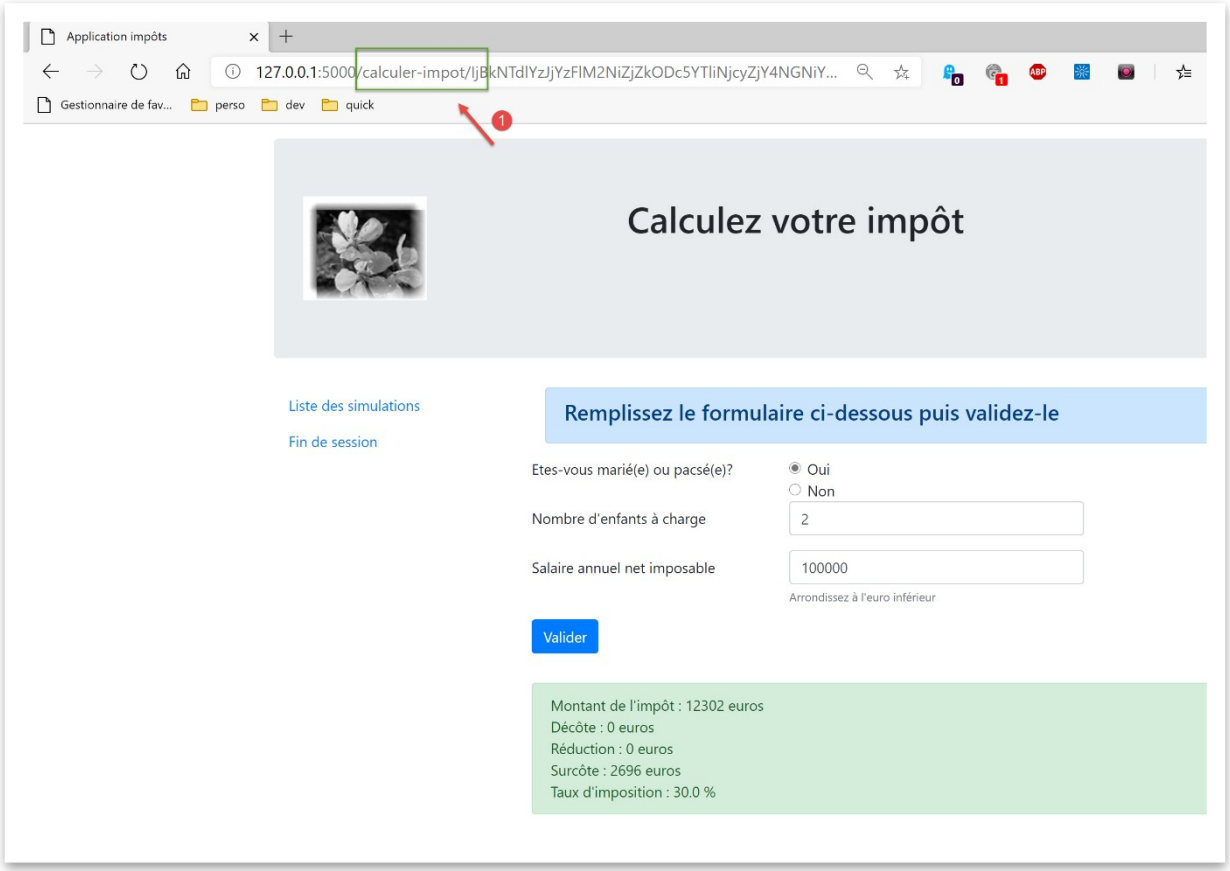
Cette version vise à résoudre des problèmes liés au rafraîchissement des pages de l'application dans le navigateur (F5). Prenons un exemple. L'utilisateur vient de suprimer la simulation d'id=3 :



Après la suppression l'URL dans le navigateur est **`[/supprimer-simulation/3/…]`**. Si l'utilisateur rafraîchit la page (F5), l'URL **`[1]`** est rejouée. On demande donc de nouveau la suppression de la simulation d'id=3. Le résultat est alors le suivant :

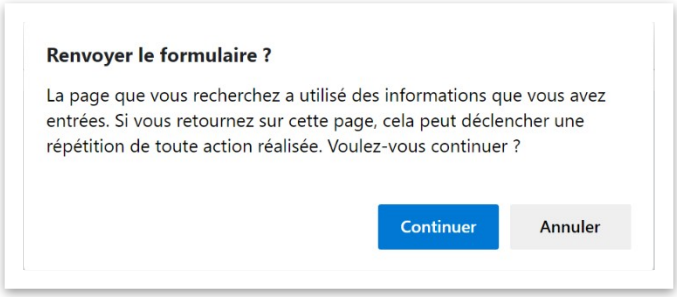


Voici un autre exemple. L'utilisateur vient de calculer un impôt :



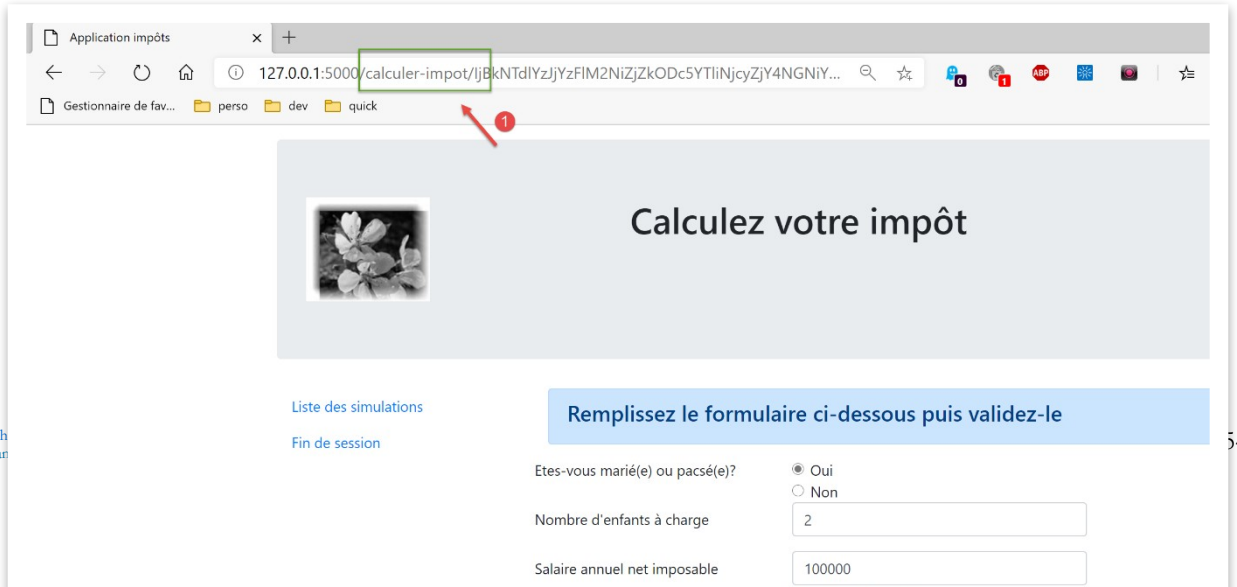
- en **[1]**, l'URL **[/calculer-impot]** qui vient d'être interrogée avec un POST ;

Si l'utilisateur rafraîchit la page (F5), il obtient un message d'avertissement :



L'opération de rafraîchissement de page rejoue la dernière requête exécutée par le navigateur, ici un **[POST /calculer-impot]**. Lorsqu'on demande à rejouer un POST, les navigateurs émettent un avertissement analogue à celui ci-dessus. Celui-ci avertit qu'il est en train de rejouer une action déjà faite. Supposons que ce POST ait effectué un achat, il serait malheureux de refaire celui-ci.

Par ailleurs, nous allons limiter la possibilité de l'utilisateur de taper des URL dans son navigateur. Prenons par exemple l'une des vues précédentes :



Les liens offerts par cette page sont :

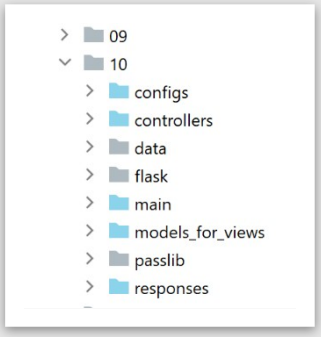
- **[Liste des simulations]** associé à l'URL **[/lister-simulations]** ;
- **[Fin de session]** associé à l'URL **[/fin-session]** ;
- **[Valider]** associé à l'URL (on ne la voit pas ci-dessus) **[/calculer-impot]** ;

Lorsque la vue du calcul de l'impôt sera affichée, nous n'accepterons que les actions **[/lister-simulations, /fin-session, /calculer-impot]**. Si l'utilisateur tape une autre action dans son navigateur, une erreur sera déclarée. On fera ce type de vérification pour les quatre vues de l'application.

Nous nous proposons de résoudre le problème du rafraîchissement des pages de la façon suivante :

- nous allons distinguer deux types d'actions :
  - les actions ADS (Action Do Something) modifiant l'état de l'application. Les actions ADS ont en général des paramètres dans l'URL ou le corps de la requête ;
  - les actions ASV (Action Show View) affichant une vue sans modifier l'état de l'application. Il y aura autant d'actions ASV que de vues V. Les actions ASV n'ont aucun paramètre ;
- les actions ADS jusqu'à maintenant s'exécutaient puis se terminaient par l'affichage d'une vue V après avoir préparé le modèle M de celle-ci. Désormais, elles mettront le modèle M de la vue en session et demanderont au navigateur de se rediriger vers l'action ASV chargée d'afficher la vue V ;
- les vues V ne seront affichées qu'à l'issue d'une action ASV. Elles prendront leur modèle dans la session ;

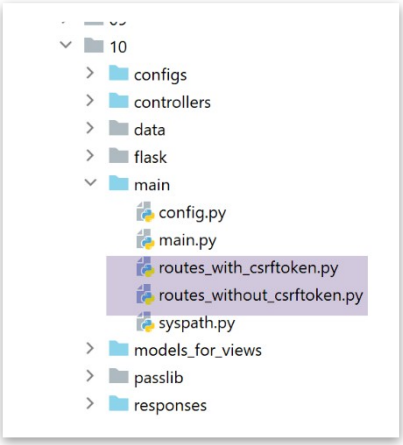
L'intérêt de cette méthode est que le navigateur affichera l'URL ASV dans son champ d'adresse. Le rafraîchissement de la page rejouera alors l'action ASV. Celle-ci ne modifie pas l'état de l'application et utilise un modèle en session. Donc la même page sera réaffichée sans effets de bord. Finalement, à cause des redirections, l'utilisateur ne verra que des URL d'actions ASV dans son navigateur et aura l'impression de naviguer de page en page ;



35.2 Implémentation

Le dossier **[impots/http-servers/10]** est obtenu initialement par recopie du dossier **[impots/http-servers/09]**. Il est ensuite modifié.

35.2.1 Les nouvelles routes



Dans les deux fichiers **[routes_with_csrftoken]** et **[routes_without_csrftoken]**, il nous faut créer les quatre routes des quatre actions ASV qui affichent les quatre vues. Les autres routes restent à l'identique.

Dans **[routes_with_csrftoken]** :

```
1.  # afficher-vue-calcul-impot
2.  @app.route('/afficher-vue-calcul-impot/<string:csrf_token>', methods=['GET'])
3.  def afficher_vue_calcul_impot(csrf_token: str) -> tuple:
4.      # on exécute le contrôleur associé à l'action
5.      return front_controller()
6.
7.  # afficher-vue-authentification
8.  @app.route('/afficher-vue-authentification/<string:csrf_token>', methods=['GET'])
9.  def afficher_vue_authentification(csrf_token: str) -> tuple:
10.     # on exécute le contrôleur associé à l'action
11.     return front_controller()
12.
13. # afficher-vue-liste-simulations
14. @app.route('/afficher-vue-liste-simulations/<string:csrf_token>', methods=['GET'])
15. def afficher_vue_liste_simulations(csrf_token: str) -> tuple:
16.     # on exécute le contrôleur associé à l'action
17.     return front_controller()
18.
19. # afficher-vue-liste_erreurs
20. @app.route('/afficher-vue-liste-erreurs/<string:csrf_token>', methods=['GET'])
21. def afficher_vue_liste_erreurs(csrf_token: str) -> tuple:
22.     # on exécute le contrôleur associé à l'action
23.     return front_controller()
```

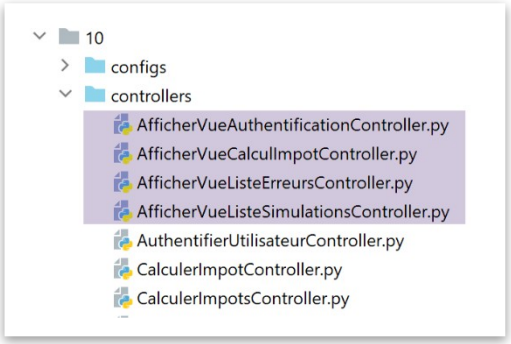
Lignes 1-23, nous avons créé quatre routes pour quatre actions ASV :

- **[/afficher-vue-authentification]**, ligne 8, affiche la vue d'authentification ;
- **[/afficher-vue-calcul-impot]**, ligne 2, affiche la vue du calcul de l'impôt ;
- **[/afficher-vue-liste-simulations]**, ligne 14, affiche la vue des simulations ;
- **[/afficher-vue-liste-erreurs]**, ligne 20, affiche la vue des erreurs inattendues ;

On fait de même dans le fichier **[routes_with_csrftoken]** des routes sans jeton :

```
1.  # routes ASV -------------------------
2.  # afficher-vue-calcul-impot
3.  @app.route('/afficher-vue-calcul-impot', methods=['GET'])
4.  def afficher_vue_calcul_impot() -> tuple:
5.      # on exécute le contrôleur associé à l'action
6.      return front_controller()
7.
8.  # afficher-vue-authentification
9.  @app.route('/afficher-vue-authentification', methods=['GET'])
10. def afficher_vue_authentification() -> tuple:
11.     # on exécute le contrôleur associé à l'action
12.     return front_controller()
13.
14. # afficher-vue-liste-simulations
15. @app.route('/afficher-vue-liste-simulations', methods=['GET'])
16. def afficher_vue_liste_simulations() -> tuple:
17.     # on exécute le contrôleur associé à l'action
18.     return front_controller()
19.
20. # afficher-vue-liste_erreurs
21. @app.route('/afficher-vue-liste-erreurs', methods=['GET'])
22. def afficher_vue_liste_erreurs() -> tuple:
23.     # on exécute le contrôleur associé à l'action
24.     return front_controller()
```

35.2.2 Les nouveaux contrôleurs



Le contrôleur **[AfficherVueAuthentificationController]** exécute l'action ASV **[/afficher-vue-authentification]** :

```
1.  from flask_api import status
2.  from werkzeug.local import LocalProxy
3.
4.  from InterfaceController import InterfaceController
5.
6.  class AfficherVueAuthentificationController(InterfaceController):
7.
8.      def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
9.          # on récupère les éléments du path
10.         params = request.path.split('/')
11.         action = params[1]
12.
13.         # changement de vue - juste un code d'état à positionner
14.         return {"action": action, "état": 1100, "réponse": ""}, status.HTTP_200_OK
```

On a dit que les actions ASV n'avaient aucun paramètre et ne modifiaient pas l'état de l'application. On se contente d'afficher la vue souhaitée en positionnant, ligne 14, un code d'état.

Le contrôleur **[AfficherVueCalculImpotController]** exécute l'action ASV **[/afficher-vue-calcul-impot]** :

```
1.  from flask_api import status
2.  from werkzeug.local import LocalProxy
3.
4.  from InterfaceController import InterfaceController
5.
6.  class AfficherVueCalculImpotController(InterfaceController):
7.
8.      def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
9.          # on récupère les éléments du path
10.         params = request.path.split('/')
11.         action = params[1]
12.
13.         # changement de vue - juste un code d'état à positionner
14.         return {"action": action, "état": 1400, "réponse": ""}, status.HTTP_200_OK
```

Le contrôleur **[AfficherVueListeSimulationsController]** exécute l'action ASV **[/afficher-vue-liste-simulations]** :

```
1.  from flask_api import status
2.  from werkzeug.local import LocalProxy
3.
4.  from InterfaceController import InterfaceController
5.
6.  class AfficherVueListeSimulationsController(InterfaceController):
7.
8.      def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
9.          # on récupère les éléments du path
10.         params = request.path.split('/')
11.         action = params[1]
12.
13.         # changement de vue - juste un code d'état à positionner
14.         return {"action": action, "état": 1200, "réponse": ""}, status.HTTP_200_OK
```

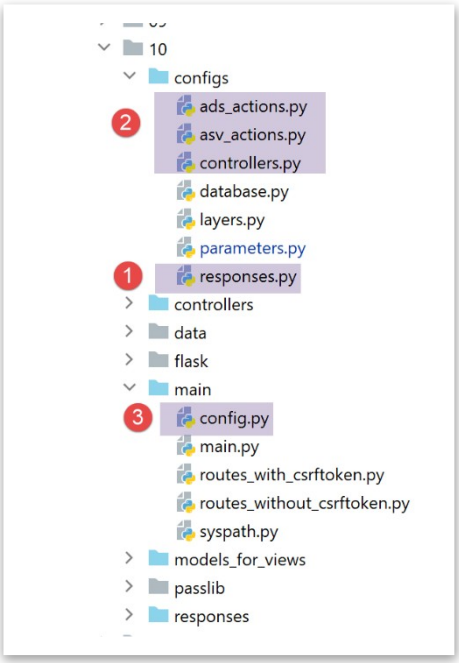
Le contrôleur **[AfficherVueListeErreursController]** exécute l'action ASV **[/afficher-vue-liste-erreurs]** :

```
1.  from flask_api import status
2.  from werkzeug.local import LocalProxy
```

```
3.
4.  from InterfaceController import InterfaceController
5.
6.  class AfficherVueListeErreursController(InterfaceController):
7.
8.      def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
9.          # on récupère les éléments du path
10.         params = request.path.split('/')
11.         action = params[1]
12.
13.         # changement de vue - juste un code d'état à positionner
14.         return {"action": action, "état": 1300, "réponse": ""}, status.HTTP_200_OK
```

Résumons les codes d'état :

- le code 1100 doit afficher la vue d'authentification ;
- le code 1400 doit afficher la vue du calcul de l'impôt ;
- le code 1200 doit afficher la vue des simulations ;
- le code 1300 doit afficher la vue des erreurs inattendues ;



35.2.3 La nouvelle configuration MVC

Parce qu'elle devenait trop importante, la configuration MVC a été éclatée sur quatre fichiers :

- **`[controllers]`** : la liste des contrôleurs C de l'application MVC ;
- **`[ads_actions]`** : liste les actions ADS (Action Do Something) ;
- **`[asv_actions]`** : liste les actions ASV (Action Show View) ;
- **`[responses]`** : liste les classes des réponses HTTP de l'application ;

Commençons par le plus simple, le fichier des réponses HTTP **`[responses]`** :

```
1.  def configure(config: dict) -> dict:
2.      # configuration de l'application MVC
3.
4.      # les réponses HTTP
5.      from HtmlResponse import HtmlResponse
6.      from JsonResponse import JsonResponse
7.      from XmlResponse import XmlResponse
8.
9.      # les différents types de réponse (json, xml, html)
10.     responses = {
11.         "json": JsonResponse(),
12.         "html": HtmlResponse(),
```

```
13.         "xml": XmlResponse()
14.     }
15.
16.     # on rend le dictionnaire des réponses HTTP
17.     return {
18.         # réponses HTTP
19.         "responses": responses,
20.     }
```

Pas de surprise.

Le fichier **[controllers]** des contrôleurs est également sans surprise. On y a simplement ajouté les nouveaux contrôleurs des actions ASV.

```
1.  def configure(config: dict) -> dict:
2.      # configuration de l'application MVC
3.
4.      # le contrôleur principal
5.      from MainController import MainController
6.
7.      # les contrôleurs d'actions ADS
8.      from AuthentifierUtilisateurController import AuthentifierUtilisateurController
9.      from CalculerImpotController import CalculerImpotController
10.     from CalculerImpotsController import CalculerImpotsController
11.     from FinSessionController import FinSessionController
12.     from GetAdminDataController import GetAdminDataController
13.     from InitSessionController import InitSessionController
14.     from ListerSimulationsController import ListerSimulationsController
15.     from SupprimerSimulationController import SupprimerSimulationController
16.     from AfficherCalculImpotController import AfficherCalculImpotController
17.
18.     # les contrôleurs d'actions ASV
19.     from AfficherVueCalculImpotController import AfficherVueCalculImpotController
20.     from AfficherVueAuthentificationController import AfficherVueAuthentificationController
21.     from AfficherVueListeErreursController import AfficherVueListeErreursController
22.     from AfficherVueListeSimulationsController import AfficherVueListeSimulationsController
23.
24.     # actions autorisées et leurs contrôleurs
25.     controllers = {
26.         # initialisation d'une session de calcul
27.         "init-session": InitSessionController(),
28.         # authentification d'un utilisateur
29.         "authentifier-utilisateur": AuthentifierUtilisateurController(),
30.         # lien vers vue calcul impot
31.         "afficher-calcul-impot": AfficherCalculImpotController(),
32.         # calcul de l'impôt en mode individuel
33.         "calculer-impot": CalculerImpotController(),
34.         # calcul de l'impôt en mode lots
35.         "calculer-impots": CalculerImpotsController(),
36.         # liste des simulations
37.         "lister-simulations": ListerSimulationsController(),
38.         # suppression d'une simulation
39.         "supprimer-simulation": SupprimerSimulationController(),
40.         # fin de la session de calcul
41.         "fin-session": FinSessionController(),
42.         # obtention des données de l'administration fiscale
43.         "get-admindata": GetAdminDataController(),
44.         # main controller
45.         "main-controller": MainController(),
46.         # affichage de la vue d'authentification
47.         "afficher-vue-authentification": AfficherVueAuthentificationController(),
48.         # affichage de la vue de calcul de l'impôt
49.         "afficher-vue-calcul-impot": AfficherVueCalculImpotController(),
50.         # affichage de la vue des simulations
51.         "afficher-vue-liste-simulations": AfficherVueListeSimulationsController(),
52.         # affichage de la vue des erreurs
53.         "afficher-vue-liste-erreurs": AfficherVueListeErreursController()
54.     }
55.
56.     # on rend la configuration des contrôeleurs
57.     return {
58.         # contrôleurs
59.         "controllers": controllers,
60.     }
```

La configuration **[asv_actions]** des actions ASV est la suivante :

```
1.  def configure(config: dict) -> dict:
2.      # configuration de l'application MVC
3.
4.      # les vues HTML et leurs modèles dépendent de l'état rendu par le contrôleur
5.      # actions ASV (Action Show view)
6.      asv = [
7.          {
8.              # vue d'authentification
9.              "états": [
10.                 1100,  # /afficher-vue-authentification
11.             ],
12.             "view_name": "views/vue-authentification.html",
13.         },
14.         {
15.             # vue du calcul de l'impôt
16.             "états": [
17.                 1400,  # /afficher-vue-calcul-impot
18.             ],
19.             "view_name": "views/vue-calcul-impot.html",
20.         },
21.         {
22.             # vue de la liste des simulations
23.             "états": [
24.                 1200,  # /afficher-vue-liste-simulations
25.             ],
26.             "view_name": "views/vue-liste-simulations.html",
27.         },
28.         {
29.             # vue de la liste des erreurs
30.             "états": [
31.                 1300,  # /afficher-vue-liste-erreurs
32.             ],
33.             "view_name": "views/vue-erreurs.html",
34.         },
35.     ]
36.
37.     # on rend la configuration ASV
38.     return {
39.         # vues et modèles
40.         "asv": asv,
41.     }
```

- le fichier **[asv_actions]** rassemble les quatre nouvelles actions dont on rappelle le fonctionnement :
  - elles n'ont aucun paramètre ;
  - elles affichent une vue précise dont le modèle est en session ;
- la liste **[asv]** des lignes 6-35, associent une vue à chaque action ASV ;

Le fichier **[ads_actions]** rassemble les actions ADS :

```
1.  def configure(config: dict) -> dict:
2.      # configuration de l'application MVC
3.
4.      # les modèles des vues
5.      from ModelForAuthentificationView import ModelForAuthentificationView
6.      from ModelForCalculImpotView import ModelForCalculImpotView
7.      from ModelForErreursView import ModelForErreursView
8.      from ModelForListeSimulationsView import ModelForListeSimulationsView
9.
10.     # actions ADS (Action Do Something)
11.     ads = [
12.         {
13.             "états": [
14.                 400,  # /fin-session réussite
15.             ],
16.             # redirection vers action ADS
17.             "to": "/init-session/html",
18.         },
19.         {
20.             "états": [
21.                 700,  # /init-session - succès
22.                 201,  # /authentifier-utilisateur échec
23.             ],
24.             # redirection vers action ASV
25.             "to": "/afficher-vue-authentification",
26.             # modèle de la vue suivante
```

```
27.             "model_for_view": ModelForAuthentificationView()
28.         },
29.         {
30.             "états": [
31.                 200,  # /authentifier-utilisateur réussite
32.                 300,  # /calculer-impot réussite
33.                 301,  # /calculer-impot échec
34.                 800,  # /afficher-calcul-impot lien
35.             ],
36.             # redirection vers action ASV
37.             "to": "/afficher-vue-calcul-impot",
38.             # modèle de la vue suivante
39.             "model_for_view": ModelForCalculImpotView()
40.         },
41.         {
42.             "états": [
43.                 500,  # /lister-simulations réussite
44.                 600,  # /supprimer-simulation réussite
45.             ],
46.             # redirection vers action ASV
47.             "to": "/afficher-vue-liste-simulations",
48.             # modèle de la vue suivante
49.             "model_for_view": ModelForListeSimulationsView()
50.         },
51.     ]
52.
53.     # vue des erreurs inattendues
54.     view_erreurs = {
55.         # redirection vers action ASV
56.         "to": "/afficher-vue-liste-erreurs",
57.         # modèle de la vue suivante
58.         "model_for_view": ModelForErreursView()
59.     }
60.
61.     # on rend la configuration MVC
62.     return {
63.         # actions ADS
64.         "ads": ads,
65.         # la vue des erreurs inattendues
66.         "view_erreurs": view_erreurs,
67.     }
```

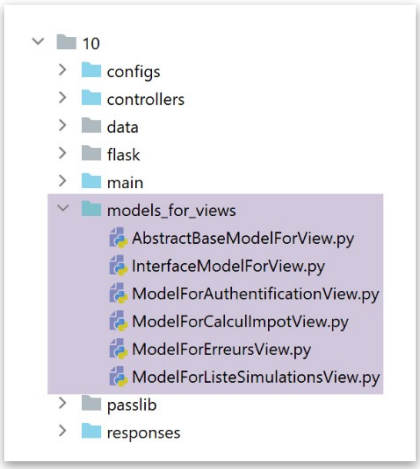
- lignes 11-51 : la liste des actions ADS (Action Do Something). On retrouve toutes les actions des versions précédentes. Leur fonctionnement a cependant changé :
  - elles n'affichent pas une vue V. Elles préparent seulement le modèle M de cette vue V ;
  - elles demandent l'affichage de la vue V via une redirection vers l'action ASV associée à la vue V ;
- les actions ADS ne mènent pas toutes à une redirection vers une action ASV : lignes 12-18, l'action ADS **[/fin-session]** mène à une redirection vers l'action ADS **[/init-session/html]**. Pour distinguer les redirections ADS -> ADS et ADS -> ASV, on peut s'aider du modèle **[model_for_view]**. Celui-ci n'existe pas pour les redirections ADS -> ADS ;

Le fichier **[main/config]** qui rassemble toutes les configurations évolue comme suit :

```
1.  def configure(config: dict) -> dict:
2.      # configuration du syspath
3.      import syspath
4.      config['syspath'] = syspath.configure(config)
5.
6.      # paramétrage de l'application
7.      import parameters
8.      config['parameters'] = parameters.configure(config)
9.
10.     # configuration de la base de données
11.     import database
12.     config["database"] = database.configure(config)
13.
14.     # instanciation des couches de l'application
15.     import layers
16.     config['layers'] = layers.configure(config)
17.
18.     # configuration MVC de la couche [web]
19.     config['mvc'] = {}
20.
21.     # configuration des contrôleurs de la couche [web]
22.     import controllers
23.     config['mvc'].update(controllers.configure(config))
```

```
24.
25.     # actions ASV (Action Show View)
26.     import asv_actions
27.     config['mvc'].update(asv_actions.configure(config))
28.
29.     # actions ADS (Action Do Something)
30.     import ads_actions
31.     config['mvc'].update(ads_actions.configure(config))
32.
33.     # configuration des réponses HTTP
34.     import responses
35.     config['mvc'].update(responses.configure(config))
36.
37.     # on rend la configuration
38.     return config
```

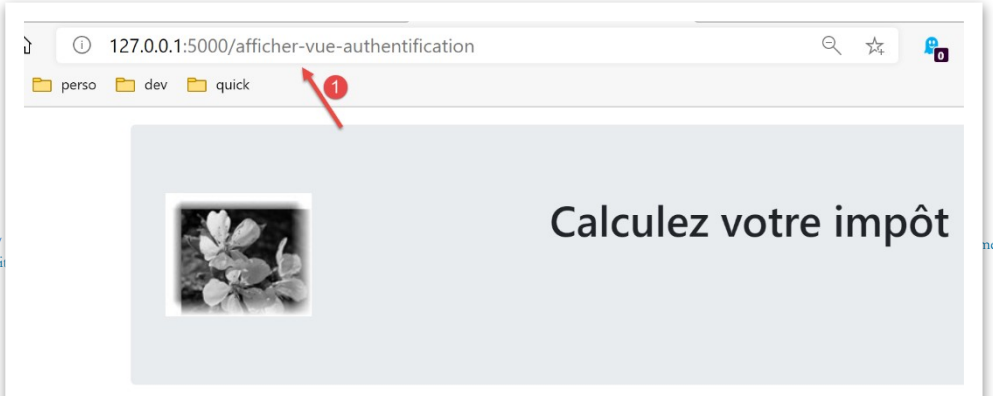
35.2.4 Les nouveaux modèles



Les modèles vont générer une nouvelle information. Prenons par exemple, le modèle de la vue d'authentification :

```
1.  from flask import Request
2.  from werkzeug.local import LocalProxy
3.
4.  from AbstractBaseModelForView import AbstractBaseModelForView
5.
6.  class ModelForAuthentificationView(AbstractBaseModelForView):
7.
8.      def get_model_for_view(self, request: Request, session: LocalProxy, config: dict, résultat: dict) -> dic
    t:
9.          # on encapsule les données de la pagé dans modèle
10.         modèle = {}
11.         …
12.
13.         # jeton csrf
14.         modèle['csrf_token'] = super().get_csrftoken(config)
15.
16.         # actions possibles à partir de la vue
17.         modèle['actions_possibles'] = ["afficher-vue-authentification", "authentifier-utilisateur"]
18.
19.         # on rend le modèle
20.         return modèle
```

- la nouveauté est ligne 17 : chaque modèle va générer une liste d'actions possibles lorsque la vue V dont il est le modèle M va s'afficher. Pour connaître ces actions, il faut revenir à la vue V. Dans le cas de la vue d'authentification :



- on voit que la vue d'authentification n'offre qu'une action, celle du bouton **[Valider]**. Cette action est **[/authentifier-utilisateur]** ;

On a écrit :

```
1.          # actions possibles à partir de la vue
2.          modèle['actions_possibles'] = ["afficher-vue-authentification", "authentifier-utilisateur"]
```

Sur la vue ci-dessus, on voit que si l'utilisateur rafraîchit la vue, l'action **[1] [/afficher-vue-authentification]** va être rejouée. Il faut donc qu'elle soit autorisée. On n'aurait pu ne pas l'autoriser auquel cas l'utilisateur aurait une erreur à chaque fois qu'il rechargerait la page. On a estimé que ce n'était pas souhaitable.

Les actions possibles sont mis dans le modèle de la vue. On sait que ce modèle va être mis en session.

On fait cela pour chacune des quatre vues. Les actions possibles sont alors les suivantes :

Vue d'authentification

```
modèle['actions_possibles'] = ["afficher-vue-authentification", "authentifier-utilisateur"]
```

Vue du calcul de l'impôt

```
# actions possibles à partir de la vue
modèle['actions_possibles'] = ["afficher-vue-calcul-impot", "calculer-impot", "lister-
simulations","fin-session"]
```

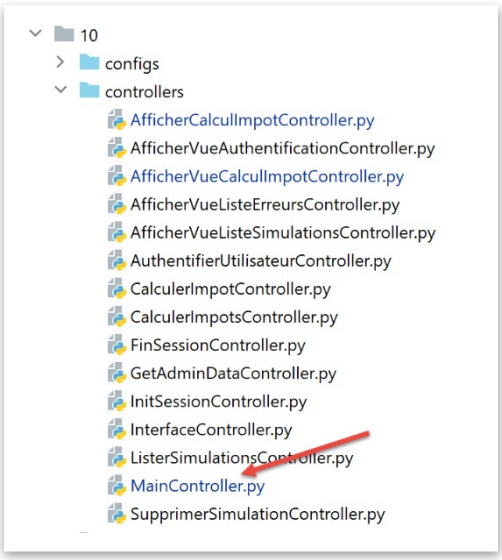
Vue de la liste des simulations

```
# actions possibles à partir de la vue
actions_possibles = ["afficher-vue-liste-simulations", "afficher-calcul-impot", "fin-session"]
if len(modèle['simulations']) != 0:
  actions_possibles.append("supprimer-simulation")
modèle['actions_possibles'] = actions_possibles
```

Vue de la liste des erreurs

```
# actions possibles à partir de la vue
modèle['actions_possibles'] = ["afficher-vue-liste-erreurs", "afficher-calcul-impot", "lister-
simulations", "fin-session"]
```

35.2.5 Le nouveau contrôleur principal



Le contrôleur **[MainController]** subit quelques modifications :

```
1.  # import des dépendances
2.  import threading
3.  import time
4.  from random import randint
5.
6.  from flask_api import status
7.  from flask_wtf.csrf import generate_csrf, validate_csrf
8.  from werkzeug.local import LocalProxy
9.  from wtforms import ValidationError
10.
11. from InterfaceController import InterfaceController
12. from Logger import Logger
13. from SendAdminMail import SendAdminMail
14.
15. def send_adminmail(config: dict, message: str):
16.     # on envoie un mail à l'administrateur de l'application
17.     config_mail = config['parameters']['adminMail']
18.     config_mail["logger"] = config['logger']
19.     SendAdminMail.send(config_mail, message)
20.
21. # contrôleur principal de l'application
22. class MainController(InterfaceController):
23.     def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
24.         # on traite la requête
25.         type_response1 = None
26.         logger = None
27.         try:
28.             # on récupère les éléments du path
29.             params = request.path.split('/')
30.
31.             # l'action est le 1er élément
32.             action = params[1]
33.
34.             # logger
35.             logger = Logger(config['parameters']['logsFilename'])
36.
37.             …
38.
39.             # l'action /afficher-vue-liste-erreurs est particulière
40.             # on ne fait aucune vérification, sinon on risque d'entrer dans une boucle infinie de redirections
41.             erreur = False
42.             if action != "afficher-vue-liste-erreurs":
43.                 # si erreur, (résultat] est le résultat à envoyer au client
44.                 (erreur, résultat, type_response1) = MainController.check_action(params, session, config)
45.
46.             # si pas d'erreur - l'action est exécutée
47.             if not erreur:
48.                 # on exécute le contrôleur associé à l'action
49.                 controller = config['mvc']['controllers'][action]
50.                 résultat, status_code = controller.execute(request, session, config)
51.
52.         except BaseException as exception:
53.             # exceptions (inattendues)
54.             résultat = {"action": action, "état": 131, "réponse": [f"{exception}"]}
55.             erreur = True
56.
57.         finally:
58.             pass
59.
60.         # erreur d'exécution
61.         if erreur:
62.             # requête erronée
63.             status_code = status.HTTP_400_BAD_REQUEST
64.
65.         if config['parameters']['with_csrftoken']:
66.             # on ajoute le csrf_token au résultat
67.             résultat['csrf_token'] = generate_csrf()
68.
69.         ….
70.
71.         # on envoie la réponse HTTP
72.         return response, status_code
73.
74.     @staticmethod
75.     def check_action(params: list, session: LocalProxy, config: dict) -> (bool, dict, str):
76.         …
77.
78.         # résultat de la méthode
79.         return erreur, résultat, type_response1
```

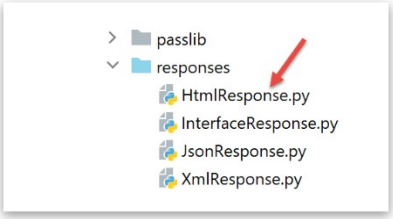
- lignes 39-44 : les premières vérifications sont faites sur l'action. Nous y reviendrons. La méthode statique **[MainController.check_action]** rend un tuple de trois éléments :
  - **[erreur]** : True si une erreur a été détectée, False sinon ;

    - **[résultat]** : un résultat d'erreur si (erreur==True), None sinon ;
    - **[type_response1]** : le type (json, xml, html, None) de la session, type trouvé en session ;
- ligne 39 : aucune vérification n'est faite si l'action est l'action ASV **[afficher-vue-liste-erreurs]** qui va afficher la liste des erreurs. En effet, si une erreur était trouvée lors de cette action, on serait dirigé de nouveau vers l'action **[/afficher-vue-liste-erreurs]** et on entrerait dans une boucle infinie de redirections ;
- la méthode statique **[check_action]** est la suivante :

```
1.      @staticmethod
2.      def check_action(params: list, session: LocalProxy, config: dict) -> (bool, dict, str):
3.          # on récupère l'action en cours
4.          action = params[1]
5.
6.          # pas d'erreur et de résultat au départ
7.          erreur = False
8.          résultat = None
9.
10.         # le type de session doit être connu avant certaines actions ADS
11.         type_response1 = session.get('typeResponse')
12.         if type_response1 is None and action != "init-session":
13.             # on note l'erreur
14.             résultat = {"action": action, "état": 101,
15.                         "réponse": ["pas de session en cours. Commencer par action [init-session]"]}
16.             erreur = True
17.
18.         # pour certaines actions ADS on doit être authentifié
19.         user = session.get('user')
20.         if user is None and action not in ["init-session",
21.                                            "authentifier-utilisateur",
22.                                            "afficher-vue-authentification"]:
23.             # on note l'erreur
24.             résultat = {"action": action, "état": 101,
25.                         "réponse": [f"action [{action}] demandée par utilisateur non authentifié"]}
26.             erreur = True
27.
28.         # à partir d'une vue seules certaines actions sont possibles
29.         if not erreur and action != "init-session":
30.             # pour l'instant pas d'actions possibles
31.             actions_possibles = None
32.             # on récupère le modèle de la future vue en session s'il existe
33.             modèle = session.get('modèle')
34.             # si on a trouvé un modèle, on récupère ses actions possibles
35.             if modèle:
36.                 actions_possibles = modèle.get('actions_possibles')
37.             # si on a une liste d'actions possibles, on vérifie que l'action en cours en fait partie
38.             if actions_possibles and action not in actions_possibles:
39.                 # on note l'erreur
40.                 résultat = {"action": action, "état": 151,
41.                             "réponse": [f"action [{action}] incorrecte dans l'environnement actuel"]}
42.                 erreur = True
43.
44.
45.         # erreur ?
46.         if not erreur and config['parameters']['with_csrftoken']:
47.             # on vérifie la validité du token csrf
48.             # le csrf_token est le dernier élément du path
49.             csrf_token = params.pop()
50.             try:
51.                 # une exception sera lancée si le csrf_token n'est pas valide
52.                 validate_csrf(csrf_token)
53.             except ValidationError as exception:
54.                 # csrf token invalide
55.                 résultat = {"action": action, "état": 121, "réponse": [f"{exception}"]}
56.                 # on note l'erreur
57.                 erreur = True
58.
59.         # résultat de la méthode
60.         return erreur, résultat, type_response1
```

- ligne 2 : la méthode **[check_action]** fait plusieurs vérifications sur la validité de l'action en cours ;
- lignes 6-26, 45-57 : les versions précédentes faisaient déjà ces vérifications ;
- lignes 28-42 : on ajoute une nouvelle vérification. On vérifie si l'action en cours est possible dans l'état actuel de l'application. Si l'action en cours n'est pas possible, on génère un état 151 (ligne 40) qui nous assure que l'action actuelle va être redirigée vers la vue des erreurs inattendues ;

35.2.6 La nouvelle réponse HTML



Les modifications en cours ne concernent que les sessions HTML. Les sessions jSON ou XML ne sont pas impactées. La classe **[HtmlResponse]** évolue de la façon suivante :

```
1.  # dépendances
2.
3.  from flask import make_response, redirect, render_template
4.  from flask.wrappers import Response
5.  from flask_api import status
6.  from flask_wtf.csrf import generate_csrf
7.  from werkzeug.local import LocalProxy
8.
9.  from InterfaceResponse import InterfaceResponse
10.
11. class HtmlResponse(InterfaceResponse):
12.
13.     def build_http_response(self, request: LocalProxy, session: LocalProxy, config: dict, status_code: int,
14.                             résultat: dict) -> (Response, int):
15.         # la réponse HTML dépend du code d'état rendu par le contrôleur
16.         état = résultat["état"]
17.
18.         # on cherche si l'état a été produit par une action ASV
19.         # auquel cas, il faut afficher une vue
20.         asv_configs = config['mvc']["asv"]
21.         trouvé = False
22.         i = 0
23.         # on parcourt la liste des vues
24.         nb_views = len(asv_configs)
25.         while not trouvé and i < nb_views:
26.             # vue n° i
27.             asv_config = asv_configs[i]
28.             # états associés à la vue n° i
29.             états = asv_config["états"]
30.             # est-ce que l'état cherché se trouve dans les états associés à la vue n° i
31.             if état in états:
32.                 trouvé = True
33.             else:
34.                 # vue suivante
35.                 i += 1
36.
37.         # trouvé ?
38.         if trouvé:
39.             # il s'agit d'une action ASV - il faut afficher une vue dont le modèle est déjà en session
40.             # on génère le code HTML de la réponse
41.             html = render_template(asv_config["view_name"], modèle=session['modèle'])
42.             # on construit la réponse HTTP
43.             response = make_response(html)
44.             response.headers['Content-Type'] = 'text/html; charset=utf-8'
45.             # on rend le résultat
46.             return response, status_code
47.
48.         # pas trouvé - il s'agit d'un code d'état d'une action ADS
49.         # celle-ci va être suivie d'une redirection
50.         redirected = False
51.         for ads in config['mvc']['ads']:
52.             # états nécessitant une redirection
53.             états = ads["états"]
54.             if état in états:
55.                 # il y a redirection
56.                 redirected = True
57.                 break
58.         # dictionnaire de redirection pour le cas des erreurs inattendues
59.         if not redirected:
60.             ads = config['mvc']['view_erreurs']
61.
```

```
62.         # est-ce une redirection vers une action ASD ou ASV ?
63.         # s'il y a un modèle, alors il s'agit d'une redirection vers une action ASV
64.         # il faut alors calculer le modèle de la vue V qui sera affichée par l'action ASV
65.         model_for_view = ads.get("model_for_view")
66.         if model_for_view:
67.             # calcul du modèle de la vue suivante
68.             modèle = model_for_view.get_model_for_view(request, session, config, résultat)
69.             # le modèle est mis en session pour la vue suivante
70.             session['modèle'] = modèle
71.
72.             # maintenant il faut générer l'URL de redirection sans oublier le jeton CSRF s'il est demandé
73.         if config['parameters']['with_csrftoken']:
74.             csrf_token = f"/{generate_csrf()}"
75.         else:
76.             csrf_token = ""
77.
78.         # réponse de redirection
79.         return redirect(f"{ads['to']}{csrf_token}"), status.HTTP_302_FOUND
```

- lignes 18-35 : on cherche si l'état produit par la dernière action exécutée est celui d'une action ASV ;
- lignes 36-46 : si oui, la vue V associée à l'action ASV est affichée avec pour modèle le modèle **trouvé en session** associé à la clé **`['modèle']`** ;
- lignes 48-60 : lorsqu'on arrive là, on sait que l'état produit par la dernière action exécutée est celui d'une action ADS. Il va alors provoquer une redirection. On cherche dans le fichier de configuration, la définition de celle-ci ;
- ligne 62 : lorsqu'on est là, on a la configuration de la redirection à faire. Il y a deux cas :
  - il s'agit d'une redirection vers une autre action ADS. Il n'y a alors pas de modèle de vue à calculer ;
  - il s'agit d'une redirection vers une action ASV. Il y a alors un modèle de vue à calculer (lignes 67-68). Ce modèle est ensuite mis en session (ligne 70) ;
- lignes 72-76 : on calcule l'URL de redirection ;
- lignes 78-79 : on envoie la réponse de redirection au client ;

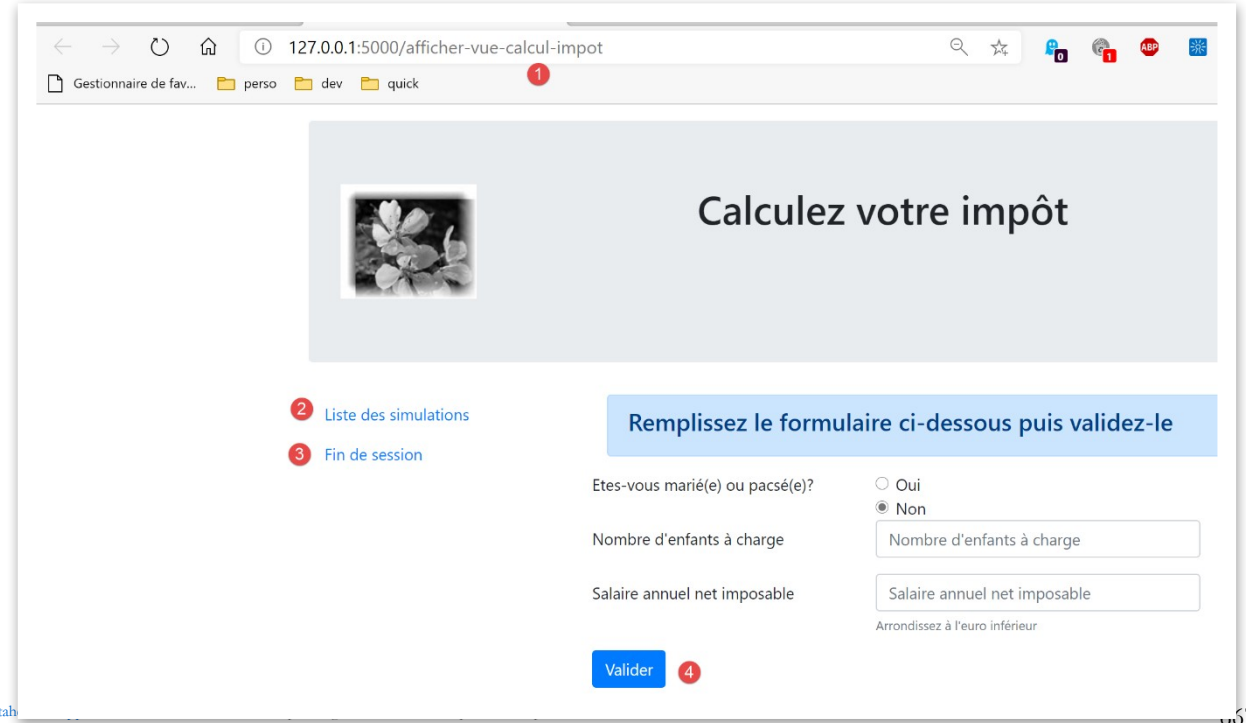
35.3 Tests

Faites les tests suivants avec un navigateur :

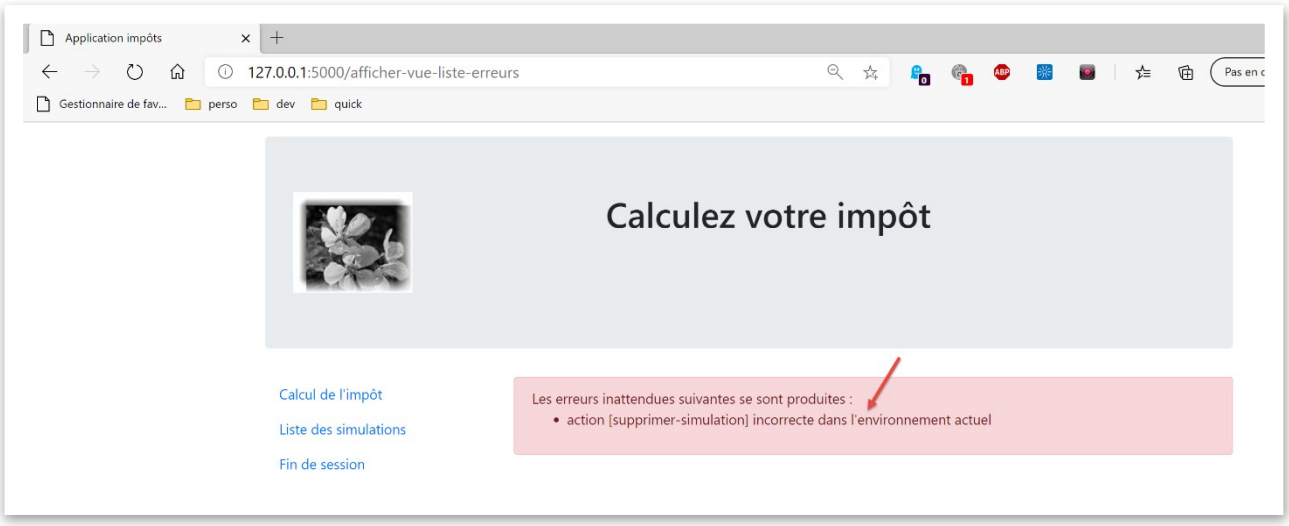
- utilisez normalement l'application. Vérifiez que les seules URL affichées par le navigateur sont des URL ASV **`[/afficher-vue-nom_de_la_vue]`** ;
- rafraîchissez les pages (F5) et constatez que la même page se réaffiche alors. Il n'y a aucun effet de bord ;

Par ailleurs, utilisez le client **`[impots/http-clients/09]`**. Comme les modifications faites concernent uniquement les sessions HTML, les clients **`[main, main2, main3, Test1HttpClientDaoWithSession, Test2HttpClientDaoWithSession]`** doivent continuer à fonctionner.

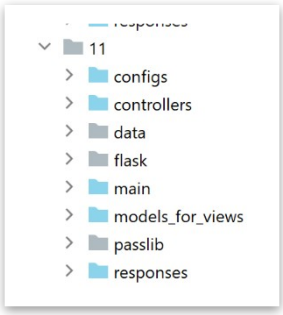
Maintenant voyons un cas d'action impossible. La vue suivante est affichée :



A la place de **[1]**, on tape l'URL **[/supprimer-simulation/1]**. L'action **[/supprimer-simulation]** ne fait pas partie des actions proposées par la vue qui sont les actions 1-4. Elle va donc être refusée. La réponse du serveur est la suivante :



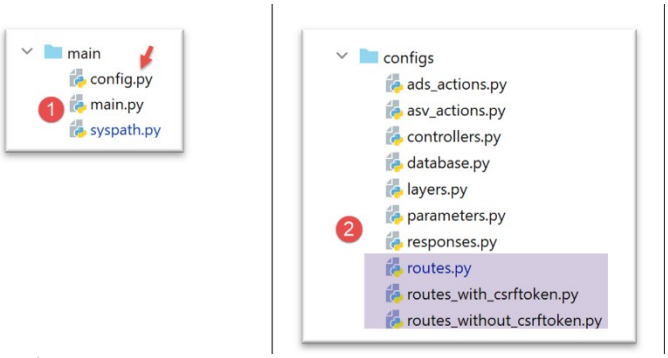
36 Exercice d'application : version 16



36.1 Introduction

Les URL de notre application sont pour l'instant de la forme **[/action/param1/param2/…]**. Nous voudrions pouvoir préfixer ces URL. Par exemple avec le préfixe **[/do]**, nous aurions des URL de la forme **[/do/action/param1/param2/…]**.

36.2 La nouvelle configuration des routes



- en **[2]**, le calcul des routes va être modifié ;
- en **[1]**, le fichier **[config]** est modifié pour refléter ce changement ;

La configuration **[config]** devient la suivante :

```
1.  def configure(config: dict) -> dict:
2.      # configuration du syspath
3.      import syspath
4.      config['syspath'] = syspath.configure(config)
5.
6.      # paramétrage de l'application
7.      import parameters
8.      config['parameters'] = parameters.configure(config)
9.
10.     # configuration de la base de données
11.     import database
12.     config["database"] = database.configure(config)
13.
14.     # instanciation des couches de l'application
15.     import layers
16.     config['layers'] = layers.configure(config)
17.
18.     # configuration MVC de la couche [web]
19.     config['mvc'] = {}
20.
21.     # configuration des contrôleurs de la couche [web]
22.     import controllers
23.     config['mvc'].update(controllers.configure(config))
24.
25.     # actions ASV (Action Show View)
26.     import asv_actions
27.     config['mvc'].update(asv_actions.configure(config))
28.
```

```
29.     # actions ADS (Action Do Something)
30.     import ads_actions
31.     config['mvc'].update(ads_actions.configure(config))
32.
33.     # configuration des réponses HTTP
34.     import responses
35.     config['mvc'].update(responses.configure(config))
36.
37.     # configuration des routes
38.     import routes
39.     routes.configure(config)
40.
41.     # on rend la configuration
42.     return config
```

- lignes 37-39 : le module **[routes]** (ligne 38) s'occupe de configurer les routes (ligne 39) ;

Le fichier des routes sans jeton CSRF **[configs/routes_without_csrftoken]** évolue de la façon suivante :

```
1.  # dépendances
2.
3.  from flask import redirect, request, session, url_for
4.  from flask_api import status
5.
6.  # configuration application
7.  config = {}
8.
9.  # le front controller
10. def front_controller() -> tuple:
11.     # on fait suivre la requête au contrôleur principal
12.     main_controller = config['mvc']['controllers']['main-controller']
13.     return main_controller.execute(request, session, config)
14.
15. # racine de l'application
16. def index() -> tuple:
17.     # redirection vers /init-session/html
18.     return redirect(url_for("init_session", type_response="html"), status.HTTP_302_FOUND)
19.
20. # init-session
21. def init_session(type_response: str) -> tuple:
22.     # on exécute le contrôleur associé à l'action
23.     return front_controller()
24.
25. # authentifier-utilisateur
26. def authentifier_utilisateur() -> tuple:
27.     # on exécute le contrôleur associé à l'action
28.     return front_controller()
29.
30. # calculer-impot
31. def calculer_impot() -> tuple:
32.     # on exécute le contrôleur associé à l'action
33.     return front_controller()
34.
35. …
```

Le fichier a été débarrassé de ses routes. Il ne reste que les fonctions associées à celles-ci.

Les fichier des routes avec jeton CSRF **[configs/routes_with_csrftoken]** subit le même sort :

```
1.  # dépendances
2.
3.  from flask import redirect, request, session, url_for
4.  from flask_api import status
5.  from flask_wtf.csrf import generate_csrf
6.
7.  # configuration
8.  config = {}
9.
10. # le front controller
11. def front_controller() -> tuple:
12.     # on fait suivre la requête au contrôleur principal
13.     main_controller = config['mvc']['controllers']['main-controller']
14.     return main_controller.execute(request, session, config)
15.
16. # racine de l'application
17. def index() -> tuple:
18.     # redirection vers /init-session/html
19.     return redirect(url_for("init_session", type_response="html", csrf_token=generate_csrf()), status.HTTP_302_FOUND)
```

```
20.
21.  # init-session
22.  def init_session(type_response: str, csrf_token: str) -> tuple:
23.      # on exécute le contrôleur associé à l'action
24.      return front_controller()
25.
26.  # authentifier-utilisateur
27.  def authentifier_utilisateur(csrf_token: str) -> tuple:
28.      # on exécute le contrôleur associé à l'action
29.      return front_controller()
30.
31.  …
32.
33.  # init-session-without-csrftoken pour les clients json et xml
34.  def init_session_without_csrftoken(type_response: str) -> tuple:
35.      # redirection vers /init-session/type_response
36.      return redirect(url_for("init_session", type_response=type_response, csrf_token=generate_csrf()),
37.                      status.HTTP_302_FOUND)
```

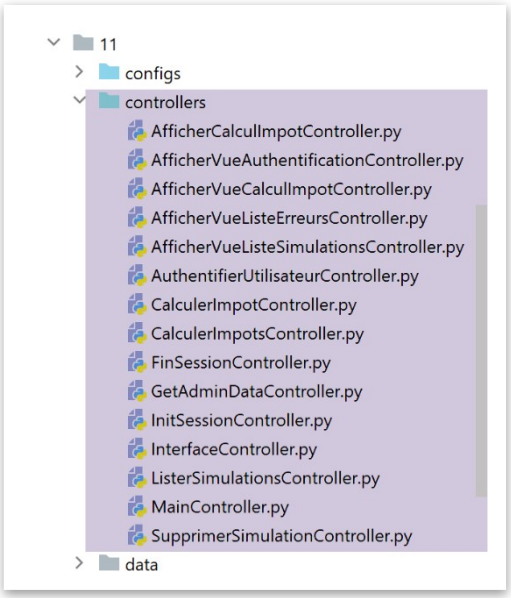
Les routes sont calculées par le module **[configs/routes]** suivant :

```
1.   from flask import Flask
2.
3.   def configure(config: dict):
4.       # paramétrage des routes
5.
6.       # application Flask
7.       app = Flask(__name__, template_folder="../flask/templates", static_folder="../flask/static")
8.       config['app'] = app
9.
10.      # import des routes de l'application web
11.      if config['parameters']['with_csrftoken']:
12.          import routes_with_csrftoken as routes
13.      else:
14.          import routes_without_csrftoken as routes
15.
16.      # on injecte la configuration dans les routes
17.      routes.config = config
18.
19.      # le préfixe des URL de l'application
20.      prefix_url = config["parameters"]["prefix_url"]
21.
22.      # jeton CSRF
23.      with_csrftoken = config["parameters"]['with_csrftoken']
24.      if with_csrftoken:
25.          csrftoken_param = f"/<string:csrf_token>"
26.      else:
27.          csrftoken_param = ""
28.
29.      # les routes de l'application Flask
30.      # racine de l'application
31.      app.add_url_rule(f'{prefix_url}/', methods=['GET'],
32.                       view_func=routes.index)
33.
34.      # init-session
35.      app.add_url_rule(f'{prefix_url}/init-session/<string:type_response>{csrftoken_param}', methods=['GET'],
36.                       view_func=routes.init_session)
37.
38.      # init-session-without-csrftoken
39.      if with_csrftoken:
40.          app.add_url_rule(f'{prefix_url}/init-session-without-csrftoken/<string:type_response>',
41.                           methods=['GET'],
42.                           view_func=routes.init_session_without_csrftoken)
43.
44.      # authentifier-utilisateur
45.      app.add_url_rule(f'{prefix_url}/authentifier-utilisateur{csrftoken_param}', methods=['POST'],
46.                       view_func=routes.authentifier_utilisateur)
47.
48.      # calculer-impot
49.      app.add_url_rule(f'{prefix_url}/calculer-impot{csrftoken_param}', methods=['POST'],
50.                       view_func=routes.calculer_impot)
51.
52.      # calcul de l'impôt par lots
53.      app.add_url_rule(f'{prefix_url}/calculer-impots{csrftoken_param}', methods=['POST'],
54.                       view_func=routes.calculer_impots)
55.
56.      # lister-simulations
57.      app.add_url_rule(f'{prefix_url}/lister-simulations{csrftoken_param}', methods=['GET'],
58.                       view_func=routes.lister_simulations)
```

```
59.
60.     # supprimer-simulation
61.     app.add_url_rule(f'{prefix_url}/supprimer-simulation/<int:numero>{csrftoken_param}', methods=['GET'],
62.                      view_func=routes.supprimer_simulation)
63.
64.     # fin-session
65.     app.add_url_rule(f'{prefix_url}/fin-session{csrftoken_param}', methods=['GET'],
66.                      view_func=routes.fin_session)
67.
68.     # afficher-calcul-impot
69.     app.add_url_rule(f'{prefix_url}/afficher-calcul-impot{csrftoken_param}', methods=['GET'],
70.                      view_func=routes.afficher_calcul_impot)
71.
72.     # get-admindata
73.     app.add_url_rule(f'{prefix_url}/get-admindata{csrftoken_param}', methods=['GET'],
74.                      view_func=routes.get_admindata)
75.
76.     # afficher-vue-calcul-impot
77.     app.add_url_rule(f'{prefix_url}/afficher-vue-calcul-impot{csrftoken_param}', methods=['GET'],
78.                      view_func=routes.afficher_vue_calcul_impot)
79.
80.     # afficher-vue-authentification
81.     app.add_url_rule(f'{prefix_url}/afficher-vue-authentification{csrftoken_param}', methods=['GET'],
82.                      view_func=routes.afficher_vue_authentification)
83.
84.     # afficher-vue-liste-simulations
85.     app.add_url_rule(f'{prefix_url}/afficher-vue-liste-simulations{csrftoken_param}', methods=['GET'],
86.                      view_func=routes.afficher_vue_liste_simulations)
87.
88.     # afficher-vue-liste_erreurs
89.     app.add_url_rule(f'{prefix_url}/afficher-vue-liste-erreurs{csrftoken_param}', methods=['GET'],
90.                      view_func=routes.afficher_vue_liste_erreurs)
91.
92.     # cas particulier
93.     if with_csrftoken:
94.         # init-session-without-csrftoken pour les clients json et xml
95.         app.add_url_rule(f'{prefix_url}/init-session-without-csrftoken', methods=['GET'],
96.                          view_func=routes.init_session_without_csrftoken)
```

- lignes 6-8 : l'application Flask est créée et mise dans la configuration ;
- lignes 10-14 : on importe le fichier de routes qui convient à la situation. On se rappelle que le fichier importé est en fait dépourvu de routes. Il ne contient que les fonctions associées à celles-ci ;
- ligne 17 : les fonctions associées aux routes ont besoin de connaître la configuration de l'application ;
- ligne 20 : on note le préfixe des URL. Celui-ci peut être vide ;
- lignes 22-27 : les routes avec jeton CSRF ont un paramètre supplémentaire que celles qui n'en ont pas. Pour gérer cette différence, on utilise la variable **`[csrftoken_param]`** :
  - elle contient la chaîne vide s'il n'y a pas de jeton CSRF dans les routes ;
  - elle contient la chaîne **`[/<string:csrf_token>]`** s'il y a un jeton CSRF ;
- lignes 29-96 : chaque route de l'application est associée à une fonction du fichier des routes importé lignes 10-14 ;

36.3 Les nouveaux contrôleurs



Dans la version précédente, tous les contrôleurs obtenaient l'action en cours de traitement de la façon suivante :

```
1.      def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
2.          # on récupère les éléments du path
3.          params = request.path.split('/')
4.          action = params[1]
```

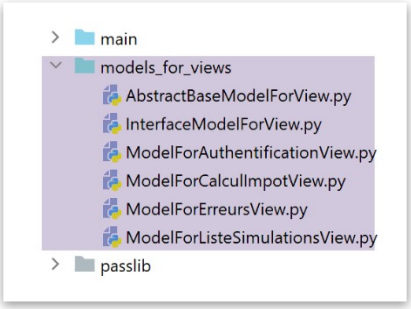
Ce code ne fonctionne plus s'il y a un préfixe, par exemple **[/do/lister-simulations]**. Dans ce cas, l'action, ligne 3 serait **[do]** et donc serait incorrecte.

On transforme ce code de la façon suivante :

```
1.      def execute(self, request: LocalProxy, session: LocalProxy, config: dict) -> (dict, int):
2.          # on récupère les éléments du path
3.          prefix_url = config["parameters"]["prefix_url"]
4.          params = request.path[len(prefix_url):].split('/')
5.          action = params[1]
```

On fait ça dans tous les contrôleurs.

36.4 Les nouveaux modèles



Dans la version précédente, chaque classe de modèle générait un modèle de la façon suivante :

```
1.  def get_model_for_view(self, request: Request, session: LocalProxy, config: dict, résultat: dict) -> dict:
2.          # on encapsule les données de la pagé dans modèle
3.          modèle = {}
4.          # état de l'application
5.          état = résultat["état"]
6.          # le modèle dépend de l'état
```

```
        …

        # jeton csrf
        modèle['csrf_token'] = super().get_csrftoken(config)

        # actions possibles à partir de la vue
        modèle['actions_possibles'] = […]

        # on rend le modèle
        return modèle
```

Désormais le modèle aura une clé supplémentaire, le préfixe des URL :

```
def get_model_for_view(self, request: Request, session: LocalProxy, config: dict, résultat: dict) -> dict:
        # on encapsule les données de la pagé dans modèle
        modèle = {}
        # état de l'application
        état = résultat["état"]
        # le modèle dépend de l'état
        …

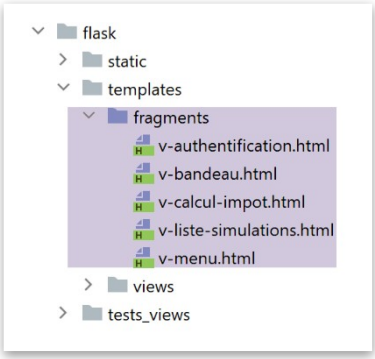
        # jeton csrf
        modèle['csrf_token'] = super().get_csrftoken(config)

        # actions possibles à partir de la vue
        modèle['actions_possibles'] = […]

        # préfixe des URL
        modèle["prefix_url"] = config["parameters"]["prefix_url"]

        # on rend le modèle
        return modèle
```

36.5 Les nouveaux fragments



Tous les fragments contenant des URL doivent être modifiés.

Le fragment `[v-authentification]`

```
<!-- formulaire HTML - on poste ses valeurs avec l'action [authentifier-utilisateur] -->
<form method="post" action="{{modèle.prefix_url}}/authentifier-utilisateur{{modèle.csrf_token}}">

    <!-- titre -->
    <div class="alert alert-primary" role="alert">
        <h4>Veuillez vous authentifier</h4>
    </div>
…

</form>
```

Le fragment `[v-calcul-impot]`

```
<!-- formulaire HTML posté -->
<form method="post" action="{{modèle.prefix_url}}/calculer-impot{{modèle.csrf_token}}">
    <!-- message sur 12 colonnes sur fond bleu -->
    …

```

```
6.  </form>
```

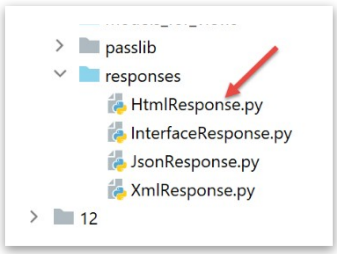
Le fragment [v-liste-simulations]

```
1.  …
2.
3.  {% if modèle.simulations is defined and modèle.simulations|length!=0 %}
4.  …
5.
6.  <!-- tableau des simulations -->
7.  <table class="table table-sm table-hover table-striped">
8.      …
9.      <tr>
10.         <th scope="row">{{simulation.id}}</th>
11.         <td>{{simulation.marié}}</td>
12.         <td>{{simulation.enfants}}</td>
13.         <td>{{simulation.salaire}}</td>
14.         <td>{{simulation.impôt}}</td>
15.         <td>{{simulation.surcôte}}</td>
16.         <td>{{simulation.décôte}}</td>
17.         <td>{{simulation.réduction}}</td>
18.         <td>{{simulation.taux}}</td>
19.         <td><a href="{{modèle.prefix_url}}/supprimer-simulation/{{simulation.id}}{{modèle.csrf_token}}">Supprimer</a></td>
20.     </tr>
21.     {% endfor %}
22.     </tr>
23.     </tbody>
24. </table>
25. {% endif %}
```

Le fragment [v-menu]

```
1.  <!-- menu Bootstrap -->
2.  <nav class="nav flex-column">
3.      <!-- affichage d'une liste de liens HTML -->
4.      {% for optionMenu in modèle.optionsMenu %}
5.      <a class="nav-link" href="{{modèle.prefix_url}}{{optionMenu.url}}
    {{modèle.csrf_token}}">{{optionMenu.text}}</a>
6.      {% endfor %}
7.  </nav>
```

36.6 La nouvelle réponse HTML



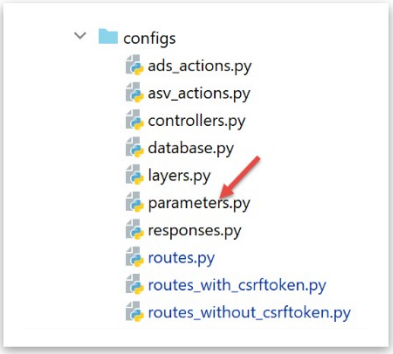
Dans la précédente version le code de la réponse HTML se terminait par une redirection :

```
1.  class HtmlResponse(InterfaceResponse):
2.
3.      def build_http_response(self, request: LocalProxy, session: LocalProxy, config: dict, status_code: int,
4.                              résultat: dict) -> (Response, int):
5.          # la réponse HTML dépend du code d'état rendu par le contrôleur
6.          état = résultat["état"]
7.
8.          …
9.
10.         # maintenant il faut générer l'URL de redirection sans oublier le jeton CSRF s'il est demandé
11.         if config['parameters']['with_csrftoken']:
12.             csrf_token = f"/{generate_csrf()}"
13.         else:
14.             csrf_token = ""
15.
16.         # réponse de redirection
17.         return redirect(f"{ads['to']}{csrf_token}"), status.HTTP_302_FOUND
```

Ce code devient maintenant le suivant :

```
1.  def build_http_response(self, request: LocalProxy, session: LocalProxy, config: dict, status_code: int,
2.                              résultat: dict) -> (Response, int):
3.          …
4.
5.          # maintenant il faut générer l'URL de redirection sans oublier le jeton CSRF s'il est demandé
6.          if config['parameters']['with_csrftoken']:
7.              csrf_token = f"/{generate_csrf()}"
8.          else:
9.              csrf_token = ""
10.
11.         # réponse de redirection
12.         return redirect(f"{config['parameters']['prefix_url']}{ads['to']}{csrf_token}"), status.HTTP_302_FOUND
```

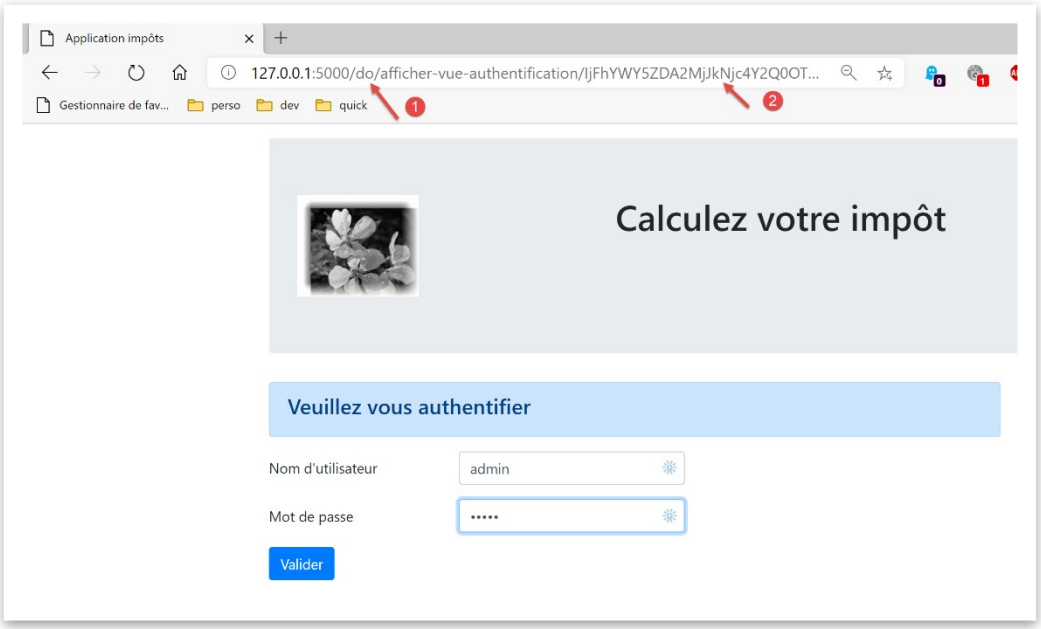
36.7 Tests



Mettons le préfixe suivant dans le fichier de configuration **[parameters]** :

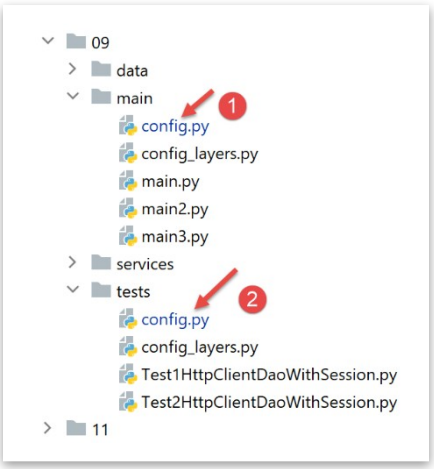
```
1.  …
2.          # token csrf
3.          "with_csrftoken": True,
4.          # bases gérées MySQL (mysql), PostgreSQL (pgres)
5.          "databases": ["mysql", "pgres"],
6.          # préfixe des URL de l'application
7.          # mettre la chaîne vide si on ne veut pas de préfixe ou /préfixe sinon
8.          "prefix_url": "/do",
9.  …
```

Lançons l'application puis demandons l'URL **[http://localhost:5000/do]**, la réponse est la suivante :



- en **[1]**, le préfixe de l'URL ;
- en **[2]**, le jeton CSRF ;

L'application peut être également testée avec les tests console de **[http-clients/09]** :



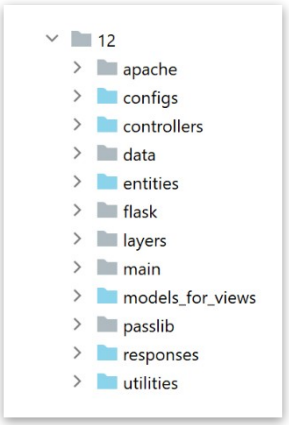
Dans les fichiers de configuration **[1]** et **[2]**, le préfixe des URL doit être inclus dans l'URL du serveur :

```
1.          "server": {
2.              # "urlServer": "http://127.0.0.1:5000",
3.              "urlServer": "http://127.0.0.1:5000/do",
4.              "user": {
5.                  "login": "admin",
6.                  "password": "admin"
7.              },
8.              "url_services": {
9.                  …
10.             }
```

- ligne 3 : l'URL du serveur inclut désormais le préfixe des URL ;

Cette modification faite, tous les tests console doivent fonctionner.

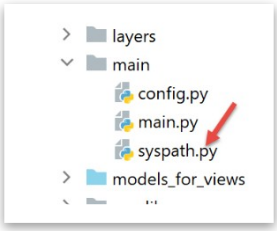
37 Exercice d'application : version 17



Cette nouvelle version amène les évolutions suivantes :

- elle va être portée sur un serveur Apache / Windows ;
- pour réaliser ce portage, la version 17 contient toutes les dépendances dont elle a besoin dans son dossier **[impots / http-servers/ 12]**. On rappelle que les versions précédentes allaient chercher leurs dépendances dans différents dossiers de l'ensemble du projet **[python-flask-2020]** ;

37.1 Relocalisation des dépendances de l'application



On rappelle que la gestion des dépendances de l'application est faite dans le script **[syspath]**. Dans la version précédente, ce script était le suivant :

```
1.  def configure(config: dict) -> dict:
2.      import os
3.
4.      # dossier de ce fichier
5.      script_dir = os.path.dirname(os.path.abspath(__file__))
6.
7.      # chemin racine
8.      root_dir = "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020"
9.
10.     # dépendances
11.     absolute_dependencies = [
12.         # dossiers du projet
13.         # BaseEntity, MyException
14.         f"{root_dir}/classes/02/entities",
15.         # InterfaceImpôtsDao, InterfaceImpôtsMétier, InterfaceImpôtsUi
16.         f"{root_dir}/impots/v04/interfaces",
17.         # AbstractImpôtsdao, ImpôtsConsole, ImpôtsMétier
18.         f"{root_dir}/impots/v04/services",
19.         # ImpotsDaoWithAdminDataInDatabase
20.         f"{root_dir}/impots/v05/services",
21.         # AdminData, ImpôtsError, TaxPayer
22.         f"{root_dir}/impots/v04/entities",
23.         # Constantes, tranches
24.         f"{root_dir}/impots/v05/entities",
25.         # Logger, SendAdminMail
26.         f"{root_dir}/impots/http-servers/02/utilities",
27.         # dossier du script principal
28.         script_dir,
29.         # configs [database, layers, parameters, controllers, views]
30.         f"{script_dir}/../configs",
```

```
31.         # contrôleurs
32.         f"{script_dir}/../controllers",
33.         # réponses HTTP
34.         f"{script_dir}/../responses",
35.         # modèles des vues
36.         f"{script_dir}/../models_for_views",
37.     ]
38.
39.     # on fixe le syspath
40.     from myutils import set_syspath
41.     set_syspath(absolute_dependencies)
42.
43.     # on rend la configuration
44.     return {
45.         "root_dir": root_dir,
46.         "script_dir": script_dir
47.     }
```

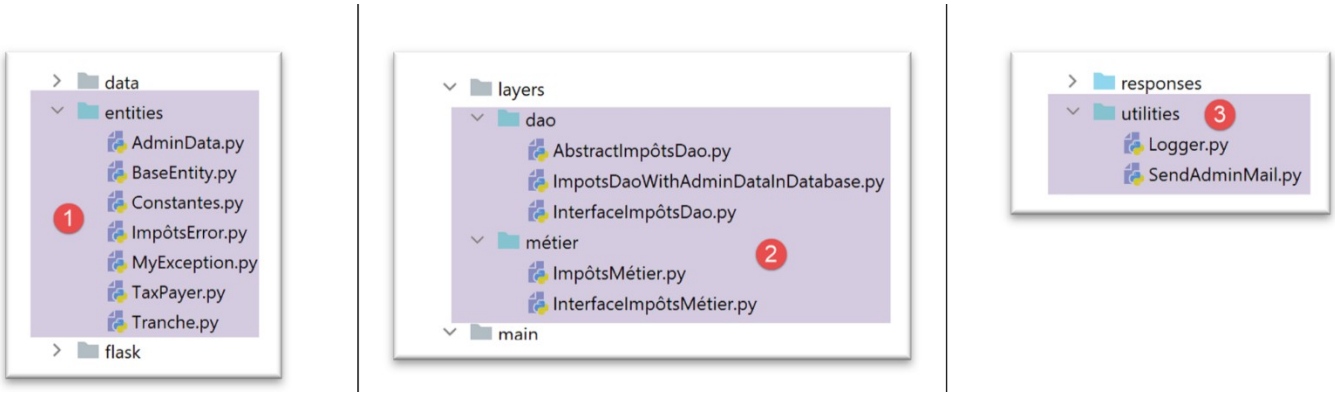
Il faut relocaliser toutes les dépendances dont le nom absolu dépend de la variable **[root_dir]** de la ligne 8, ç-à-d les lignes 13-26.

Le script **[syspath]** de la nouvelle version sera le suivant :

```
1.  def configure(config: dict) -> dict:
2.      import os
3.
4.      # dossier de ce fichier
5.      script_dir = os.path.dirname(os.path.abspath(__file__))
6.
7.      # dépendances
8.      absolute_dependencies = [
9.          # entités de l'application
10.         f"{script_dir}/../entities",
11.         # couche [dao]
12.         f"{script_dir}/../layers/dao",
13.         # couche [métier]
14.         f"{script_dir}/../layers/métier",
15.         # utilitaires
16.         f"{script_dir}/../utilities",
17.         # dossier du script principal
18.         script_dir,
19.         # configs [database, layers, parameters, controllers, views]
20.         f"{script_dir}/../configs",
21.         # contrôleurs
22.         f"{script_dir}/../controllers",
23.         # réponses HTTP
24.         f"{script_dir}/../responses",
25.         # modèles des vues
26.         f"{script_dir}/../models_for_views",
27.     ]
28.
29.     # on fixe le syspath
30.     import sys
31.     # on lui ajoute les dépendances absolues du projet
32.     for directory in absolute_dependencies:
33.         # on vérifie l'existence du dossier
34.         existe = os.path.exists(directory) and os.path.isdir(directory)
35.         if not existe:
36.             # on avertit le développeur
37.             raise BaseException(f"[set_syspath] le dossier du Python Path [{directory}] n'existe pas")
38.         else:
39.             # on insère le dossier au début du syspath
40.             sys.path.insert(0, directory)
41.
42.     # on rend la configuration
43.     return {
44.         "script_dir": script_dir,
45.     }
```

- lignes 8-27 : toutes les dépendances sont désormais relatives à la variable **[script_dir]** de la ligne 5 ;
- lignes 42-45 : la variable **[root_dir]** a disparu de la configuration du syspath ;
- ligne 10 : les entités de l'application sont dans le dossier **[entities] [1]** ;
- ligne 12 : la couche **[dao]** est dans le dossier **[layers/dao] [2]** ;
- ligne 14 : la couche **[métier]** est dans le dossier **[layers/métier] [2]** ;
- ligne 16 : les utilitaires **[Logger, SendMail]** sont dans le dossier **[utilities] [3]** ;

- lignes 29-40 : on calcule le Python Path de l'application sans importer le module **[myutils]** ;



37.2 Tests

A ce stade, la version 17 doit fonctionner. Vérifiez-le.

37.3 Portage d'une application Python / Flask sur un serveur Apache / Windows

37.3.1 Sources

Pour réaliser le portage d'une application Flask sur Apache / Windows, j'ai dû chercher sur Internet. Voici le lien qui m'a aidé à démarrer : [https://medium.com/@madumalt/flask-app-deployment-in-windows-apache-server-mod-wsgi-82e1cfeeb2ed] ;

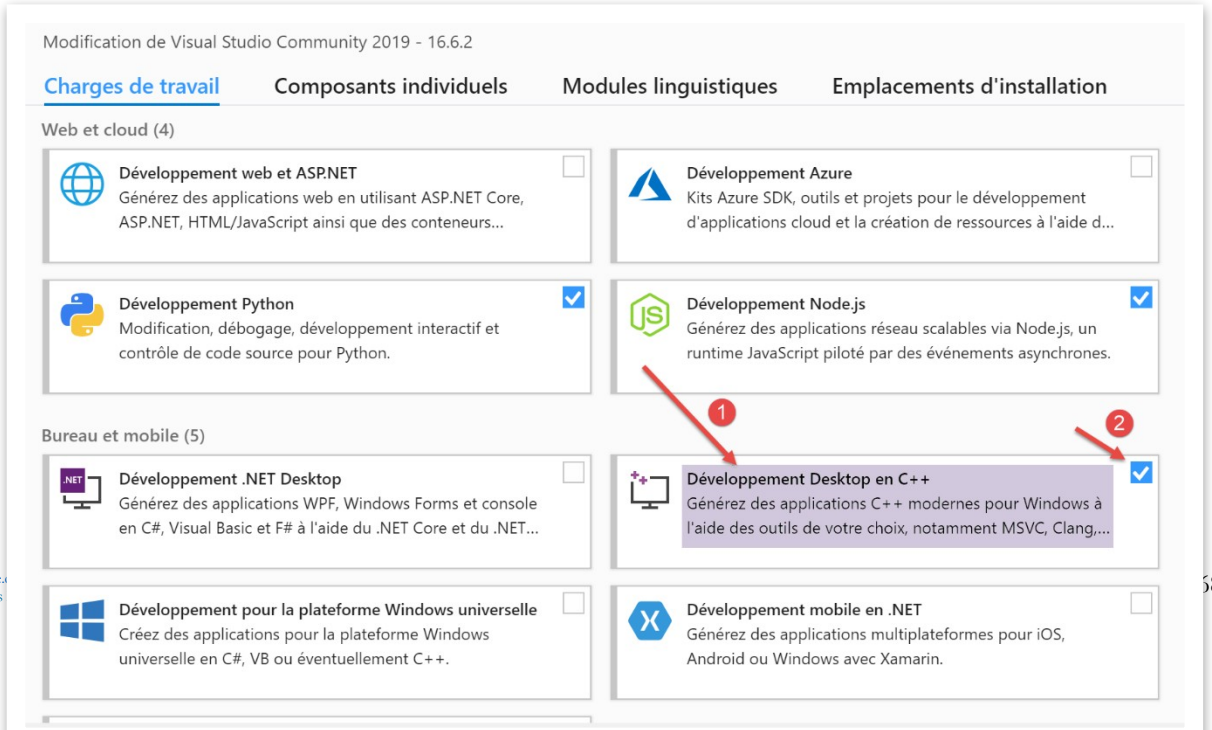
J'ai utilisé les informations de ce lien sauf pour la configuration du serveur Apache. Pour celle-ci j'ai utilisé un exemple de configuration du serveur Apache de Laragon.

37.3.2 Installation du module Python mod_wsgi

L'application Python / Flask que nous avons développée utilisait le serveur WSGI (Web Server Gateway Interface) **[werkzeug]** délivré avec Flask. Ce serveur est décrit |ici|. Le lien [https://www.fullstackpython.com/wsgi-servers.html] décrit le fonctionnement des serveurs WSGI. Il existe différents |serveurs WSGI|. L'un de ceux-ci est le serveur Apache fonctionnant en mode WSGI. C'est la solution adoptée ici parce que Laragon, que nous avons installé, vient avec un serveur Apache.

Pour que le serveur Apache puisse héberger une application Python, il nous faut installer le module Python **[mod_wsgi]**. L'installation de ce module est délicate parce qu'au cours de cette installation une compilation C++ a lieu. Pour réussir l'installation, il faut un compilateur Microsoft C++. Une solution simple est d'installer la version courante de Visual Studio Community [https://visualstudio.microsoft.com/fr/vs/community/].

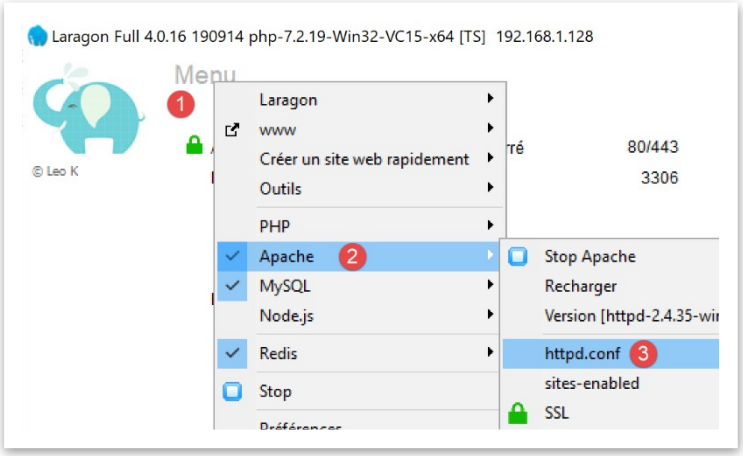
Si on n'a pas l'utilité de Visual Studio autrement que pour **[mod_wsgi]**, on peut limiter l'installation à l'environnement C++ :



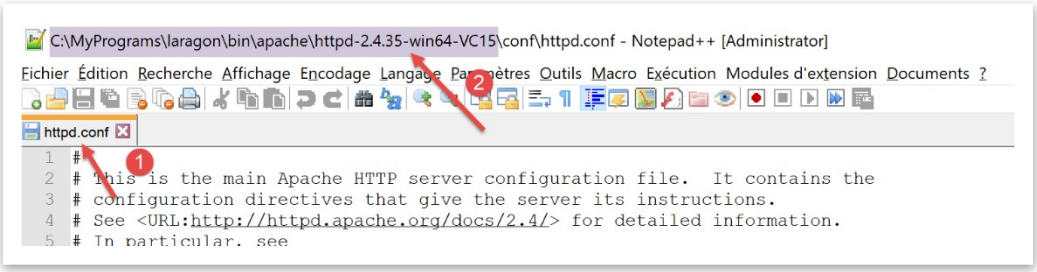
Une fois le compilateur C++ installé, l'installation du module **[mod_wsgi]** se fait dans un terminal PyCharm :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\impots\http-servers\11>SET
    MOD_WSGI_APACHE_ROOTDIR=C:\MyPrograms\laragon\bin\apache\httpd-2.4.35-win64-VC15
2.
3.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\impots\http-servers\11>pip install mod_wsgi
4.  Collecting mod_wsgi
5.    Using cached mod_wsgi-4.7.1.tar.gz (498 kB)
6.  Using legacy setup.py install for mod-wsgi, since package 'wheel' is not installed.
7.  Installing collected packages: mod-wsgi
8.      Running setup.py install for mod-wsgi ... done
9.  Successfully installed mod-wsgi-4.7.1
```

- ligne 1 : on fixe la valeur de la variable d'environnement **[MOD_WSGI_APACHE_ROOTDIR]**. Cette valeur est l'emplacement du serveur Apache dans le système de fichiers. Ici cet emplacement est **[<laragon>\bin\apache\httpd-2.4.35-win64-VC15]** où <laragon> est le dossier d'installation de Laragon. Vous pouvez obtenir cet emplacement de diverses façons. En voici une obtenue à partir d'une des options de Laragon :



En **[1-3]**, le fichier **[httpd.conf]** est le fichier de configuration principal du serveur Apache. Le fichier en question est alors ouvert dans un éditeur de texte (Notepad ++ ci-dessous) :



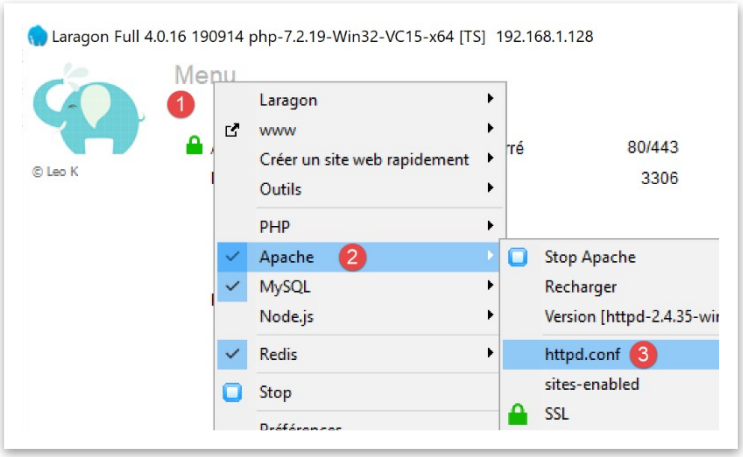
En **[2]**, le dossier d'installation d'Apache est ce qui précède la chaîne **[conf\httpd.conf]**.

Revenons à l'installation du module **[mod_wsgi]** :

- lignes 3-9 : installation du module **[mod_wsgi]** ;

37.3.3 Configuration du serveur Apache de Laragon

Nous allons configurer le serveur Apache de Laragon. Nous commençons par son fichier de configuration principal **[httpd.conf]** :



Nous nous plaçons à la fin du fichier **[httpd.conf]** :

```
1.  …
2.  IncludeOptional "C:/MyPrograms/laragon/etc/apache2/alias/*.conf"
3.  IncludeOptional "C:/MyPrograms/laragon/etc/apache2/sites-enabled/*.conf"
4.  Include "C:/MyPrograms/laragon/etc/apache2/httpd-ssl.conf"
5.  Include "C:/MyPrograms/laragon/etc/apache2/mod_php.conf"
6.
7.  # python mod_wsgi
8.  LoadModule wsgi_module "c:/data/st-2020/dev/python/cours-2020/python3-flask-2020/venv/lib/site-packages/
    mod_wsgi/server/mod_wsgi.cp38-win_amd64.pyd"
```

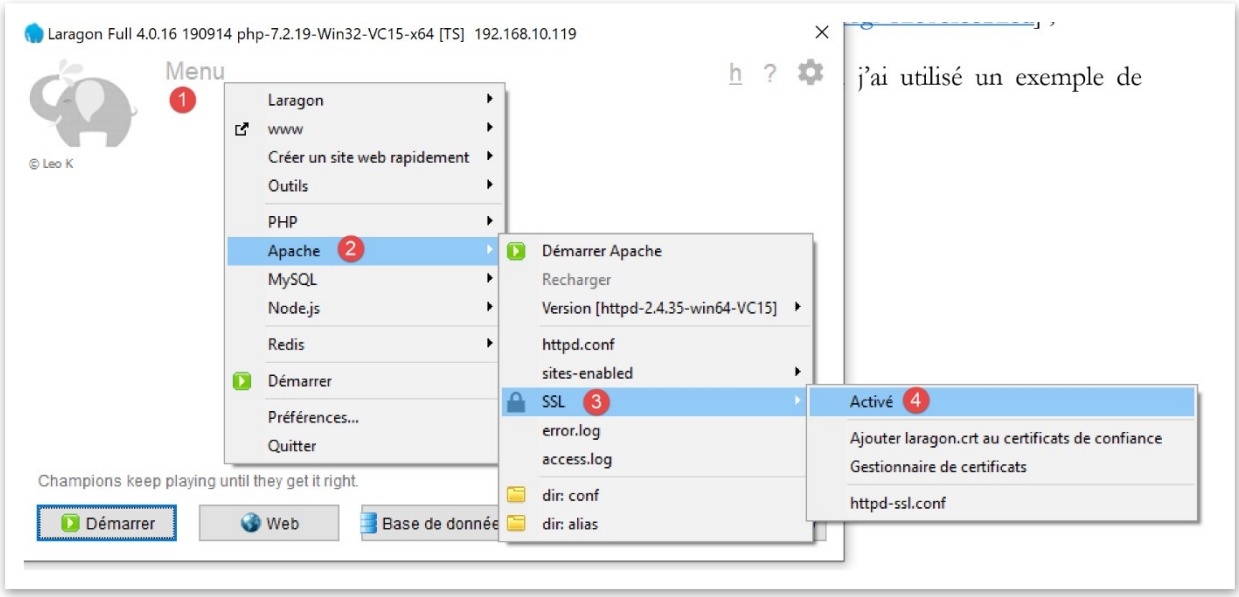
- la ligne 8 a été ajoutée à l'existant du fichier **[httpd.conf]** en fin de fichier. Elle indique au serveur Apache où se trouve un élément du module **[mod_wsgi]** que nous venons d'installer ;

Une façon simple d'avoir le chemin de la ligne 8 est d'exécuter la commande suivante dans un terminal PyCharm :

```
1.  (venv) C:\Data\st-2020\dev\python\cours-2020\python3-flask-2020\impots\http-servers\11>mod_wsgi-express
    module-config
2.  LoadModule wsgi_module "c:/data/st-2020/dev/python/cours-2020/python3-flask-2020/venv/lib/site-packages/
    mod_wsgi/server/mod_wsgi.cp38-win_amd64.pyd"
3.  WSGIPythonHome "c:/data/st-2020/dev/python/cours-2020/python3-flask-2020/venv"
```

Certaines documentations disent qu'il faut ajouter les lignes 2 et 3 à la fin du fichier **[httpd.conf]**. Dans mon cas, la ligne 3 ci-dessus provoquait une erreur (module **[encodings]** manquant). Aussi n'a-t-elle pas été mise dans le fichier **[httpd.conf]**. Seule la ligne 2 y a été mise. La signification des différents paramètres du module **[mod_wsgi]** utilisables dans les fichiers de configuration d'Apache sont décrits |ici|.

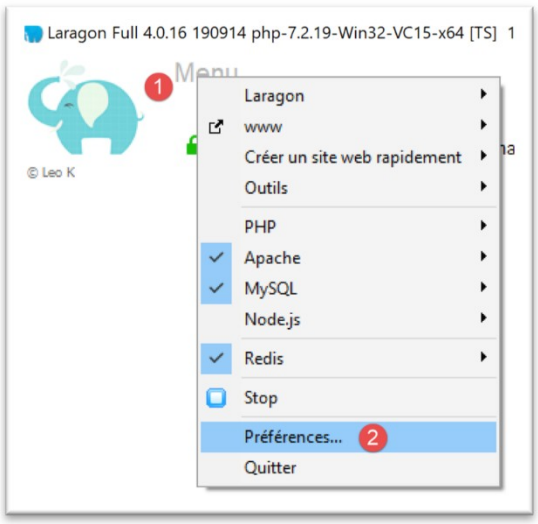
Ensuite, nous allons activer le protocole HTTPS du serveur Apache :

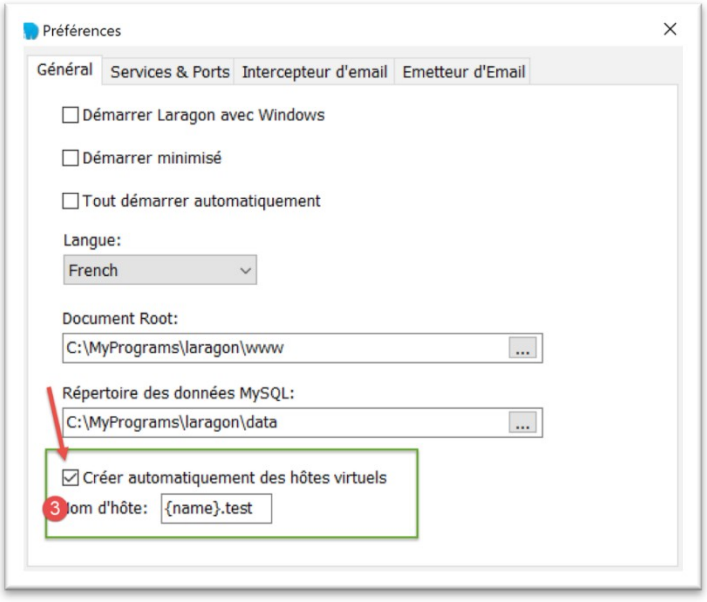


- en **[1-4]**, on active le protocole HTTPS du serveur Apache ;

Désormais, nous pourrons utiliser des URL **[https://serveur/chemin]** ;

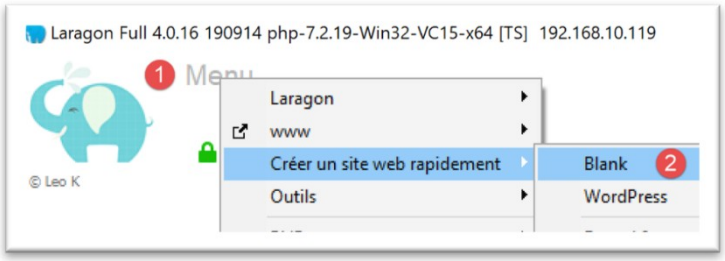
Pour configurer Apache de façon à servir une application Flask, le lien référencé |plus haut|utilise des serveurs virtuels. Laragon propose également de gérer des serveurs virtuels :

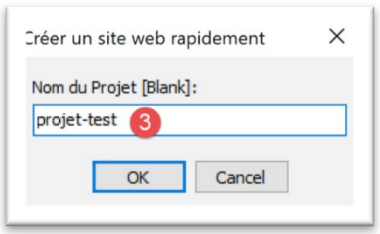


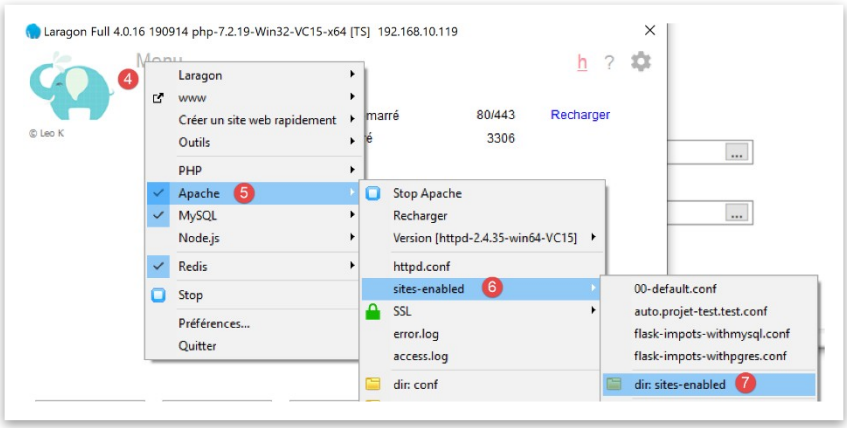


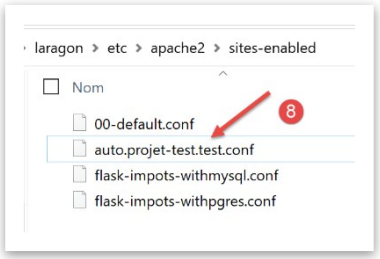
- en **[1-3]**, on demande à Laragon de créer automatiquement des hôtes virtuels ;

L'étape suivante est de créer un projet web avec Laragon :



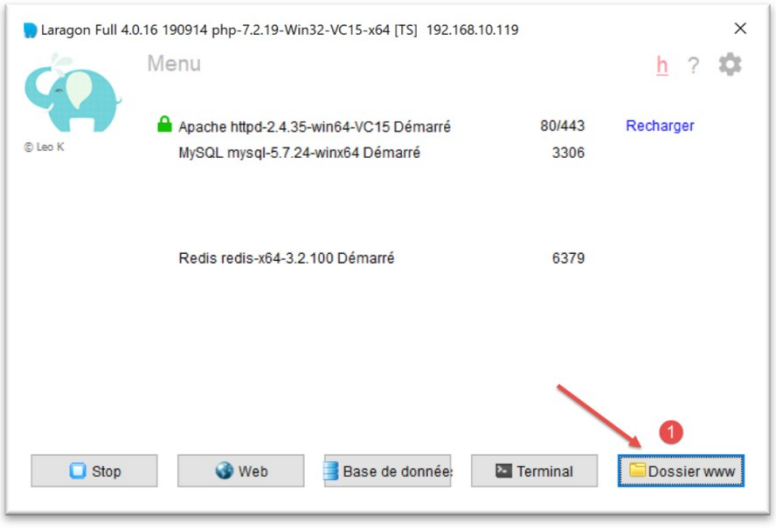


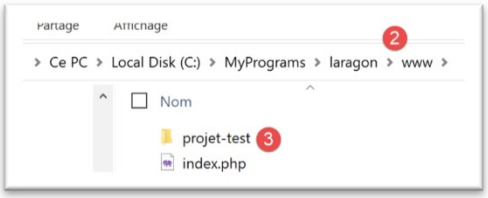


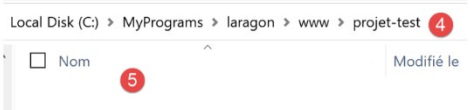


- en **[1-3]**, on crée un projet PHP vide ;
- en **[4-8]**, Laragon a créé un site virtuel appelé **[auto.projet-test.test]** configuré par le fichier **[auto.projet-test.test.conf] [8]** du dossier **[sites-enabled] [7]**. Ce dossier se trouve à l'adresse **[<laragon>\etc\apache2\sites-enabled]** où **[laragon]** est le dossier d'installation de Laragon ;

Bien que ça n'entre pas dans ce qui est fait en ce moment, vous pouvez avoir la curiosité d'aller voir ce qu'est ce site web **[projet-test]** que nous venons de créer :







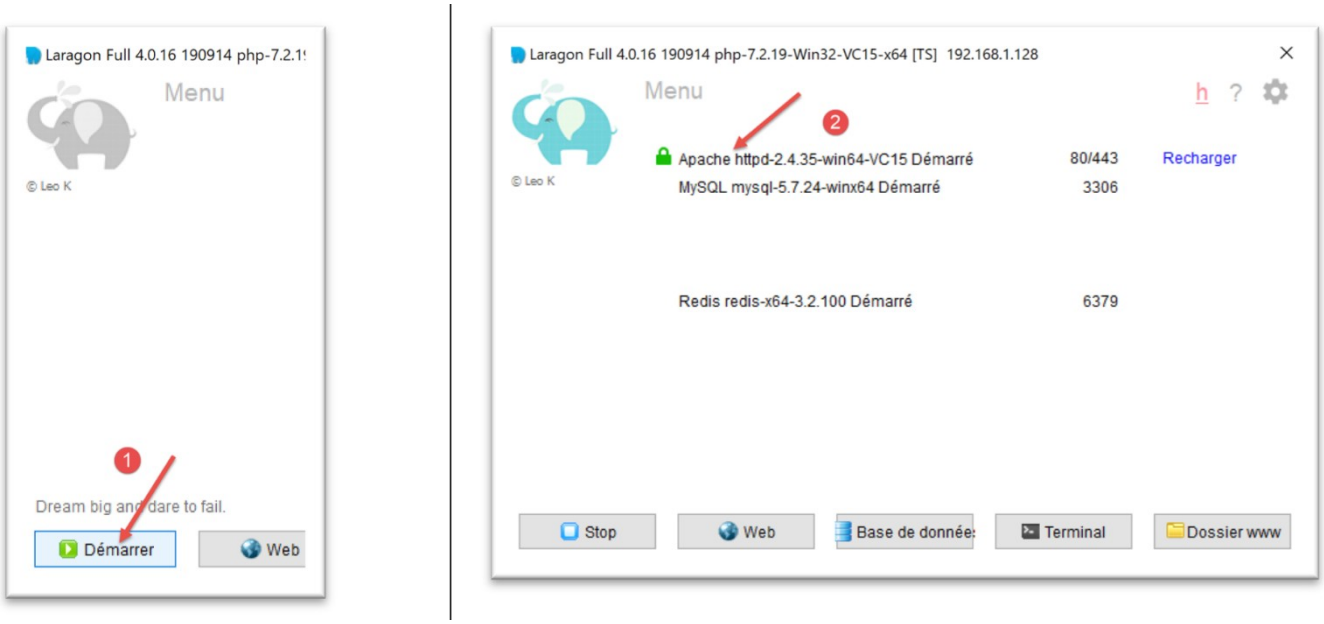
- en **[1-5]**, un projet vide a été créé. C'est un projet PHP situé dans le dossier **[<laragon>/www]** où **[laragon]** est le dossier d'installation de Laragon ;

Maintenant examinons le fichier **[auto.projet-test.test.conf]** généré par Laragon dans le dossier **[<laragon>\etc\apache2\sites-enabled]** :

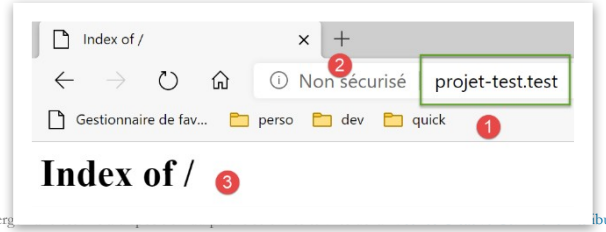
```
1.  define ROOT "C:/MyPrograms/laragon/www/projet-test/"
2.  define SITE "projet-test.test"
3.
4.  <VirtualHost *:80>
5.      DocumentRoot "${ROOT}"
6.      ServerName ${SITE}
7.      ServerAlias *.${SITE}
8.      <Directory "${ROOT}">
9.          AllowOverride All
10.         Require all granted
11.     </Directory>
12. </VirtualHost>
13.
14. <VirtualHost *:443>
15.     DocumentRoot "${ROOT}"
16.     ServerName ${SITE}
17.     ServerAlias *.${SITE}
18.     <Directory "${ROOT}">
19.         AllowOverride All
20.         Require all granted
21.     </Directory>
22.
23.     SSLEngine on
24.     SSLCertificateFile      C:/MyPrograms/laragon/etc/ssl/laragon.crt
25.     SSLCertificateKeyFile   C:/MyPrograms/laragon/etc/ssl/laragon.key
26.
27. </VirtualHost>
```

- ligne 1 : la racine, dans le système de fichiers, du projet créé ;
- ligne 2 : le nom du serveur virtuel. Les URL pour ce serveur seront de la forme **[http(s)://projet-test.test/chemin]** ;
- lignes 4-12 : configuration du site virtuel pour le port 80 (ligne 4) et le protocole HTTP ;
- lignes 14-27 : configuration du site virtuel pour le port 443 (ligne 14) et le protocole HTTPS ;

Voyons comment fonctionne un serveur virtuel. Lançons tout d'abord le serveur Apache et PHP :

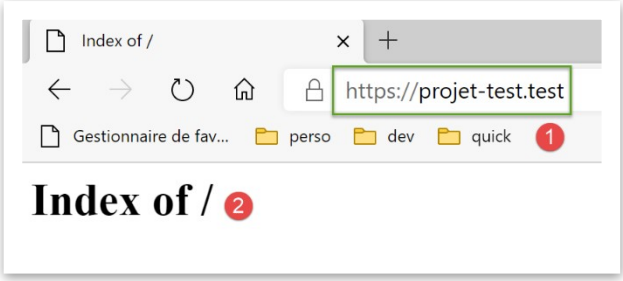


Avec un navigateur nous demandons ensuite l'URL [http://projet-test.test/] :



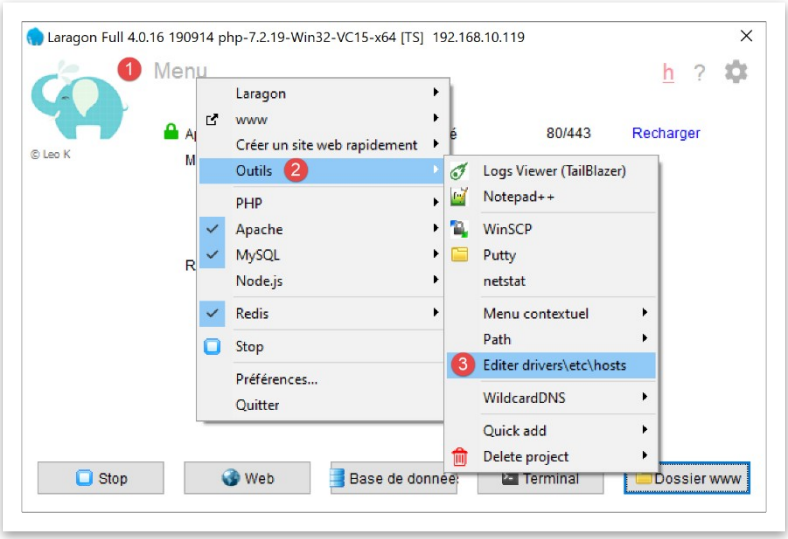
- en **[1]**, l'URL demandée ;
- en **[2]**, le protocole HTTP a été utilisé ;
- en **[3]**, parce que le projet **[projet-test]** est vide, on obtient l'index de son dossier (liste de son contenu), index vide ;

Maintenant demandons l'URL sécurisée [https://projet-test.test/] :



- en **[1-2]**, on obtient la même réponse que précédemment mais avec le protocole HTTPS **[1]** ;

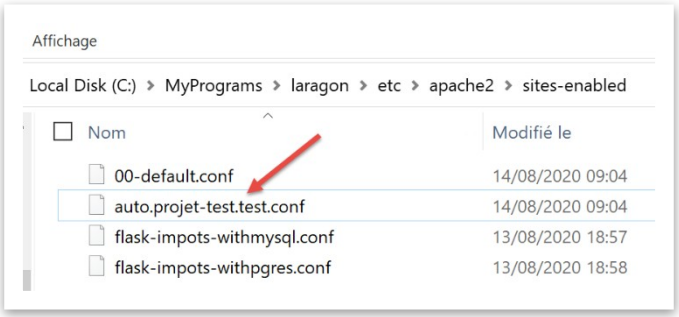
La création du serveur virtuel **[projet-test.test]** a créé une nouvelle entrée dans le fichier **[<windows>/system32/drivers/etc/hosts]** :



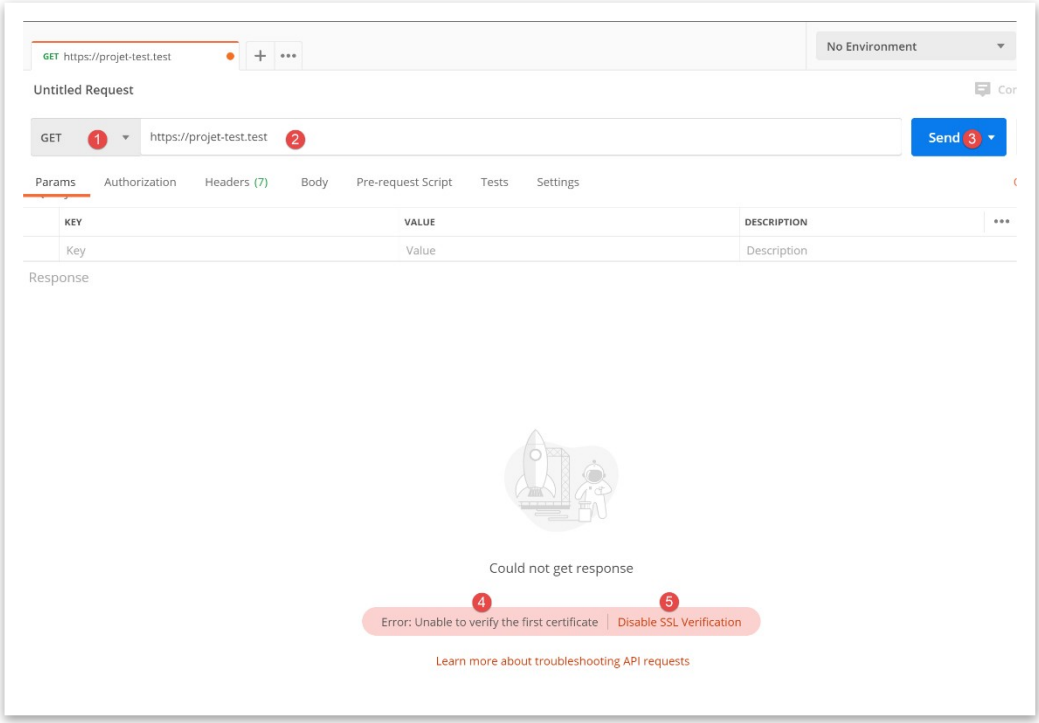

```
1.  # Copyright (c) 1993-2009 Microsoft Corp.
2.  #
3.  # This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
4.  #
5.  # This file contains the mappings of IP addresses to host names. Each
6.  # entry should be kept on an individual line. The IP address should
7.  # be placed in the first column followed by the corresponding host name.
8.  # The IP address and the host name should be separated by at least one
9.  # space.
10. #
11. # Additionally, comments (such as these) may be inserted on individual
12. # lines or following the machine name denoted by a '#' symbol.
13. #
14. # For example:
15. #
16. #      102.54.94.97     rhino.acme.com          # source server
17. #       38.25.63.10     x.acme.com              # x client host
18.
19. # localhost name resolution is handled within DNS itself.
20. #       127.0.0.1       localhost
21. #       ::1             localhost
22.
23. 127.0.0.1       projet-test.test     #laragon magic!
```

- ligne 23 : l'adresse IP du nom **[projet-test.test]** est 127.0.0.1, ç-à-d l'adresse de **[localhost]** (ligne 20), la machine locale. Ainsi lorsque dans un navigateur on tape l'URL **[http://projet-test.test/chemin]**, la requête est envoyée à l'adresse 127.0.0.1 sur le port 80. C'est alors le serveur Apache de la machine locale (localhost) qui répond.

On peut se demander pourquoi lorsqu'on tape la requête [http://projet-test.test/], le serveur Apache utilise la configuration du fichier **[<laragon>\etc\apache2\sites-enabled\auto.projet-test.test.conf]** :

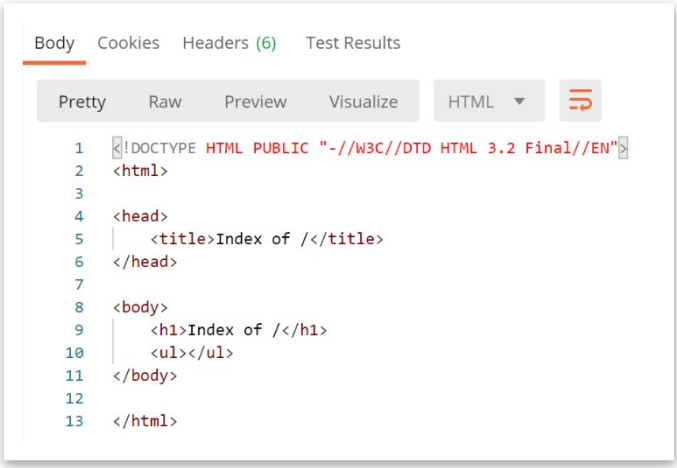


Pour le comprendre, il faut voir ce qu'envoie le navigateur au serveur Apache lorsqu'on fait cette requête. Faisons-la avec Postman :



- en **[1-3]**, on envoie une requête HTTPS **[1]** ;
- en **[4]**, Postman indique qu'il n'a pas reconnu le certificat de sécurité. Le protocole HTTPS établit une liaison cryptée entre le client web (ici Postman) et le serveur Apache. Cette liaison cryptée se fait au moyen de certificats échangés entre le client et le serveur. C'est le serveur qui initie le dialogue d'établissement de la liaison cryptée en envoyant au client un certificat de sécurité. Pour que ce certificat soit accepté par le client il faut qu'il soit signé, en clair acheté à des entreprises habilitées à délivrer des certificats de sécurité. Lorsque nous avons activé le procole HTTPS de Laragon, Laragon a créé lui-même le certificat de sécurité. On dit alors du certificat qu'il est auto-signé. La plupart des clients web émettent un avertissement lorsqu'ils reçoivent un certificat auto-signé. C'est ce que fait Postman en **[4]**. La plupart des clients web proposent alors de désactiver la vérification du certificat de sécurité envoyé par le serveur. C'est ce que propose Postman en **[5]** ;

Nous cliquons sur le lien **[5]** pour désactiver la vérification SSL (Secure Sockets Layer). SSL / TSL (Transport Layer Security) est un protocole de sécurité qui crée un canal de communication sécurisé entre deux machines de l'internet. C'est le protocole utilisé ici par Apache. La réponse est la suivante :



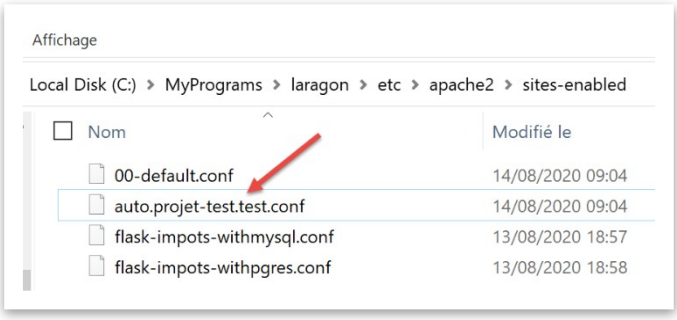
Nous recevons la même page qu'avec un navigateur traditionnel. Maintenant voyons le dialogue client / serveur dans la console Postman (Ctrl-Alt-C) :

```
1.  GET / HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.2
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: d153f711-ad99-4e1d-93c3-61c25374d1be
6.  Host: projet-test.test
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
9.
10. HTTP/1.1 200 OK
11. Date: Fri, 14 Aug 2020 09:19:24 GMT
12. Server: Apache/2.4.35 (Win64) OpenSSL/1.1.1b PHP/7.2.19 mod_wsgi/4.7.1 Python/3.8
13. Content-Length: 161
14. Keep-Alive: timeout=5, max=100
15. Connection: Keep-Alive
16. Content-Type: text/html;charset=UTF-8
```

- ligne 6 : l'entête HTTP **[Host]** précise le nom du serveur ciblé par le client web. C'est le principe des serveurs virtuels. A une même adresse IP (ici 127.0.0.1), un serveur web peut héberger plusieurs sites web avec des noms différents. L'entête HTTP **[Host]** permet au client de dire à quel serveur (ici de l'adresse 127.0.0.1) il s'adresse ;

Maintenant que fait Apache ?

Lorsqu'il démarre, Apache lit tous les fichiers de configuration trouvés dans le dossier **[[<laragon>\etc\apache2\sites-enabled]** :



Chaque fichier de configuration définit un serveur virtuel. Par exemple dans le fichier **[auto.projet-test.test.conf]**, on trouve la ligne suivante :

```
1.  define ROOT "C:/MyPrograms/laragon/www/projet-test/"
2.  define SITE "projet-test.test"
3.  …
```

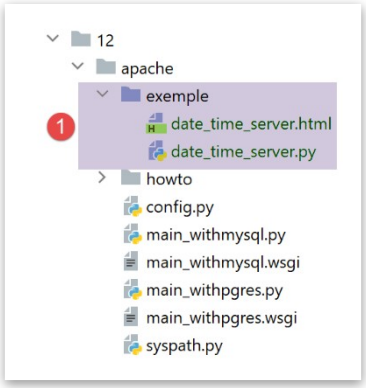
La ligne 2 définit le serveur virtuel **[projet-test.test]**. Le fichier **[auto.projet-test.test.conf]** est la configuration de ce serveur virtuel. Parce qu'il lit au démarrage tous les fichiers de configuration du dossier **[<laragon>\etc\apache2\sites-enabled]**, le serveur Apache sait qu'il existe un serveur virtuel appelé **[projet-test.test]**. Ainsi lorsqu'il reçoit du client Postman, la requête HTTPS :

```
1.  GET / HTTP/1.1
2.  User-Agent: PostmanRuntime/7.26.2
3.  Accept: */*
4.  Cache-Control: no-cache
5.  Postman-Token: d153f711-ad99-4e1d-93c3-61c25374d1be
6.  Host: projet-test.test
7.  Accept-Encoding: gzip, deflate, br
8.  Connection: keep-alive
```

il reconnaît que la requête est adressée au serveur virtuel **[projet-test.test]** (ligne 6) et que celui-ci existe. Il utilise alors la configuration du serveur virtuel **[projet-test.test]** pour répondre au client Postman.

37.4 Création d'un premier serveur virtuel Apache

Maintenant que nous savons à quoi servent des serveurs virtuels et comment les définir, nous allons en créer un. Il servira à exécuter une application Python Flask installée dans un dossier **[Apache]** de la version 17 en cours de déploiement sur le serveur Apache:



Nous avons mis dans le dossier **[http-servers/12/apache/exemple]** l'application développée au paragraphe |lien|, un service web de date /heure :

Le serveur **[date_time_server.py]** est le suivant :

```
1.  # imports
2.  import os
3.  import sys
4.  import time
5.
6.  # il nous faut mettre dans le Python Path le dossier des modules
7.  # pour portage vers apache windows
8.  sys.path.insert(0, "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/venv/lib/site-packages")
9.
10. from flask import Flask, make_response, render_template
11.
12. # application Flask
13. script_dir = os.path.dirname(os.path.abspath(__file__))
14. application = Flask(__name__, template_folder=f"{script_dir}")
15.
16. # Home URL
17. @application.route('/')
18. def index():
19.     # envoi heure au client
20.     # time.localtime : nb de millisecondes depuis 01/01/1970
21.     # time.strftime permet de formater l'heure et la date
22.     # format affichage date-heure
23.     # d: jour sur 2 chiffres
24.     # m: mois sur 2 chiffres
25.     # y : année sur 2 chiffres
26.     # H : heure 0,23
27.     # M : minutes
28.     # S: secondes
29.
30.     # date / heure du moment
31.     time_of_day = time.strftime('%d/%m/%y %H:%M:%S', time.localtime())
32.     # on génère le document à envoyer au client
33.     page = {"date_heure": time_of_day}
34.     document = render_template("date_time_server.html", page=page)
35.     print("document", type(document), document)
```

```
36.     # réponse HTTP au client
37.     response = make_response(document)
38.     print("response", type(response), response)
39.     return response
40.
41. # main seulement
42. if __name__ == '__main__':
43.     application.config.update(ENV="development", DEBUG=True)
44.     application.run()
```

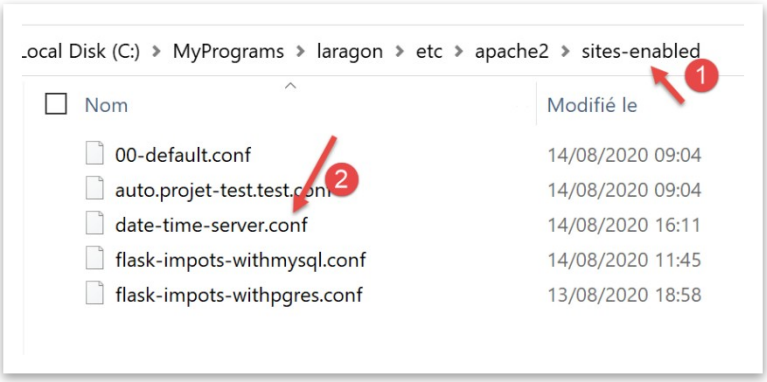
L'application Flask est référencée par l'identifiant **[application]** (lignes 14, 43, 44). Ce nom est **obligatoire**. Si on référence l'application Flask avec un autre identifiant l'application ne va pas marcher avec un message d'erreur indiquant qu'elle ne trouve pas l'URL demandée. Ce message d'erreur ne donne aucune indication sur la source de l'erreur. Il faut donc être vigilant sur ce point.

Le fichier HTML référencé ligne 34 est le suivant :

```
1.  <!DOCTYPE html>
2.  <html lang="fr">
3.  <head>
4.      <meta charset="UTF-8">
5.      <title>Date et heure du moment</title>
6.  </head>
7.  <body>
8.      <b>Date et heure du moment : {{page.date_heure}}</b>
9.  </body>
10. </html>
```

- **[date-time-server]** sera le serveur virtuel qui hébergera cette application. Il sera configuré par le fichier **[<laragon>\etc\apache2\sites-enabled\date-time-server.conf]** (on rappelle que ce nom est libre – Apache lit tous les fichiers présents dans **[sites-enabled]** pour découvrir les sites virtuels hébergés) ;

Nous obtenons ce fichier tout d'abord par recopie du fichier **[auto.projet-test.test.conf]** puis nous le modifions.



Le fichier **[date-time-server.conf]** sera le suivant :

```
1.  # dossier du script python-flask de l'application
2.  define ROOT "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/http-servers/12/apache/exemple"
3.
4.  # nom du site web configuré par ce fichier
5.  # ici il s'appellera date-time-server
6.  # les URL seront du type http(s)://date-time-server/chemin
7.  define SITE "date-time-server"
8.
9.  # mettre l'adresse IP 127.0.0.1 pour site SITE dans c:/windows/system32/drivers/etc/hosts
10.
11. # URL HTTP
12. <VirtualHost *:80>
13.     # avec l'alias / les URL auront la forme http(s)://date-time-server/chemin/...
14.     WSGIScriptAlias / "${ROOT}/date_time_server.py"
15.     DocumentRoot "${ROOT}"
16.     ServerName ${SITE}
17.     ServerAlias *.${SITE}
18.     <Directory "${ROOT}">
19.         AllowOverride All
20.         Require all granted
21.     </Directory>
22. </VirtualHost>
23.
24. # URL sécurisées avec HTTPS
25. <VirtualHost *:443>
26.     # avec l'alias / les URL auront la forme http(s)://date-time-server/chemin/...
```

```
27.     WSGIScriptAlias / "${ROOT}/date_time_server.py"
28.     DocumentRoot "${ROOT}"
29.     ServerName ${SITE}
30.     ServerAlias *.${SITE}
31.     <Directory "${ROOT}">
32.         AllowOverride All
33.         Require all granted
34.     </Directory>
35.
36.     SSLEngine on
37.     SSLCertificateFile      C:/MyPrograms/laragon/etc/ssl/laragon.crt
38.     SSLCertificateKeyFile   C:/myprograms/laragon/etc/ssl/laragon.key
39.
40. </VirtualHost>
```

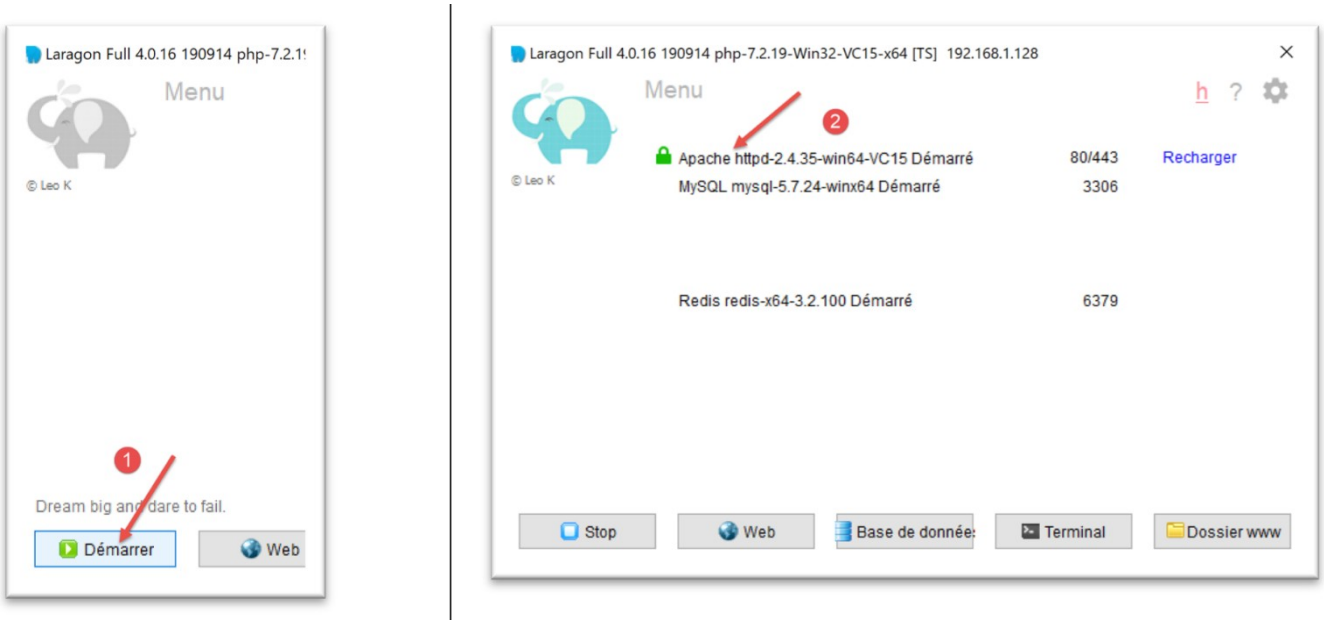
- ligne 7 : on donne un nom au serveur virtuel configuré par le fichier ;
- ligne 2 : on donne la valeur de la variable **[ROOT]** utilisée aux lignes 14 et 27 ;
- lignes 14 et 27 : on indique le chemin du script Python qui doit être exécuté lorsque le serveur virtuel reçoit une requête. On indique ici que les requêtes pour le serveur **[date-time-server]** sont traitées par le script Python **[date_time_server.py]**. Cette différence avec le fichier **[auto.projet-test.test.conf]** vient du fait que ce fichier configurait un serveur PHP alors que le fichier **[date-time-server.conf]** configure un serveur Python ;
- lignes 14 et 27 : l'attribut **[WSGIScriptAlias /]** indique ici que la racine du serveur **[date-time-server]** sera **[/]**. Ainsi les URL de l'application auront la forme **[http(s)://date-time-server/chemin]** ;
- lignes 14 et 27, on peut donner une autre racine à l'application par exemple **[WSGIScriptAlias /show]**. Alors les URL de l'application auront la forme **[http(s)://show/date-time-server/chemin]** ;

Il nous faut également ajouter une ligne au fichier **[<windows>/system32/drivers/etc/hosts]** :

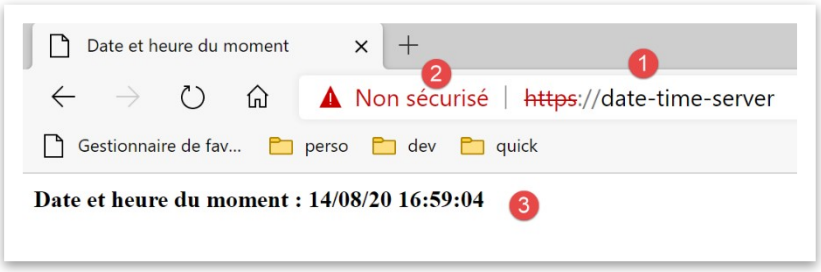
```
1.  # Copyright (c) 1993-2009 Microsoft Corp.
2.  #
3.  # This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
4.  #
5.  # This file contains the mappings of IP addresses to host names. Each
6.  # entry should be kept on an individual line. The IP address should
7.  # be placed in the first column followed by the corresponding host name.
8.  # The IP address and the host name should be separated by at least one
9.  # space.
10. #
11. # Additionally, comments (such as these) may be inserted on individual
12. # lines or following the machine name denoted by a '#' symbol.
13. #
14. # For example:
15. #
16. #      102.54.94.97     rhino.acme.com          # source server
17. #       38.25.63.10     x.acme.com              # x client host
18.
19. # localhost name resolution is handled within DNS itself.
20. #       127.0.0.1       localhost
21. #       ::1             localhost
22.
23. 127.0.0.1       flask-impots-withmysql          # cours python - appli impôts avec MySQL
24. 127.0.0.1       flask-impots-withpgres          # cours python - appli impôts avec PostgreSQL
25. 127.0.0.1       date-time-server                # cours python - heure et date du moment
26. 127.0.0.1       projet-test.test                #laragon magic!
```

On ajoute la ligne 25, pour donner l'adresse IP **[127.0.0.1]** au serveur virtuel **[date-time-server]**.

Vérifions tout cela. Nous lançons le serveur Apache :



Nous demandons ensuite l'URL **[https://date-time-server]** avec un navigateur :

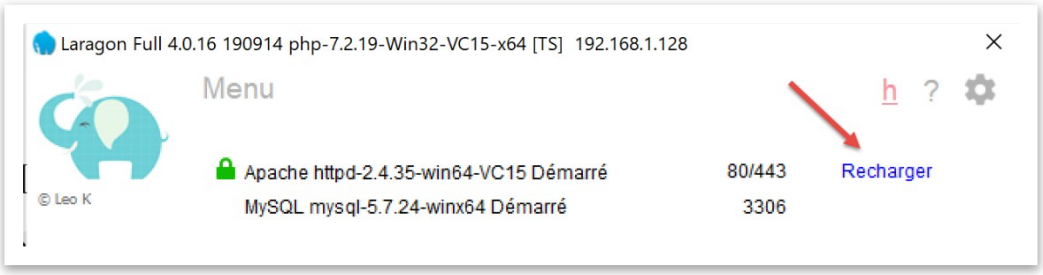


- en **[1]**, l'URL demandée ;
- en **[3]**, la réponse du serveur ;
- en **[2]**, le navigateur indique que la connection HTTPS n'est pas sûre car il a détecté que le certificat envoyé par le serveur Apache était auto-signé ;

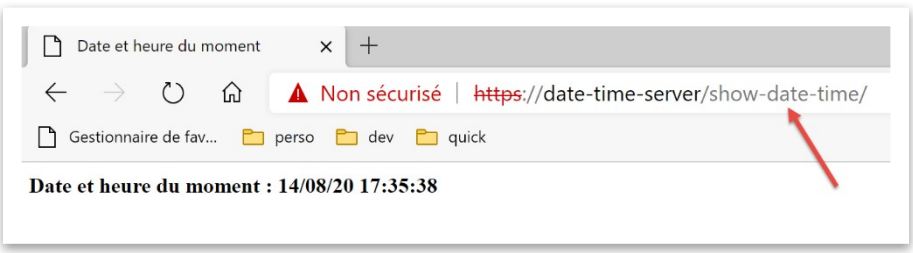
Maintenant dans le fichier **[date-time-server.conf]**, mettons un alias aux lignes 14 et 27 :

```
    WSGIScriptAlias /show-date-time "${ROOT}/date_time_server.py"
```

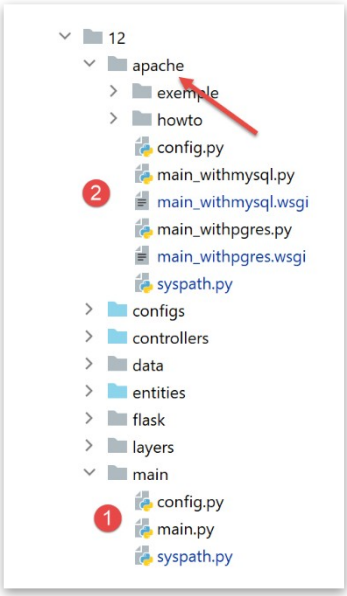
La modification n'est pas prise en compte immédiatement par le serveur Apache. Il faut le recharger :



On demande ensuite l'URL **[https://date-time-server/show-date-time]**. La réponse du serveur est alors la suivante :



37.5 Portage de l'application de calcul de l'impôt sur Apache / Windows



Le dossier **[apache] [2]** est obtenu initialement par recopie du dossier **[main]**. Il est important qu'ils soient au même niveau pour que les chemins du script **[syspath.py]** copié de **[1]** vers **[2]** restent valides. Pour ne pas polluer l'application **[impots / http-servers/ 12]** qui fonctionne, nous mettons dans **[apache]** la configuration qui sera exécutée par le serveur Apache ;

- le fichier **[config]** de **[2]** est le même que **[config]** de **[1]** ;
- le fichier **[syspath]** de **[2]** est le même que **[syspath]** de **[1]** ;
- le fichier **[main_withmysql]** de **[2]** est le fichier **[main]** de **[1]** avec les modifications suivantes :

  Le script principal **[main]** recevait un paramètre **[mysql / pgres]** qui lui disait quel SGBD utilisé. Le script **[main_withmysql]** utilise le SGBD MySQL :

```
1.  # on attend un paramètre mysql ou pgres
2.  import os
3.  import sys
4.
5.  # on configure l'application avec MySQL
6.  import config
7.  config = config.configure({'sgbd': "mysql"})
8.
9.  # dépendances
10. from SendAdminMail import SendAdminMail
11. from Logger import Logger
12. from ImpôtsError import ImpôtsError
13. …
```

  Ligne 7, on fixe le SGBD à MySQL.

- le fichier **[main_withpgres]** de **[2]** est le fichier **[main]** de **[1]** avec les modifications suivantes : il utilise le SGBD PostgreSQL :

```
1.  # on attend un paramètre mysql ou pgres
2.  import os
3.  import sys
4.
5.  # on configure l'application avec MySQL
6.  import config
7.  config = config.configure({'sgbd': "pgres"})
8.
9.  # dépendances
10. from SendAdminMail import SendAdminMail
11. from Logger import Logger
12. from ImpôtsError import ImpôtsError
13. …
```

Ligne 7, on fixe le SGBD à PostgreSQL.

Ceci fait, on crée le script **[main_withmysql.wsgi]** (le suffixe utilisé n'a pas d'importance) suivant :

```
1.  # dossier de ce fichier
2.  import os
3.  script_dir = os.path.dirname(os.path.abspath(__file__))
4.
5.  # on l'ajoute au syspath pour que l'import qui suit soit possible
6.  import sys
7.  sys.path.insert(0, script_dir)
8.
9.  # on importe l'application Flask [app] en lui donnant le nom [application]
10. from main_withmysql import app as application
```

Le script **[main_withmysql.wsgi]** sera la cible exécutée par le serveur Apache en mode WSGI :

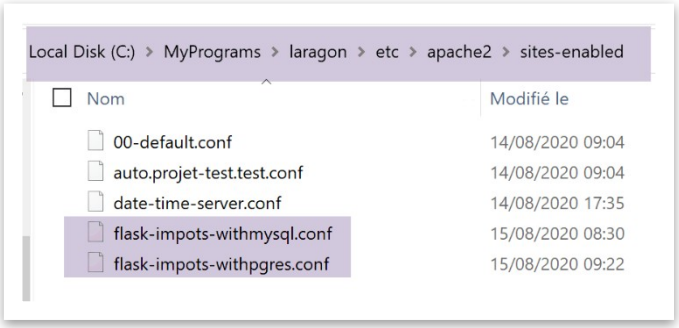
- la cible du serveur Apache aurait pu être le script **[main_withmysql.py]** comme il a été fait précédemment avec le script **[date_time_server.py]**. Mais il aurait fallu le modifier un peu :
  - contrairement au mode d'exécution avec un script console, avec Apache, le dossier contenant la cible **[main_withmysql.py]** ne fait pas partie du Python Path. Aussi, la ligne 6 du script **[main_withmysql.py]** provoque-t-elle une erreur ;
  - la seconde modification qu'il aurait fallu faire est que dans **[main_withmysql]** l'application Flask est référencée par l'identifiant **[app]**. On sait que pour Apache / WSGI elle doit être également référencée par un identifiant **[application]** ;
- plutôt que de modifier **[main_withmysql.py]**, on change la cible d'Apache. Ce sera désormais le script **[main_withmysql.wsgi]** ci-dessus :
  - lignes 1-7 : on met le dossier du script dans le Python Path. Du coup, la ligne 6 de **[main_withmysql.py]** ne provoque plus d'erreur ;
  - lignes 9-10 : l'importation de **[main_withmysql.py]** provoque son exécution. Par ailleurs on référence l'application Flask **[app]** trouvée dans **[main_withmysql.py]** avec l'identificateur **[application]** dont a besoin Apache en mode WSGI ;

On fait de même avec le script **[main_withpgres.wsgi]** :

```
1.  # dossier de ce fichier
2.  import os
3.  script_dir = os.path.dirname(os.path.abspath(__file__))
4.
5.  # on l'ajoute au syspath pour que l'import qui suit soit possible
6.  import sys
7.  sys.path.insert(0, script_dir)
8.
9.  # on importe l'application Flask [app] en lui donnant le nom [application]
10. from main_withpgres import app as application
```

Nous avons désormais les cibles exécutables pour le serveur Apache. Il nous faut maintenant créer deux serveurs virtuels, un pour chaque cible.

Dans **[<laragon>\etc\apache2\sites-enabled]**, nous créons le fichier **[flask-impots-withmysql.conf]** (le nom donné n'a pas d'importance) :




```
1.  # dossier du script .wsgi
2.  define ROOT "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/http-servers/12/apache"
```

```
3.
4.  # nom du site web configuré par ce fichier
5.  # ici il s'appellera flask-impots-withmysql
6.  # les URL seront du type http(s)://flask-impots-withmysql/chemin
7.  define SITE "flask-impots-withmysql"
8.
9.  # mettre l'adresse IP 127.0.0.1 pour site SITE dans c:/windows/system32/drivers/etc/hosts
10.
11. # mettre ici les chemins des bibliothèques Python à utiliser - les séparer par des virgules
12. # ici les bibliothèques d'un environnement virtuel Python
13. WSGIPythonPath  "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/venv/lib/site-packages"
14.
15. # Python Home - nécessaire uniquement s'il y a plusieurs versions de Python installées
16. # WSGIPythonHome "C:/Program Files/Python38"
17.
18. # URL HTTP
19. <VirtualHost *:80>
20.     # avec l'alias / les URL auront la forme /{prefixe_url}/action/...
21.     # avec l'alias /impots les URL auront la forme /impots/{prefixe_url}/action/...
22.     # où [prefixe_url] est défini dans parameters.py
23.     WSGIScriptAlias / "${ROOT}/main_withmysql.wsgi"
24.     DocumentRoot "${ROOT}"
25.     ServerName ${SITE}
26.     ServerAlias *.${SITE}
27.     <Directory "${ROOT}">
28.         AllowOverride All
29.         Require all granted
30.     </Directory>
31. </VirtualHost>
32.
33. # URL sécurisées avec HTTPS
34. <VirtualHost *:443>
35.     # avec l'alias / les URL auront la forme /{prefixe_url}/action/...
36.     # avec l'alias /impots les URL auront la forme /impots/{prefixe_url}/action/...
37.     # où [prefixe_url] est défini dans parameters.py
38.     WSGIScriptAlias / "${ROOT}/main_withmysql.wsgi"
39.     DocumentRoot "${ROOT}"
40.     ServerName ${SITE}
41.     ServerAlias *.${SITE}
42.     <Directory "${ROOT}">
43.         AllowOverride All
44.         Require all granted
45.     </Directory>
46.
47.     SSLEngine on
48.     SSLCertificateFile      C:/MyPrograms/laragon/etc/ssl/laragon.crt
49.     SSLCertificateKeyFile   C:/myprograms/laragon/etc/ssl/laragon.key
50.
51. </VirtualHost>
```

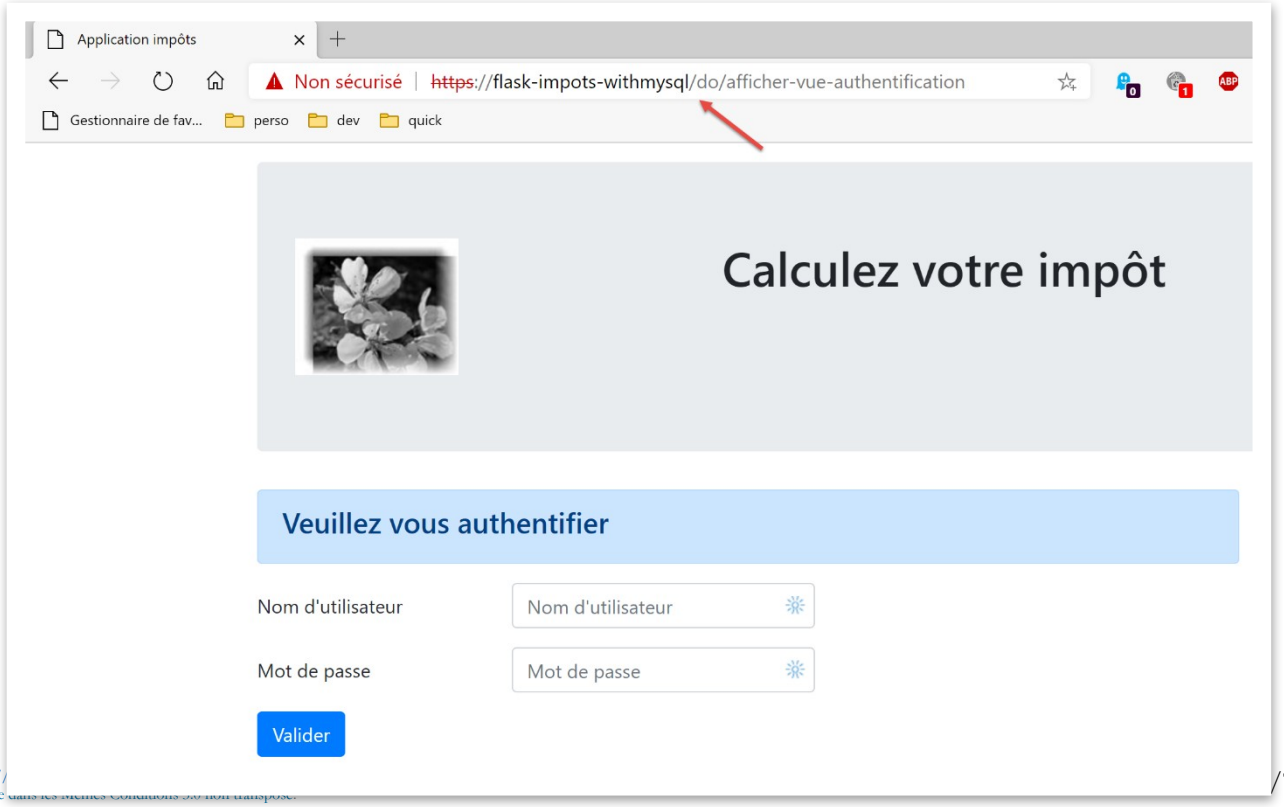
- ligne 2 : la racine de l'application, le dossier **[apache]** que nous avons créé ;
- lignes 23, 38 : la cible **[main_witmysql.wsgi]** que nous avons créée :
- ligne 7 : le serveur virtuel s'appellera **[flask-impots-withmysql]** ;
- ligne 13 : la directive **[WSGIPythonPath]** permet d'ajouter des dossiers au Python Path. Ici, Apache n'a pas connaissance que nous avons utilisé un environnement virtuel pour développer l'application et que tous les modules utilisés par l'application sont dans cet environnement virtuel. Aussi, ligne 13, nous ajoutons le dossier qui contient tous les modules de l'environnement virtuel utilisé. Une possibilité est de copier ce dossier à un autre endroit du système de fichiers et de référencer cet endroit. Une autre possibilité est d'ajouter ce dossier dans le Python Path directement dans la cible **[main_witmysql.wsgi]** (c'est probablement une meilleure solution) ;
- ligne 16 : on peut indiquer à Apache le dossier d'installation de Python dans le système de fichiers. Normalement, celui-ci est dans le PATH de la machine et souvent cette ligne est inutile (c'était le cas ici). Il peut cependant y avoir plusieurs installations de Python sur la machine et celle désirée n'est pas dans le PATH de la machine. Alors cette ligne résoud le problème ;

On crée de la même façon un fichier **[flask-impots-withpgres.conf]** :

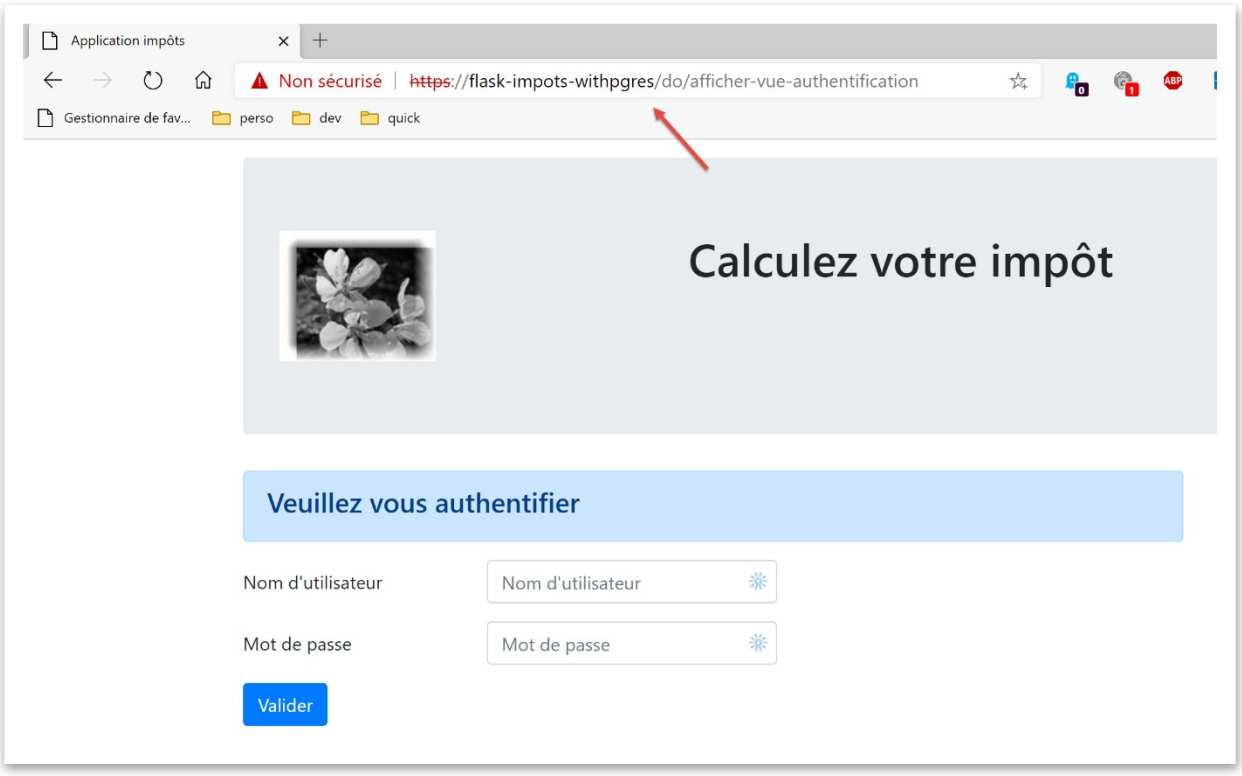
```
1.  # dossier du script .wsgi
2.  define ROOT "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/http-servers/12/apache"
3.
4.  # nom du site web configuré par ce fichier
5.  # ici il s'appellera flask-impots-withmysql
6.  # les URL seront du type http(s)://flask-impots-withmysql/chemin
7.  define SITE "flask-impots-withpgres"
8.
9.  # mettre l'adresse IP 127.0.0.1 pour site SITE dans c:/windows/system32/drivers/etc/hosts
```

```
10.
11. # mettre ici les chemins des bibliothèques Python à utiliser - les séparer par des virgules
12. # ici les bibliothèques d'un environnement virtuel Python
13. WSGIPythonPath  "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/venv/lib/site-packages"
14.
15. # Python Home - nécessaire uniquement s'il y a plusieurs versions de Python installées
16. # WSGIPythonHome "C:/Program Files/Python38"
17.
18. # URL HTTP
19. <VirtualHost *:80>
20.     # avec l'alias / les URL auront la forme /{prefixe_url}/action/...
21.     # avec l'alias /impots les URL auront la forme /impots/{prefixe_url}/action/...
22.     # où [prefixe_url] est défini dans parameters.py
23.     WSGIScriptAlias / "${ROOT}/main_withpgres.wsgi"
24.     DocumentRoot "${ROOT}"
25.     ServerName ${SITE}
26.     ServerAlias *.${SITE}
27.     <Directory "${ROOT}">
28.         AllowOverride All
29.         Require all granted
30.     </Directory>
31. </VirtualHost>
32.
33. # URL sécurisées avec HTTPS
34. <VirtualHost *:443>
35.     # avec l'alias / les URL auront la forme /{prefixe_url}/action/...
36.     # avec l'alias /impots les URL auront la forme /impots/{prefixe_url}/action/...
37.     # où [prefixe_url] est défini dans parameters.py
38.     WSGIScriptAlias / "${ROOT}/main_withpgres.wsgi"
39.     DocumentRoot "${ROOT}"
40.     ServerName ${SITE}
41.     ServerAlias *.${SITE}
42.     <Directory "${ROOT}">
43.         AllowOverride All
44.         Require all granted
45.     </Directory>
46.
47.     SSLEngine on
48.     SSLCertificateFile      C:/MyPrograms/laragon/etc/ssl/laragon.crt
49.     SSLCertificateKeyFile   C:/myprograms/laragon/etc/ssl/laragon.key
50.
51. </VirtualHost>
```

Nous sauvegardons tous ces fichiers, lançons le serveur Apache et les SGBD MySQL et PostgreSQL. L'application est configurée avec le préfixe d'URL **[/do]** et **[with_csrftoken=False]** (pas de jeton CSRF) dans **[configs/parameters.py]**. Nous demandons l'URL **[https://flask-impots-withmysql/do]**. La réponse du serveur est la suivante :



Nous demandons maintenant l'URL **[https://flask-impots-pgres/do]**. La réponse est la suivante :



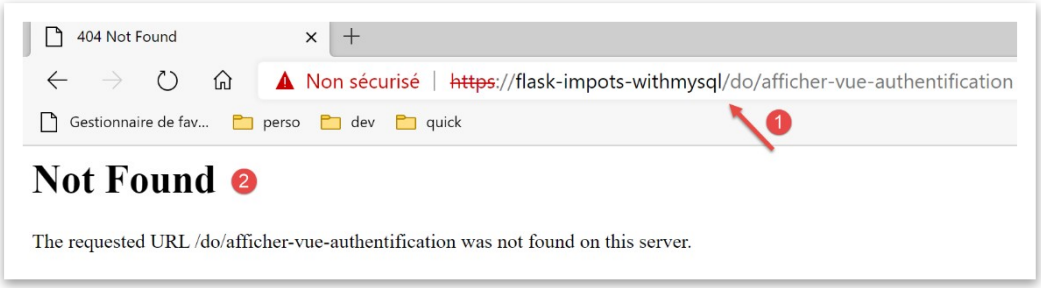
Les deux applications fonctionnent normalement.

Maintenant modifions le paramètre **[WSGIScriptAlias]** dans **[flask-impots-withmysql.conf]** :

```
1.  # dossier du script .wsgi
2.  define ROOT "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/http-servers/12/apache"
3.
4.  …
5.
6.  # URL HTTP
7.  <VirtualHost *:80>
8.      # avec l'alias / les URL auront la forme /{prefixe_url}/action/...
9.      # avec l'alias /impots les URL auront la forme /impots/{prefixe_url}/action/...
10.     # où [prefixe_url] est défini dans parameters.py
11.     WSGIScriptAlias /impots "${ROOT}/main_withmysql.wsgi"
12.     …
13. </VirtualHost>
14.
15. # URL sécurisées avec HTTPS
16. <VirtualHost *:443>
17.     # avec l'alias / les URL auront la forme /{prefixe_url}/action/...
18.     # avec l'alias /impots les URL auront la forme /impots/{prefixe_url}/action/...
19.     # où [prefixe_url] est défini dans parameters.py
20.     WSGIScriptAlias /impots "${ROOT}/main_withmysql.wsgi"
21.     …
22.
23. </VirtualHost>
```

- lignes 11 et 20, l'alias WSI est désormais **[/impots]** ;

Nous arrêtons / relançons le serveur Apache puis nous demandons l'URL **[https://flask-impots-withmysql/impots/do]**. La réponse du serveur est la suivante :



Il y a un plantage. L'URL **[1]** nous en donne la cause. Elle aurait du être **[https://flask-impots-withmysql/impots/do/afficher-vue-authentification]**. L'alias WSGI manque. C'est une erreur de notre application. Elle sait gérer un préfixe d'URL (/do est bien présent). On pourrait penser qu'en mettant le préfixe **[/impots/do]** à notre application cela résoudrait le problème précédent. Mais non. On rencontre alors d'autres types de problèmes. L'alias WSGI ne se comporte pas comme un préfixe d'URL.

Essayons de comprendre ce qui s'est passé. Nous avons demandé l'URL **[https://flask-impots-withmysql/impots/do]**. Nous nous attendions à avoir la vue d'authentification. En **[1]**, ci-dessus on voit que l'application a demandé son affichage mais pas avec la bonne URL. Examinons le cheminement de la requête **[https://flask-impots-withmysql/impots/do]**.

Tout d'abord la route suivante (configs/routes.py) a été exécutée :

```
1.      # les routes de l'application Flask
2.      # racine de l'application
3.      app.add_url_rule(f'{prefix_url}/', methods=['GET'],
4.                       view_func=routes.index)
```

La route de la ligne 3 est dans notre exemple **[https://flask-impots-withmysql/impots/do]**. On voit que la route a été débarrassée de la partie **[https://flask-impots-withmysql/impots]** pour devenir simplement **[/do]**. Pour la partie **[https://flask-impots-withmysql]** c'est normal, le nom du serveur n'est pas repris dans la route. Mais on voit qu'il ne reprend pas non plus l'alias WSGI **[/impots]**. C'est un point important. Même avec un alais WSGI nos routes initiales restent valides.

Maintenant voyons ce que fait la fonction **[index]** de la ligne 4 (configs/routes_without_csrftoken) :

```
1.  # racine de l'application
2.  def index() -> tuple:
3.      # redirection vers /init-session/html
4.      return redirect(url_for("init_session", type_response="html"), status.HTTP_302_FOUND)
```

Ligne 4, on est redirigé vers l'URL de la fonction **[init_session]**. Dans **[configs/routes.py]**, cette fonction a été associée à la route **[/do/init-session/html]** :

```
1.      # init-session
2.      app.add_url_rule(f'{prefix_url}/init-session/<string:type_response>{csrftoken_param}', methods=['GET'],
3.                       view_func=routes.init_session)
```

Ligne 2, dans notre test **[csrftoken_param]** est la chaîne vide. L'application ne gère pas ici de jeton CSRF.

La fonction **[init_session]** est définie de la façon suivante (configs/routes_without_csrftoken) :

```
1.  # init-session
2.  def init_session(type_response: str) -> tuple:
3.      # on exécute le contrôleur associé à l'action
4.      return front_controller()
```

Ligne 4, on commence la chaîne de traitement de l'action **[init-session]**. Cette chaîne se termine de la façon suivante dans **[responses/HtmlResponse]** :

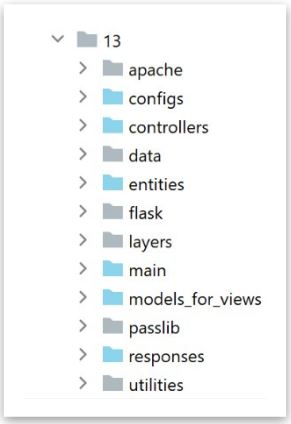
```
1.  …
2.  # maintenant il faut générer l'URL de redirection sans oublier le jeton CSRF s'il est demandé
3.          if config['parameters']['with_csrftoken']:
4.              csrf_token = f"/{generate_csrf()}"
5.          else:
6.              csrf_token = ""
7.
8.          # réponse de redirection
9.          return redirect(f"{config['parameters']['prefix_url']}{ads['to']}{csrf_token}"), status.HTTP_302_FOUND
```

L'action **[init-session]** est une action ADS (Action Do Something) qui se termine pas une redirection vers une vue, ligne 9. C'est là que réside le problème. La fonction **[redirect]** de la ligne 9, n'ajoute pas automatiquement l'alias WSGI à l'URL de redirection. C'est ce que nous montre la copie d'écran ci-dessus. Il manque l'alias /impots dans l'URL objet de la redirection.

La version suivante apporte une solution au problème d'alias WSGI.

38 Exercice d'application : version 18

38.1 Implémentation



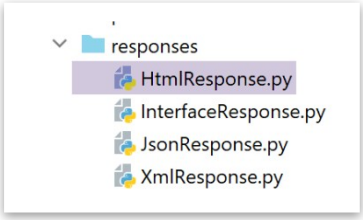
Le dossier **[impots/http-servers/13]** est obtenu initialement par recopie du dossier **[impots/http-servers/12]** puis modifié partiellement.

Nous commençons par ajouter un nouveau paramètre dans le fichier **[configs/parameters]** :

```
1.  …
2.          # token csrf
3.          "with_csrftoken": False,
4.          # bases gérées MySQL (mysql), PostgreSQL (pgres)
5.          "databases": ["mysql", "pgres"],
6.          # préfixe des URL de l'application
7.          # mettre la chaîne vide si on ne veut pas de préfixe ou /préfixe sinon
8.          "prefix_url": "/do",
9.          # url racine du serveur Apache - mettre la chaîne vide pour une exécution en-dehors d'Apache
10.         "application_root": "/impots"
11. …
```

Ligne 10, le paramètre **[application_root]** représentera l'alias WSGI du serveur virtuel Apache.

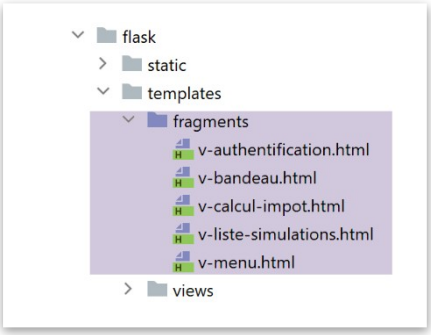
Avec ce paramètre, nous pouvons corriger l'instruction de **[responses/HtmlResponse]** qui avait provoqué l'erreur :




```
1.  …
2.  # maintenant il faut générer l'URL de redirection sans oublier le jeton CSRF s'il est demandé
3.          if config['parameters']['with_csrftoken']:
4.              csrf_token = f"/{generate_csrf()}"
5.          else:
6.              csrf_token = ""
7.
8.          # réponse de redirection
9.          return redirect(f"{config['parameters']['application_root']}{config['parameters']['prefix_url']}
    {ads['to']}{csrf_token}")
10. , status.HTTP_302_FOUND
```

- ligne 9 : nous avons ajouté la racine de l'application au début de l'URL cible de la redirection ;

Il nous faut également corriger tous les fragments pour que les URL qu'ils contiennent commencent par la racine de l'application (ou alias WSGI) :



Le fragment [v-authentification]

```
<!-- formulaire HTML - on poste ses valeurs avec l'action [authentifier-utilisateur] -->
<form method="post" action="{{modèle.application_root}}{{modèle.prefix_url}}/authentifier-utilisateur{{modèle.csrf_token}}"
>

    <!-- titre -->
    <div class="alert alert-primary" role="alert">
        <h4>Veuillez vous authentifier</h4>
    </div>
…

</form>
```

Le fragment [v-calcul-impot]

```
<!-- formulaire HTML posté -->
<form method="post" action="{{modèle.application_root}}{{modèle.prefix_url}}/calculer-
impot{{modèle.csrf_token}}">
    <!-- message sur 12 colonnes sur fond bleu -->
    …

</form>
```

Le fragment [v-liste-simulations]

```
…

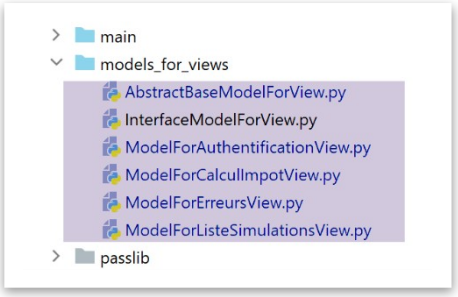
{% if modèle.simulations is defined and modèle.simulations|length!=0 %}
…

<!-- tableau des simulations -->
<table class="table table-sm table-hover table-striped">
    …
    <tr>
        <th scope="row">{{simulation.id}}</th>
        <td>{{simulation.marié}}</td>
        <td>{{simulation.enfants}}</td>
        <td>{{simulation.salaire}}</td>
        <td>{{simulation.impôt}}</td>
        <td>{{simulation.surcôte}}</td>
        <td>{{simulation.décôte}}</td>
        <td>{{simulation.réduction}}</td>
        <td>{{simulation.taux}}</td>
        <td><a href="{{modèle.application_root}}{{modèle.prefix_url}}/supprimer-simulation/{{simulation.id}}
{{modèle.csrf_token}}">Supprimer</a></td>
    </tr>
    {% endfor %}
    </tr>
    </tbody>
</table>
{% endif %}
```

Le fragment [v-menu]

```
<!-- menu Bootstrap -->
<nav class="nav flex-column">
    <!-- affichage d'une liste de liens HTML -->
    {% for optionMenu in modèle.optionsMenu %}
    <a class="nav-link" href="{{modèle.application_root}}{{modèle.prefix_url}}{{optionMenu.url}}
{{modèle.csrf_token}}">{{optionMenu.text}}</a>
    {% endfor %}
</nav>
```

Les fragments ci-dessus utilisent tous le modèle **[modèle.application_root]**. Pour l'instant la clé **[application_root]** n'existe pas dans les modèles générés par les classes de modèles.



La classe **[AbstractBaseModelForView]** qui est la classe parent de toutes les classes générant un modèle devient la suivante :

```
1.  from abc import abstractmethod
2.
3.  from flask import Request
4.  from flask_wtf.csrf import generate_csrf
5.  from werkzeug.local import LocalProxy
6.
7.  from InterfaceModelForView import InterfaceModelForView
8.
9.  class AbstractBaseModelForView(InterfaceModelForView):
10.
11.     @abstractmethod
12.     def get_model_for_view(self, request: Request, session: LocalProxy, config: dict, résultat: dict) -> dict:
13.         pass
14.
15.     def update_model(self, modèle: dict, config: dict):
16.         # calcul du jeton CSRF
17.         if config['parameters']['with_csrftoken']:
18.             csrf_token = f"/{generate_csrf()}"
19.         else:
20.             csrf_token = ""
21.         # mise à jour du modèle passé en paramètre
22.         modèle.update({
23.             # jeton csrf
24.             'csrf_token': csrf_token,
25.             # prefix_url
26.             'prefix_url': config["parameters"]["prefix_url"],
27.             # application_root
28.             'application_root': config["parameters"]["application_root"],
29.         })
```

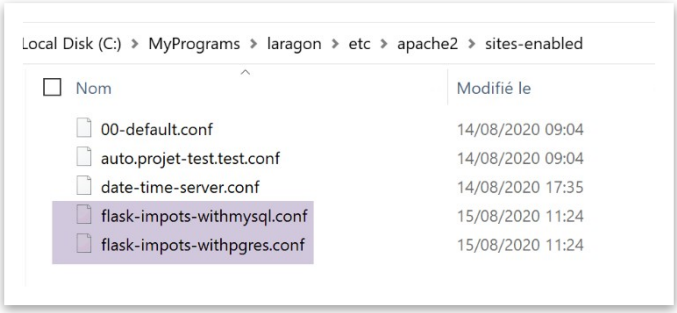
- ligne 15 : la méthode **[update_model]** a pour rôle de mettre dans le modèle des vues :
  - ligne 24 : le jeton CSRF ;
  - ligne 26 : le préfixe des URL ;
  - ligne 28 : la racine de l'application ou alias WSGI ;

Les quatre classes filles font appel à la classe parent avec le code suivant :

```
1.  …
2.          # actions possibles à partir de la vue
3.          modèle['actions_possibles'] = ["afficher-vue-authentification", "authentifier-utilisateur"]
4.
5.          # finition du modèle par la classe parent
6.          super().update_model(modèle, config)
7.
8.          # on rend le modèle
9.          return modèle
```

- ligne 6 : chaque classe fille appelle sa classe parent pour mettre à jour le modèle qu'elle a créé ;

La version 18 est prête. Nous reprenons les deux serveurs virtuels d'Apache de la version 17 et nous les modifions :



Les deux fichiers **[flask-impots-withXX.conf]** ne sont modifiés qu'en un seul point :

```
1.  # dossier du script .wsgi
2.  define ROOT "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/impots/http-servers/13/apache"
3.
4.  # nom du site web configuré par ce fichier
5.  # ici il s'appellera flask-impots-withmysql
6.  # les URL seront du type http(s)://flask-impots-withmysql/chemin
7.  define SITE "flask-impots-withmysql"
8.
9.  # mettre l'adresse IP 127.0.0.1 pour site SITE dans c:/windows/system32/drivers/etc/hosts
10.
11. # mettre ici les chemins des bibliothèques Python à utiliser - les séparer par des virgules
12. # ici les bibliothèques d'un environnement virtuel Python
13. WSGIPythonPath  "C:/Data/st-2020/dev/python/cours-2020/python3-flask-2020/venv/lib/site-packages"
14.
15. # Python Home - nécessaire uniquement s'il y a plusieurs versions de Python installées
16. # WSGIPythonHome "C:/Program Files/Python38"
17.
18. # URL HTTP
19. <VirtualHost *:80>
20.     # avec l'alias / les URL auront la forme /{prefixe_url}/action/...
21.     # avec l'alias /impots les URL auront la forme /impots/{prefixe_url}/action/...
22.     # où [prefixe_url] est défini dans parameters.py
23.     WSGIScriptAlias /impots "${ROOT}/main_withmysql.wsgi"
24.     …
25. </VirtualHost>
26.
27. # URL sécurisées avec HTTPS
28. <VirtualHost *:443>
29.     # avec l'alias / les URL auront la forme /{prefixe_url}/action/...
30.     # avec l'alias /impots les URL auront la forme /impots/{prefixe_url}/action/...
31.     # où [prefixe_url] est défini dans parameters.py
32.     WSGIScriptAlias /impots "${ROOT}/main_withmysql.wsgi"
33.     ..
34.
35. </VirtualHost>
```

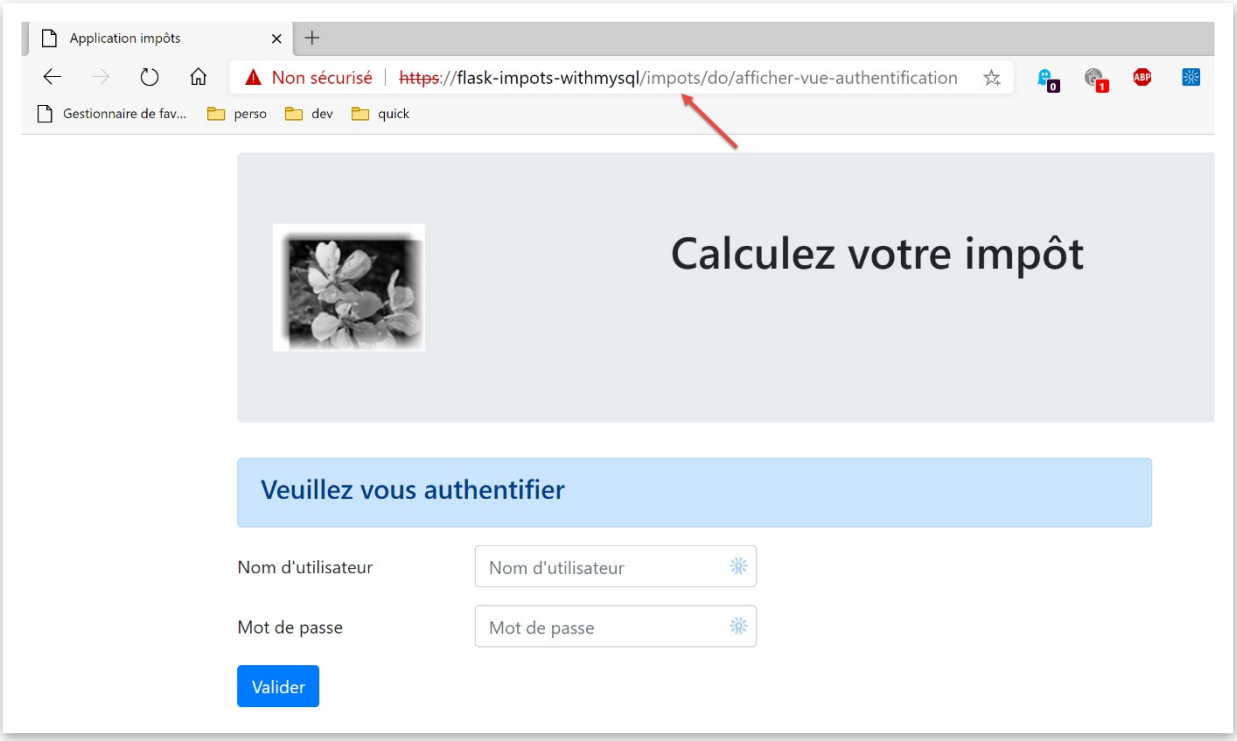
- ligne 12, on utilise maintenant le dossier **[impots/http-servers/13/apache]**.

Nous sommes prêts pour les tests de la version 18 portée sur Apache. La configuration est la suivante :

- l'alias WSGI est /impots dans les deux fichiers de configuration des serveurs virtuels ;
- dans le fichier de paramétrage **[configs/parameters]**, les paramètres sont les suivants :

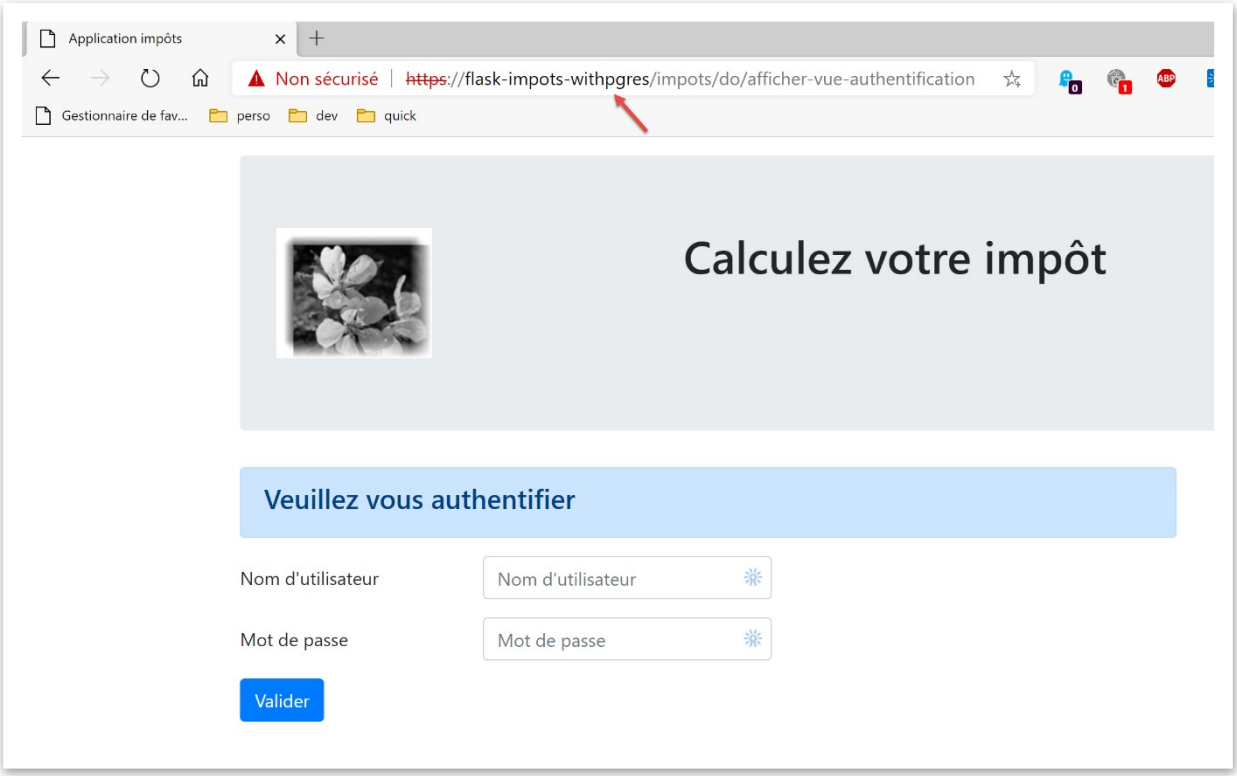
```
1.          # token csrf
2.          "with_csrftoken": False,
3.          # préfixe des URL de l'application
4.          # mettre la chaîne vide si on ne veut pas de préfixe ou /préfixe sinon
5.          "prefix_url": "/do",
6.          # url racine du serveur Apache - mettre la chaîne vide pour une exécution en-dehors d'Apache
7.          "application_root": "/impots"
```

On lance le serveur Apache ainsi que les deux SGBD. On demande l'URL **[https://flask-impots-withmysql/impots/do]**. La réponse du serveur est la suivante :



On obtient bien la vue d'authentification qu'on n'avait pu obtenir dans la version précédente. Le reste de l'application fonctionne normalement.

Maintenant on teste l'autre serveur virtuel. On demande l'URL **[https://flask-impots-withpgres/impots/do]**. La réponse du serveur est la suivante :



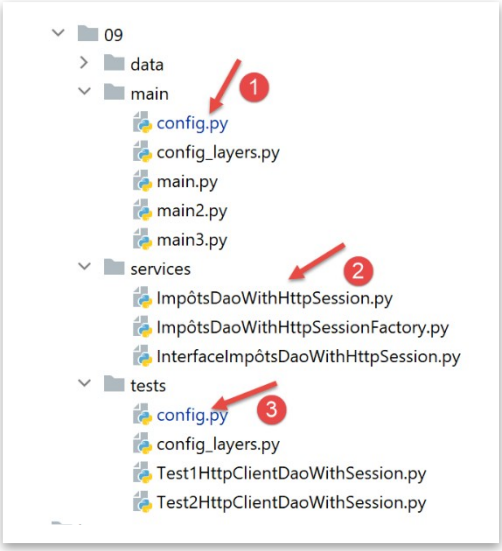
Les notions d'alias WSGI et de préfixe d'URL jouent le même rôle. L'une de ces deux notions est redondante. Ainsi pour préfixer les URL du serveur Apache avec la chaîne **[/impots/do]**, on peut s'y prendre de trois manières :

1 – **[WGSIAlias /impots]** et **[prefix_url='/do']** ;

2 - **[WGSIAlias /]** et **[prefix_url='/impots/do']** ;
3 - **[WGSIAlias /impots/do]** et **[prefix_url='']** ;

38.2 Tests console

On utilise de nouveau les tests console du client **[http-clients/09]** :



- l'URL du serveur doit être modifiée dans les configurations **[1]** et **[3]** ;
- il faut apporter une modification à la couche **[dao]** pour qu'elle supporte le protocole HTTPS du serveur Apache ;

Dans les fichiers **[config]**, l'URL du serveur devient la suivante :

```
1.          "server": {
2.              # "urlServer": "http://127.0.0.1:5000",
3.              # "urlServer": "http://127.0.0.1:5000/do",
4.              "urlServer": "https://flask-impots-withmysql/impots/do",
5.              "user": {
6.                  "login": "admin",
7.                  "password": "admin"
8.              },
9.              "url_services": {
10.                 …
11.             }
12.         },
13.         # mode debug
14.         "debug": True,
15.         # csrf_token
16.         "with_csrftoken": False,
```

- ligne 4 : la nouvelle URL du serveur. Pour la 1ère fois dans ce document, le client utilise le protocole HTTPS ;

La classe **[ImpôtsDaoWihHttpSession]** de la couche **[dao]** évolue de la façon suivante :

```
1.  …
2.      # étape request / response
3.      def get_response(self, method: str, url_service: str, data_value: dict = None, json_value=None):
4.          # [method] : méthode HTTP GET ou POST
5.          # [url_service] : URL de service
6.          # [data] : paramètres du POST en x-www-form-urlencoded
7.          # [json] : paramètres du POST en json
8.          # [cookies]: cookies à inclure dans la requête
9.
10.         # on doit avoir une session XML ou JSON, sinon on ne pourra pas gérer la réponse
11.         if self.__session_type not in ['json', 'xml']:
12.             raise ImpôtsError(73, "il n'y a pas de session valide en cours")
13.
14.         # on ajoute le jeton CSRF à l'URL de service
15.         if self.__csrf_token:
16.             url_service = f"{url_service}/{self.__csrf_token}"
17.
```

```
18.         # exécution de la requête
19.         response = requests.request(method,
20.                                     url_service,
21.                                     data=data_value,
22.                                     json=json_value,
23.                                     cookies=self.__cookies,
24.                                     allow_redirects=True,
25.                                     # pour le protocole https
26.                                     verify=False)
27.
28.         # mode debug ?
29.         if self.__debug:
30.             # logueur
31.             if not self.__logger:
32.                 self.__logger = self.__config['logger']
33.             # on logue
34.             self.__logger.write(f"{response.text}\n")
35.
36.         …
37.
38.         # on rend le résultat
39.         return résultat['réponse']
```

- ligne 26, on ajoute le paramètre **[verify=False]** à cause du protocole HTTPS utilisé par le serveur Apache. Le module **[requests]** (ligne 19) sait gérer nativement le protocole HTTPS. Par défaut, il vérifie la validité du certificat de sécurité que lui envoie le serveur HTTPS et lance une exception si le certificat reçu n'est pas valide. C'est le cas ici où le serveur Apache de Laragon envoie un certificat auto-signé. Pour éviter l'exception, on utilise le paramètre **[verify=False]** pour dire au module **[requests]** de ne pas lancer d'exception. **[requests]** se contente alors d'afficher un avertissement (warning) sur la console.

Ces modifications faites, tous les tests console doivent fonctionner.

39 Conclusion

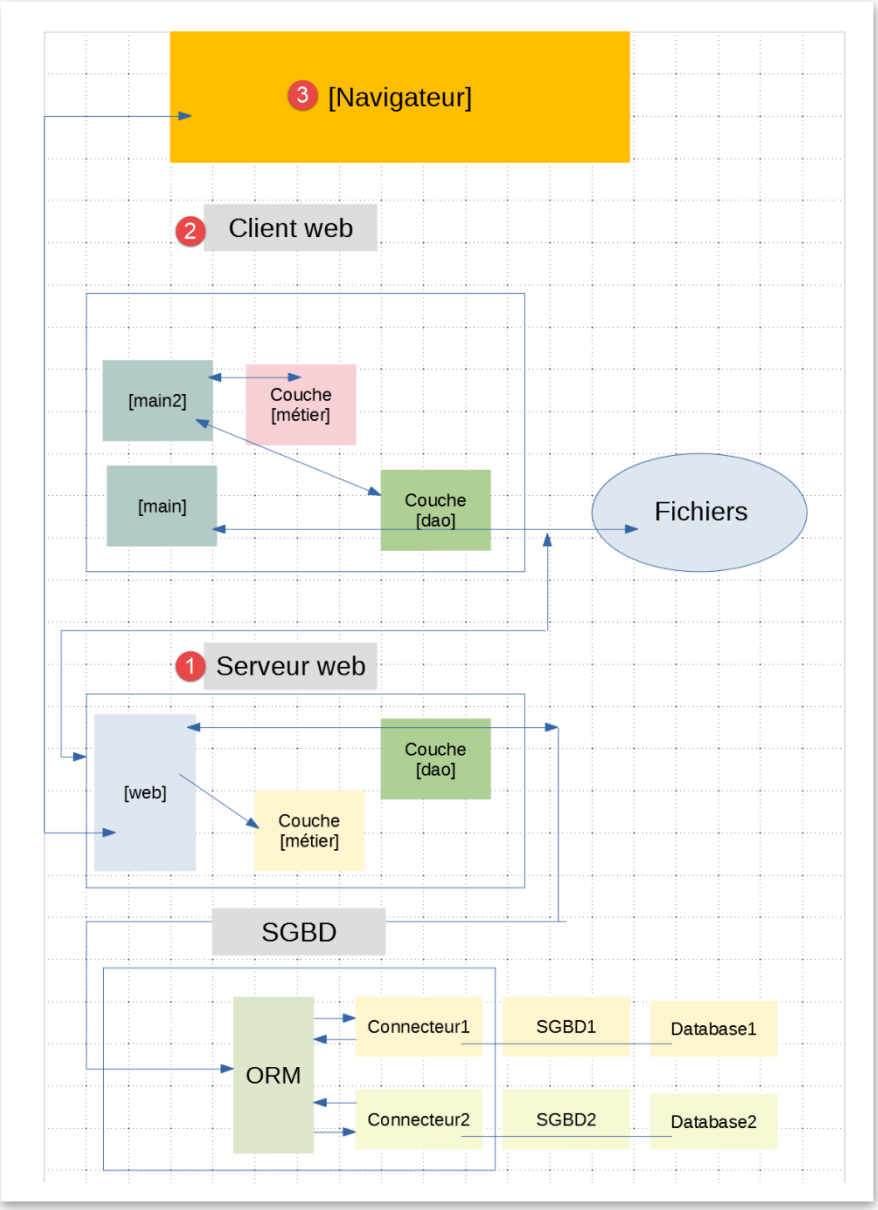
Rappelons le travail fait dans ce document :

Chapitre	Dossier	Contenu
Chapitre 1		Présentation du cours
Chapitre 2		Installation d'un environnement de travail
Chapitre 3	`[bases]`	Les bases du langage Python – structures du langage – type de données – fonctions – affichage console – chaînes de formatage – changement de types – les listes – les dictionnaires – les expression srégulières
Chapitre 4	`[strings]`	Notation des chaînes de caractères – méthodes de la classe <str> - encodage / décodage des chaînes de caractères en UTF-8
Chapitre 5	`[exceptions]`	Gestion des exceptions
Chapitre 6	`[fonctions]`	Portée des variables – mode de passage des paramètres – utiliser des modules – le Python Path – paramètres nommés – fonctions récursives
Chapitre 7	`[fichiers]`	Lecture / écriture d'un fichier texte – gestion des fichiers encodés en UTF-8 – gestion des fichiers jSON
Chapitre 8	`[impots/v01]`	Version 1 de l'exercice d'application, un calcul d'impôt sur les revenus. L'application est déclinée en 18 versions – La version 1 implémente une solution procédurale
Chapitre 9	`[imports]`	Gestion des dépendances d'une application par importation de modules – une méthode de gestion des dépendances est présentée – elle est utilisée dans tout le document – gestion du Python Path
Chapitre 10	`[impots/v02]`	La version 2 de l'application reprend la version 1 en rassemblant toutes les constantes de la configuration dans un fichier de configuration
Chapitre 11	`[impots/v03]`	La version 3 de l'application reprend la version 2 en utilisant des fonctions encapsulées dans un module – la gestion des dépendances est faite par configuration – introduction de fichier jSON pour lire les données nécessaires au calcul de l'impôt et écrire les résultats des calculs
Chapitre 12	`[classes/01]`	Classes – héritage – méthodes et propriétés – getters / setters – constructeur – propriété `[__dict__]`
Chapitre 13	`[classes/02]`	Présentation des classes `[BaseEntity]` et `[MyException]` utilisées dans le reste du document – `[BaseEntity]` facilite les conversions objet / dictionnaire
Chapitre 14	`[troiscouches]`	Architecture en couches et programmation par interfaces. Ce chapitre présente les méthodes de programmation utilisées dans le reste du document
Chapitre 15	`[impots/v04]`	Version 4 de l'application – cette version implémente une solution avec une architecture en couches, la programmation par interfaces, l'utilisation de classes dérivées de `[BaseEntity]` et `[MyException]`
Chapitre 16	`[databases/mysql]`	Installation du SGBD MySQL – connexion à la base de données – création d'une table – exécution d'ordres SQL SELECT, UPDATE, DELETE, INSERT – transaction – requêtes SQL paramétrées
Chapitre 17	`[databases/postgresql]`	Installation du SGBD PostgreSQL – connexion à la base de données – création d'une table – exécution d'ordres SQL SELECT, UPDATE,

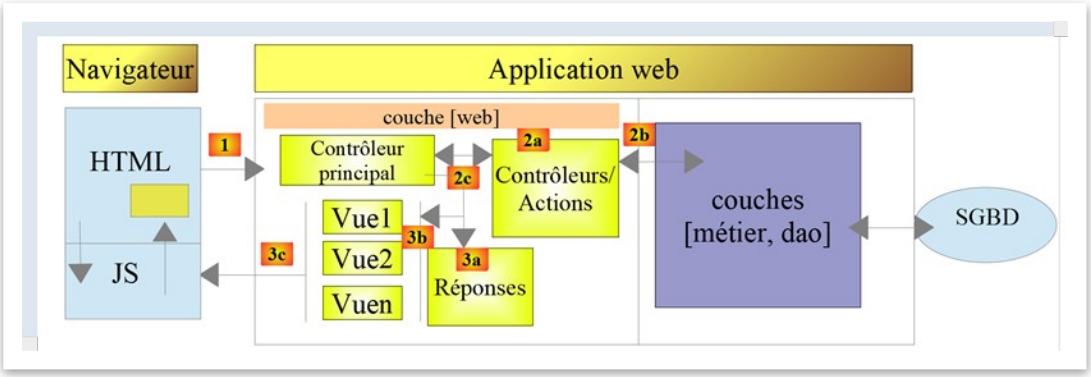
		DELETE, INSERT – transaction – requêtes SQL paramétrées
Chapitre 18	**[databases/anysgbd]**	Ecrire du code indépendant du SGBD
Chapitre 19	**[databases/sqlalchemy]**	L'ORM (Object Relational Mapper) SqlAlchemy – un ORM permet de travailler de façon unifiée avec différents SGBD - mapping classes / tables SQL – opérations sur les classes images des tables SQL
Chapitre 20	**[impots/v05]**	Version 5 de l'application de calcul de l'impôt – Utilisation de l'architecture en couches de la version 04 et de l'ORM SqlAlchemy pour travailler avec les SGBD MySQL et PostgreSQL
Chapitre 21	**[inet]**	Programmation internet – protocole TCP / IP (Transfer Control Protocol / Internet Protocol) - protocoles HTTP (HyperText Transfer Protocol) – SMTP (Simple Mail Transfer Protocol) – POP (Post Office Protocol) – IMAP (Internet Message Access Protocol)
Chapitre 22	**[flask]**	Services web avec le framework web Flask – affichage d'une page HTML – service web jSON – requêtes GET et POST – gstion d'une session web
Chapitre 23	**[impots/http-servers/01]** **[impots/http-clients/01]**	Version 6 de l'exercice d'application - Création d'un service web jSON de calcul de l'impôt avec une architecture multicouche - Ecriture d'un client web pour ce serveur avec une architecture multicouche – programmation client / serveur – utilisation du module **[requests]**
Chapitre 24	**[impots/http-servers/02]** **[impots/http-clients/02]**	Version 7 de l'exercice de l'application – la version 6 est améliorée : le client et le serveur sont multithtreadés – utilitaires **[Logger]** pour loguer les échanges client / serveur – **[SendMail]** pour envoyer un mail à l'administrateur de l'application
Chapitre 25	**[impots/http-servers/03]** **[impots/http-clients/03]**	Version 8 de l'exercice d'application – la version 7 est améliorée par l'usage d'une session
Chapitre 26	**[xml]**	Gestion XML avec le module **[xmltodict]**
Chapitre 27	**[impots/http-servers/04]** **[impots/http-clients/04]**	Version 9 de l'exercice d'application – la version 8 est modifiée pour avoir des échanges client / serveur en XML ;
Chapitre 28	**[impots/http-servers/05]** **[impots/http-clients/05]**	Version 10 de l'exercice d'application – au lieu de traiter N contribuables par N requêtes GET, on utilise une unique requête POST avec les N contribuables dans le corps du POST
Chapitre 29	**[impots/http-servers/06]** **[impots/http-clients/06]**	Version 11 de l'exercice d'application – l'architecture client / serveur de l'application est modifiée : la couche **[métier]** passe du serveur au client
Chapitre 30	**[impots/http-servers/07]**	Version 12 de l'exercice d'application – cette version implémente un serveur MVC (Model – View – Controller) délivrant indifféremment, à la demande du client, du jSON, du XML et du HTML. Ce chapitre implémente les versions jSON et XML du serveur
Chapitre 31	**[impots/http-clients/07]**	Implémentation des clients jSON et XML du serveur MVC de la version 12
Chapitre 32	**[impots/http-servers/07]**	Implémentation du serveur HTML de la version 12 – utilisation du framework CSS Bootstrap –
Chapitre 33	**[impots/http-servers/08]**	Version 13 de l'exercice d'application - Refactorisation du code de la version 12 – gestion de la session avec le module **[flask_session]** et un serveur Redis – utilisation de mots de passe cryptés
Chapitre 34	**[impots/http-servers/09]**	Version 14 de l'exercice d'application – implémentation d'URL avec un

	[impots/http-clients/09]	jeton CSRF (Cross Site Request Forgery)
Chapitre 35	**[impots/http-servers/10]**	Version 15 de l'exercice d'application – refactorisation du code de la version 14 pour gérer deux types d'actions : ASV (Action Show View) qui ne servent qu'à afficher une vue sans modifier l'état du serveur, ADS (Action Do Something) qui font une action qui modifie l'état du serveur – ces actions se terminent toutes par une redirection vers une action ASV – cela permet de gérer correctement les rafraîchissements de page du navigateur client
Chapitre 36	**[impots/http-servers/11]**	Version 16 de l'application – gestion des URL avec préfixe
Chapitre 37	**[impots/http-servers/12]**	Version 17 de l'application – portage de la version 16 sur un serveur Apache / Windows
Chapitre 38	**[impots/http-servers/13]**	Version 18 de l'application – corrige une anomalie de la version 17

Les applications client / serveur du calcul de l'impôt ont implémenté l'architecture suivante :



La couche **[web]** ci-dessus a été implémentée avec une architecture MVC :



Le contenu du document est dense. Le lecteur qui ira jusqu'au bout aura une bonne vision de la programmation web MVC en Python / Flask et au-delà une bonne vision de la programmation web MVC dans d'autres langages.

On pourra trouver des informations complémentaires sur le framework Flask dans le document [https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world] de l'auteur Miguel Grinberg. J'ai lu quelques éléments de son cours et ceux-ci m'ont semblé très pédagogiques. L'auteur présente beaucoup de notions non abordées dans ce document.

1 Avant-propos....2
2 Installation d'un environnement de travail....8
2.1 Python 3.8.1....8
2.2 L'IDE PyCharm Community....10
2.2.1 Introduction....10
2.2.2 Environnement virtuel d'exécution....15
2.2.3 Git....17
2.3 Conventions d'écriture du code Python....27
3 Les bases de Python....28
3.1 Script [bases_01] : opérations élémentaires....28
3.2 Script [bases_02] : chaînes de formatage....30
3.3 Script [bases_03] : changements de types....31
3.4 Script [bases_04]: portée des variables....33
3.5 Script [bases_05] : listes - 1....33
3.6 Script [bases_06] : listes - 2....34
3.7 script [bases_07] : le dictionnaire....35
3.8 script [bases_08] : les tuples....36
3.9 Script [bases_09] : les listes et dictionnaires à plusieurs dimensions....37
3.10 Script [bases_10] : liens entre chaînes et listes....38
3.11 Script [bases_11] : les expressions régulières....39
4 Les chaînes de caractères....43
4.1 Script [str_01] : notation des chaînes de caractères....43
4.2 Script [str_02] : les méthodes de la classe <str>....43
4.3 Script [str_03] : codage des chaînes de caractères (1)....44
4.4 Script [str_04] : encodage des chaînes de caractères (2)....46
5 Les exceptions....48
5.1 script [exceptions_01]....48
5.2 script [exceptions_02]....49
5.3 script [exceptions_03]....51
6 Les fonctions....54
6.1 Script [fonc_01] : portée des variables....54
6.2 Script [fonc_02] : portée des variables....55
6.3 Script [fonc_03] : portée des variables....56
6.4 Script [fonc_04] : mode de passage des paramètres....57
6.5 Script [fonc_05] : ordre d'écriture des fonctions dans un script....57
6.6 Script [fonc_06] : ordre d'écriture des fonctions dans un script....58
6.7 Script [fonc_07] : utilisation de modules....58
6.8 Script [fonc_08] : ajouter des dossiers au [Python Path]....62
6.9 Script [fonc_09] : déclaration du type des paramètres....63

6.10 Script [fonc_10] : paramètres nommés....64
6.11 Script [fonc_11] : fonction récursive....64
6.12 Script [fonc_12] : fonction récursive....65
7 Les fichiers texte....67
7.1 Script [fic_01] : lecture / écriture d'un fichier texte....67
7.2 Script [fic_02] : gérer des fichiers texte encodés en UTF-8....68
7.3 Script [fic_03] : gérer des fichiers texte encodés en ISO-8859-1....69
7.4 Script [json_01] : gestion d'un fichier jSON....70
7.5 Script [json_02] : gestion des fichiers jSON codés en UTF-8....72
8 Exercice d'application – version 1....75
8.1 Le problème....75
8.1.1 Calcul de l'impôt brut....76
8.1.2 Plafonnement du quotient familial....77
8.1.3 Calcul de la décote....77
8.1.4 Calcul de la réduction d'impôts....78
8.1.5 Calcul de l'impôt net....78
8.1.6 Cas des hauts revenus....78
8.1.7 Chiffres officiels....79
8.2 Version 1....80
8.2.1 Le script principal....80
8.2.2 Le module [impots.v01.shared.impôts_module_01]....82
8.2.3 La fonction [get_taxpayers_data]....82
8.2.4 La fonction [calcul_impôt]....83
8.2.5 La fonction [calcul_impôt_2]....84
8.2.6 La fonction [get_décôte]....85
8.2.7 La fonction [get_réduction]....85
8.2.8 La fonction [get_revenu_imposable]....85
8.2.9 La fonction [record_results]....86
8.2.10 Les résultats....86
9 Les imports....88
9.1 Scripts [import_01]....88
9.2 Script [import_02]....90
9.3 Scripts [import_03]....91
9.4 Scripts [import_04]....92
9.5 Scripts [import_05]....94
9.6 Scripts [import_06]....96
9.7 Scripts [import_07]....98
9.7.1 Installation d'un module de portée machine....99
9.7.2 Le script [config.py]....100

9.7.3 Le script [main.py] 101
10 Exercice d'application : version 2 102
11 Exercice d'application : version 3 104
11.1 Le script de configuration [config.py] 105
11.2 Script principal [main.py] 105
11.3 Le module [impots.v02.modules.impôts_module_02] 106
11.4 Lecture des données de l'administration fiscale 106
11.5 Enregistrement des résultats 106
11.6 Modification des fonctions 107
11.7 Résultats 108
12 Les classes et objets 109
12.1 script [classes_01] : une classe Objet 109
12.2 Script [classes_02] : une classe Personne 110
12.3 Script [classes_03] : la classe Personne avec un constructeur 111
12.4 Script [classes_04] : méthode statiques 111
12.5 Script [classes_05] : contrôles de validité des attributs 113
12.6 Script [classes_06] : ajout d'une méthode d'initialisation de l'objet 116
12.7 Script [classes_07] : une liste d'objets Personne 117
12.8 Script [classes_08] : création d'une classe dérivée de la classe Personne 118
12.9 Script [classes_09] : seconde classe dérivée de la classe Personne 119
12.10 Script [classes_10] : la propriété [__dict__] 120
13 Les classes génériques [BaseEntity] et [MyException] 121
13.1 La classe MyException 121
13.2 La classe [BaseEntity] 122
13.2.1 La méthode [BaseEntity.fromdict] 123
13.2.2 La méthode [BaseEntity.asdict] 128
13.2.3 La méthode [BaseEntity.asjson] 131
13.2.4 La méthode [BaseEntity.fromjson] 132
13.2.5 Le script [main] 132
14 Architecture en couches et programmation par interfaces 134
14.1 Introduction 134
14.2 Exemple 1 134
14.2.1 Les entités de l'application 135
14.2.2 Configuration de l'application 139
14.2.3 Tests des entités 140
14.2.4 La couche [dao] 149
14.2.5 La couche [métier] 156
14.2.6 La couche [ui] 159
14.3 Le script principal [main] 161

14.4 Exemple 2....162
14.4.1 La couche [dao]....162
14.4.2 La couche [métier]....163
14.4.3 La couche [ui]....165
14.4.4 Les fichiers de configuration....166
14.4.5 Le script principal [main]....168
15 Exercice d'application - version 4....172
15.1 Version 4 – application 1....173
15.1.1 Les entités....173
15.1.2 La couche [dao]....177
15.1.3 La couche [métier]....182
15.1.4 Tests des couches [dao] et [métier]....184
15.1.5 Script principal....189
15.2 Version 4 – application 2....191
15.2.1 L'interface [InterfaceImpôtsUi]....191
15.2.2 La classe [ImpôtsConsole]....192
15.2.3 Le script principal....193
16 Utilisation du SGBD MySQL....195
16.1 Intallation du SGBD MySQL....195
16.1.1 Installation de Laragon....195
16.1.2 Création d'une base de données....197
16.2 Intallation du package [mysql-connector-python]....201
16.3 script [mysql_01] : connexion à une base MySQL - 1....203
16.4 script [mysql_02] : connexion à une base MySQL - 2....204
16.5 script [mysql_03] : création d'une table MySQL....205
16.6 script [mysql_04] : exécution d'un fichier d'ordres SQL....207
16.7 script [mysql_05] : utilisation de requêtes paramétrées....216
17 Utilisation du SGBD PostgreSQL....219
17.1 Installation du SGBD PostgreSQL....219
17.2 Administrer PostgreSQL avec l'outil [pgAdmin]....223
17.3 Installation du connecteur Python du SGBD PostgreSQL....226
17.4 Portage des scripts MySQL vers des scripts PostgreSQL....227
17.4.1 module [pgres_module]....227
17.4.2 script [pgres_01]....227
17.4.3 script [pgres_02]....228
17.4.4 script [pgres_03]....229
17.4.5 script [pgres_04]....230
17.4.6 script [pgres_05]....233
17.5 Conclusion....234

18 Ecrire du code indépendant du SGBD....235
19 Utilisation de l'ORM SQLALCHEMY....240
19.1 Installation de l'ORM [sqlalchemy]....240
19.2 Scripts 01 : les bases....241
19.2.1 Configuration....241
19.2.2 Script [démo]....241
19.2.3 Le script [main]....243
19.3 Scripts 02 : les mappings de [sqlalchemy]....247
19.4 Scripts 03 : manipulation des entités de la session [sqlalchemy]....250
19.5 Scripts 04 : utilisation d'une base [PostgreSQL]....253
19.6 Scripts 05 : exemple complet....255
19.6.1 L'architecture de l'application....255
19.6.2 Les bases de données....255
19.6.3 Les entités manipulées par l'application....257
19.6.4 Configuration....260
19.6.5 La couche [dao] - 1....265
19.6.6 Initialisation de la base de données....268
19.6.7 La couche [dao] – 2....272
19.6.8 Le script [main_joined_queries]....274
19.6.9 Le script [main_stats_for_élève]....278
20 Exercice d'application : version 5....280
20.1 Application 1 : initialisation de la base de données....280
20.1.1 Le fichier [admindata.json]....281
20.1.2 Création des bases de données....281
20.1.3 Les entitées mappées par [sqlalchemy]....282
20.1.4 Le fichier de configuration de [sqlalchemy]....284
20.1.5 La couche [dao]....285
20.1.6 Configuration de l'application....288
20.1.7 Le script [main] de l'application....289
20.2 Application 2 : calcul de l'impôt en mode batch....294
20.2.1 Architecture....294
20.2.2 Configuration de l'application....296
20.2.3 La couche [dao]....297
20.2.4 Test de la couche [dao]....298
20.2.5 Le script principal....302
20.3 Application 3 : calcul de l'impôt en mode interactif....304
21 Fonctions réseau....306
21.1 Les bases de la programmation internet....306
21.1.1 Généralités....306

21.1.2 Les caractéristiques du protocole TCP....306
21.1.3 La relation client-serveur....307
21.1.4 Architecture d'un client....307
21.1.5 Architecture d'un serveur....307
21.2 Découvrir les protocoles de communication de l'internet....308
21.2.1 Introduction....308
21.2.2 Utilitaires TCP....308
21.3 Obtenir le nom ou l'adresse IP d'une machine de l'Internet....311
21.4 Le protocole HTTP (HyperText Transfer Protocol)....313
21.4.1 Exemple 1....313
21.4.2 Exemple 2....314
21.4.3 Exemple 3....319
21.4.4 Exemple 4....326
21.4.5 Exemple 5....332
21.4.6 Conclusion....338
21.5 Le protocole SMTP (Simple Mail Transfer Protocol)....338
21.5.1 Introduction....338
21.5.2 Création d'une adresse **[gmail]**....338
21.5.3 Installation d'un serveur SMTP....339
21.5.4 Installation d'un lecteur de courrier....346
21.5.5 Le protocole SMTP....354
21.5.6 scripts **[smtp/01]** : un client SMTP basique....358
21.5.7 scripts **[smtp/02]** : un lient SMTP écrit avec la bibliothèque **[smtplib]**....363
21.5.8 scripts **[smtp/03]** : gestion des fichiers attachés....367
21.6 Le protocole POP3....372
21.6.1 Introduction....372
21.6.2 Découverte du protocole POP3....373
21.6.3 scripts **[pop3/01]** : un client POP3 basique....376
21.6.4 scripts **[pop3/02]** : client POP3 avec les modules **[poplib]** et **[email]**....381
21.7 Le protocole IMAP....397
21.7.1 Introduction....397
21.7.2 script **[imap/main]** : client IMAP avec le module **[imaplib]**....400
22 Services web avec le framework Flask....404
22.1 Introduction....404
22.2 scripts **[flask/01]** : premiers éléments de programmation web....405
22.2.1 script **[exemple_01]** : rudiments du langage HTML....406
22.2.2 script **[exemple_02]** : générer un document HTML dynamiquement....411
22.2.3 script **[exemple_03]** : utiliser des fragments de page....414
22.3 scripts **[flask/02]** : service web de date et heure....415

22.4 scripts [flask/03] : services web générant du texte....417
22.4.1 script [main_01]....417
22.4.2 Postman....420
22.4.3 script [main_02]....424
22.4.4 script [main_03]....425
22.5 scripts [flask/04] : informations encapsulées dans la requête....427
22.6 scripts [flask-05] : gestion de la mémoire de l'utilisateur....433
22.6.1 Introduction....433
22.6.2 script [session_scope_01]....435
22.6.3 script [session_scope_02]....439
22.6.4 script [session_scope_03]....441
22.7 scripts [flask/06] : informations partagées par les utilisateurs....443
22.7.1 Introduction....443
22.7.2 script [application_scope_01]....444
22.7.3 script [application_scope_02]....446
22.7.4 script [application_scope_03]....448
22.8 scripts [flask/07] : gestion des routes....449
22.8.1 script [main_01] : routes paramétrées....449
22.8.2 script [main_02] : externalisation des routes....451
23 Exercice d'application : version 6....453
23.1 Introduction....453
23.2 Le serveur web de calcul de l'impôt....454
23.2.1 Version 1....454
23.2.2 Version 2....463
23.2.3 Version 3....465
23.3 Le client web du serveur de calcul de l'impôt....468
23.3.1 Introduction....468
23.3.2 Configuration du client web....471
23.3.3 Le script principal [main]....472
23.3.4 Implémentation de la couche [dao]....473
23.3.5 Exécution....475
23.4 Tests de la couche [dao]....476
24 Exercice d'application : version 7....479
24.1 Introduction....479
24.2 Les utilitaires....480
24.2.1 La classe [Logger]....480
24.2.2 La classe [SendAdminMail]....482
24.3 Le serveur web....483
24.3.1 Configuration....484

24.3.2 Le script principal [main] 485
24.3.3 Le contrôleur [index_controller] 488
24.3.4 Exécution 489
24.4 Le client web 490
24.4.1 La configuration 490
24.4.2 La couche [dao] 491
24.4.3 Le script principal 492
24.4.4 Exécution 494
24.5 Tests de la couche [dao] 496
25 Exercice d'application : version 8 499
25.1 Introduction 499
25.2 Le serveur web 500
25.2.1 La configuration 500
25.2.2 Le script principal [main] 500
25.2.3 Le contrôleur [index_controller] 501
25.3 Le client web 502
25.3.1 La couche [dao] 502
25.3.2 La configuration 505
25.3.3 Le script principal du client 505
25.3.4 Exécution du client 506
25.4 Tests de la couche [dao] 509
26 Du dictionnaire à XML et vice-versa 512
27 Exercice d'application : version 9 516
27.1 Le serveur web 516
27.2 Le client web 518
27.2.1 Le code 518
27.2.2 Test de la couche [dao] du client 520
28 Exercice d'application : version 10 521
28.1 Introduction 521
28.2 Le serveur web 522
28.2.1 Configuration 522
28.2.2 Le script principal [main] 522
28.2.3 Le contrôleur [index_controller] 522
28.2.4 Tests du serveur 524
28.3 Le client web 530
28.3.1 La couche [dao] 530
28.3.2 Le script principal [main] 531
28.3.3 Exécution du client 532
28.3.4 Tests de la couche [dao] du client 532

29 Exercice d'application : version 11....534
29.1 Introduction....534
29.2 Le serveur web....535
29.3 Configuration....535
29.4 Le script principal [main]....535
29.5 Les contrôleurs....537
29.6 Tests Postman....537
29.7 Le client web....539
29.7.1 Configuration des couches du client....540
29.7.2 Implémentation de la couche [dao]....541
29.8 Les scripts [main, main2]....542
29.9 Tests du client....543
30 Exercice d'application : version 12....546
30.1 Architecture MVC....546
30.2 Architecture de l'application client / serveur....548
30.3 L'arborescence du code du serveur....549
30.4 Les URL de service de l'application....550
30.5 Configuration du serveur....553
30.6 Cheminement d'une requête client au sein du serveur....556
30.6.1 Le script [main]....556
30.6.2 Le contrôleur principal [MainController]....560
30.7 Traitement spécifique à une action....562
30.8 Elaboration de la réponse HTTP du serveur....563
30.9 Premiers tests....565
30.10 L'action [authentifier-utilisateur]....569
30.11 L'action [calculer_impot]....572
30.12 L'action [lister-simulations]....576
30.13 L'action [supprimer-simulation]....578
30.14 L'action [fin-session]....580
30.15 L'action [get-admindata]....581
30.16 L'action [calculer-impots]....583
31 Clients web pour les services jSON et XML de la version 12....587
31.1 L'arborescence des scripts des clients....587
31.2 La couche [dao] des clients....588
31.2.1 Interface....589
31.2.2 Implémentation....590
31.2.3 La factory de la couche [dao]....594
31.3 Configuration des clients....594
31.4 Le client [main]....596

31.5 Le client [main2] 599
31.6 Le client [main3] 600
31.7 La classe de test [Test2HttpClientDaoWithSession] 603
32 Le mode HTML de la version 12 607
32.1 Architecture MVC 607
32.2 L'arborescence des scripts du serveur HTML 608
32.3 Présentation des vues 608
32.4 Configuration des vues 610
32.5 La vue d'authentification 612
32.5.1 Présentation de la vue 612
32.5.2 Le fragment [v-bandeau.html] 614
32.5.3 Le fragment [v-authentification.html] 615
32.5.4 Tests visuels 617
32.5.5 Calcul du modèle de la vue 618
32.5.6 Génération des réponses HTML 620
32.5.7 Tests [Postman] 622
32.5.8 Conclusion 625
32.6 La vue de calcul de l'impôt 625
32.6.1 Présentation de la vue 625
32.6.2 Le fragment [v-calcul-impot.html] 627
32.6.3 Le fragment [v-menu.html] 629
32.6.4 Test visuel 630
32.6.5 Calcul du modèle de la vue 631
32.6.6 Tests [Postman] 633
32.7 La vue de la liste des simulations 635
32.7.1 Présentation de la vue 635
32.7.2 Test visuel 638
32.7.3 Calcul du modèle de la vue 639
32.7.4 Tests [Postman] 640
32.8 La vue des erreurs inattendues 641
32.8.1 Présentation de la vue 642
32.8.2 Test visuel 643
32.8.3 Calcul du modèle de la vue 644
32.8.4 Tests [Postman] 645
32.9 Implémentation des actions du menu de l'application 646
32.9.1 L'action [/afficher-calcul-impot] 646
32.9.2 L'action [/fin-session] 648
32.10 Tests de l'application HTML en conditions réelles 650
33 Exercice d'application : version 13 655

33.1 Refactorisation de la configuration de l'application....655
33.2 Refactorisation du script principal `[main]`....659
33.2.1 Les modules `[flask_session]`, et `[redis]`....661
33.2.2 Le serveur Redis....661
33.2.3 Gestion du serveur Redis dans le script principal `[main]`....663
33.2.4 Gestion des routes dans le script principal `[main]`....664
33.3 Refactorisation du contrôleur principal....666
33.4 Gestion des mots de passe cryptés....667
33.5 Tests....669
34 Exercice d'application : version 14....670
34.1 Introduction....670
34.2 Configuration....671
34.3 Implémentation CSRF....671
34.3.1 Le module `[flask_wtf]`....671
34.3.2 Les modèles des vues....672
34.3.3 Les vues....673
34.3.4 Les routes....674
34.3.5 Le contrôleur `[MainController]`....675
34.4 Tests avec un navigateur....676
34.5 Clients console....678
35 Exercice d'application : version 15....685
35.1 Introduction....685
35.2 Implémentation....688
35.2.1 Les nouvelles routes....688
35.2.2 Les nouveaux contrôleurs....689
35.2.3 La nouvelle configuration MVC....691
35.2.4 Les nouveaux modèles....695
35.2.5 Le nouveau contrôleur principal....697
35.2.6 La nouvelle réponse HTML....699
35.3 Tests....700
36 Exercice d'application : version 16....702
36.1 Introduction....702
36.2 La nouvelle configuration des routes....702
36.3 Les nouveaux contrôleurs....706
36.4 Les nouveaux modèles....706
36.5 Les nouveaux fragments....707
36.6 La nouvelle réponse HTML....708
36.7 Tests....709
37 Exercice d'application : version 17....711

37.1 Relocalisation des dépendances de l'application....711
37.2 Tests....713
37.3 Portage d'une application Python / Flask sur un serveur Apache / Windows....713
37.3.1 Sources....713
37.3.2 Installation du module Python mod_wsgi....713
37.3.3 Configuration du serveur Apache de Laragon....715
37.4 Création d'un premier serveur virtuel Apache....722
37.5 Portage de l'application de calcul de l'impôt sur Apache / Windows....726
38 Exercice d'application : version 18....733
38.1 Implémentation....733
38.2 Tests console....738
39 Conclusion....740