

ESIEE Paris – Département Santé Energie Environnement
SEV-5204E - " e-health data : web services and information systems "

Tutoriel Web Services REST

PARTIE 2

4. Conception, développement et test d' un agent serveur simple exposant des ressources uniquement accessibles en lecture (méthode GET).
5. Conception, développement et test d' un agent serveur avancé exposant des ressources enregistrées dans une base de données MySql et accessibles uniquement en mode lecture. Test de la consommation des ressources exposées par l' agent client simple (envoi d'une requête GET).

4 Conception, développement et test d' un agent serveur simple exposant des ressources uniquement accessibles en lecture (méthode GET).

La première partie du tutoriel était essentiellement focalisée sur l'agent client du Web Service à travers l'envoi de requêtes http de type GET permettant de consommer une des deux représentations possibles (json ou html) des ressources exposées par un agent serveur déjà existant. Dans la partie 2, nous allons aborder la conception et la programmation d'un agent serveur. Avant cela, il est nécessaire de rappeler les règles à suivre pour normaliser et simplifier la définition et l'utilisation des URI et de bien comprendre où et comment les scripts des agents serveurs sont hébergés.

Hébergement des scripts côté serveur

Un agent client peut être hébergé n'importe où : sur votre poste de travail personnel comme c'est le cas dans le cadre de ce tutoriel ou sur un hôte distant. Un **agent serveur doit quant-à lui être hébergé côté serveur**. De plus, pour pouvoir être exécuté, le script d'un serveur doit être placé dans répertoire spécifique du système de fichier du serveur. Dans le cadre de ce tutoriel, vous devrez les placer dans un sous-dossier nommé **ws** qui a été créé à cet effet dans le répertoire **public_html** se trouvant à la racine de votre compte utilisateur, sur le serveur **www.gaalactic.fr**. Ainsi, grâce à un paramétrage spécifique du serveur Web, un programme Python nommé **messages.py** placé dans le répertoire **ws** pourra être appelé par un agent client et exécuté via l'URL :

https://www.gaalactic.fr/~votre_login/ws/messages.py

ATTENTION : tout script Python placé en dehors du répertoire **votre_login/public_html/ws** ne pourra pas être exécuté !

Normalisation de la syntaxe des URI et mécanisme de réécriture sous-jacent

Nous avons également vu en cours qu'il existait deux modes possibles pour récupérer le contenu des ressources depuis un serveur Web :

- le **mode statique**, qui permet de réaliser le simple téléchargement d'une **page html** au contenu prédéfini,
- le **mode dynamique**, qui permet de lancer un **programme** (*script Python*, php, programme Java,...) qui construit une **représentation des ressources** dans un **contexte d'exécution** donné.

Or le nom des scripts Python est habituellement complété par l'extension **.py**. Si l'on dispose d'un script Python nommé **messages.py** comme agent serveur d'un Web Service et que l'on suit le formalisme standard de construction des URI " réels " :

scheme://server_name:tcp_port/pathname/resource_name?arg1=value1&arg2=value2&...argN=valueN ,

Donc l'expression de l'URI qui permet de lancer l'exécution à distance du script Python **messages.py** pour qu'il renvoie l'ensemble de la collection des messages devrait être :

<https://www.gaalactic.fr/~lacombea/ws/messages.py> (1),

et l'expression de l'URI " réel " permettant d'accéder au contenu du message destiné aux seuls garçons serait :

<https://www.gaalactic.fr/~lacombea/ws/messages.py?dest=boys> (2)

L'URI N° (2) contient un paramètre **dest** dont la valeur est **boys** qui est envoyé au script **messages.py** et permet de modifier son contexte d'exécution afin de récupérer le seul contenu du message destiné aux garçons.

Cependant les URI utilisés dans la partie N° 1 du tutoriel étaient du type :

<https://www.gaalactic.fr/~lacombea/ws/messages> (3)

<https://www.gaalactic.fr/~lacombea/ws/messages/boys> (4)

Ce formalisme de construction a été détaillé et on rappelle que son objectif est de donner aux clients une représentation explicite de la nature et de la structure hiérarchique des ressources exposées : le terme signifiant le plus à gauche (ici **messages**) correspond au nom de la collection complète (ici l'ensemble des messages) et les termes les plus à droite correspondent successivement à des sous-ensembles de la collection progressivement plus restreints pour cibler, in fine, une des ressources individuelles (ici **boys** pour le message destiné aux seuls garçons).

Les URI N° 1 et 2 correspondent à une structure " réelle ", à savoir celle qui est comprise par les serveurs Web, alors que les URI N° 3 et 4 correspondent à une modalité de représentation " virtuelle ".

Puisque l'on souhaite que les clients finaux du Web Service accèdent aux ressources grâce aux URI " virtuels ", il est nécessaire de réaliser une transposition entre URI virtuels et réels. Ce mécanisme fait partie des fonctionnalités particulières des serveurs Web appelé la **réécriture d'URL**. Sa mise en œuvre est basée sur un ensemble de règles exprimées dans un fichier spécifique (fichier **.htaccess**) placé dans le même répertoire que les scripts serveurs (ici le répertoire **ws**). Le contenu de ce fichier est listé ci-dessous .

nota : la première partie du fichier .htaccess a été volontairement masquée dans cet exemple

```
#####  
##### DEFINITION OF URL REWRITE RULES #####  
#####  
# RULE n° 1 : REWRITES last URI TERM "messages" IN "messages.py"  
# Ex. : URL https://www.gaalactic.fr/~login/ws/messages IS CHANGED IN https://www.gaalactic.fr/~login/ws/messages.py  
RewriteRule messages$ messages.py  
  
# RULE n° 2 : REWRITES last URI term "messages/" FOLLOWED BY ANY TERM CONTAINING LOWER-CASE ASCII LETTERS IN  
"messages.py?dest=term"  
# Ex. : URL https://www.gaalactic.fr/~login/ws/messages/men IS CHANGED IN  
https://www.gaalactic.fr/~login/ws/messages.py?dest=men  
RewriteRule messages/([a-z]+)$ messages.py?dest=$1
```

On y dénombre 2 règles de réécriture qui opèrent chacune 1 type de transformation (en supposant que le login utilisateur est *lacombea*) :

1. <https://www.gaalactic.fr/~lacombea/ws/messages>
est transposé en
<https://www.gaalactic.fr/~lacombea/ws/messages.py>
2. <https://www.gaalactic.fr/~lacombea/ws/messages/men>
est transposé en
<https://www.gaalactic.fr/~lacombea/ws/messages.py?dest=men>

Vous noterez que la deuxième règle est une règle généralisante qui permet, grâce à l'utilisation d'une expression régulière, de réaliser la transposition pour toute suite de caractères minuscules non

accentués constituant le dernier terme (donc 'boys', 'girls', mais pas 'Boys' par exemple), ce qui signifie que si le dernier terme contient une majuscule, la transposition ne se fera pas correctement et le serveur renverra un message d'erreur 404 indiquant que la ressource n'a pas pu être trouvée.

Vous pouvez vérifier que l'utilisation des URI (3) et (4) donne respectivement le même résultat que l'utilisation des URI (1) et (2).

Remarque : un fichier ".htaccess" a été placé dans le répertoire ws afin que **vous n'avez pas dans un premier temps à vous préoccuper des règles de réécriture, à condition de respecter strictement le nom donné au script de l'agent serveur (qui doit être appelé messages.py)**. Si cette condition n'est pas vérifiée, vous recevrez un message et un code d'erreur du serveur.

★★ Travail à faire :

- vérifiez la connexion sur ssh.gaalactic.fr avec votre client ftp (filezilla, winscp,...) à l'aide des identifiants reçus par mail. Contrôlez le contenu du répertoire `public_html` (doit contenir un fichier `index.html` et le répertoire `ws`) et le contenu du répertoire `ws` (doit contenir le fichier `.htaccess`);
- lancez une requête GET sur l'URI ci-dessous et comparez le source html de la page affichée au contenu du fichier `index.html` :

https://www.gaalactic.fr/~votre_login

Comme vous pouvez le constater, le serveur Web hébergé sur l'hôte www.gaalactic.fr a trouvé et renvoyé automatiquement le contenu du fichier **`index.html`** présent dans le répertoire **`public_html`**. Cela est dû à une troisième fonctionnalité spécifique du serveur qui peut être étendue à n'importe quel nom de fichier. Grâce à cette fonctionnalité (paramétrable par l'administrateur du serveur), le serveur Web recherche automatiquement tous les fichiers nommés `index.html` ou `index.py` lorsqu'aucun nom de fichier n'est présent à la fin de l'URI. Donc, si un fichier nommé `index.py` existe dans le répertoire `ws`, son contenu est automatiquement exécuté. C'est ce qui s'est produit dans la partie 1 du tutoriel lorsque vous avez lancé une requête http avec l'URI :

<https://www.gaalactic.fr/~lacombea/ws>

et que vous avez reçu en retour le jeu de données descriptif du Web Service existant.

Ainsi, si vous voulez documenter le Web Service que vous allez développer en renvoyant son jeu de données descriptif, il suffira de placer, dans le répertoire `public_html/ws`, un script Python nommé `index.py` qui produit le jeu de données correspondant.

★★ Travail à faire :

- écrivez le script `index.py` qui génère la chaîne json contenant le jeu de données descriptif de votre futur agent serveur, testez-le et placez-le dans le répertoire `ws` sur le serveur [gaalactic.fr](https://ssh.gaalactic.fr).
- testez le retour du jeu de données à l'aide d'une requête GET sur l'URI :
https://www.gaalactic.fr/~your_login_name/ws

Comment procéder ?

1 - identifiez les types de structures de données Python qui correspondent au contenu de la chaîne json qui sera renvoyée lors de l' exécution du script index.py :

- quel est le type de la structure principale: liste, dictionnaire?
- quels sont les éléments de la structure principale et quel est leur nom ?
- quel type de données ou de structure est contenu dans chaque élément principal: liste, dictionnaire, chaîne, entier?
- etc... jusqu' au niveau le plus bas.

2 – construisez et renseignez la structure de données.

3 – sérialisez, à l'aide de la fonction **json.dumps()** le jeu de données structuré dans une aide de pour créer la chaîne de caractère au format json afin de la faire imprimer par le script à l' aide de la fonction print().

Pour vous aider dans ces différentes tâches, vous pouvez consulter les documentations suivantes:

<https://docs.python.org/3.8/tutorial/datastructures.html#dictionaries>

<https://docs.python.org/3.8/tutorial/introduction.html#lists>

<https://docs.python.org/fr/3.8/library/json.html>

Note :

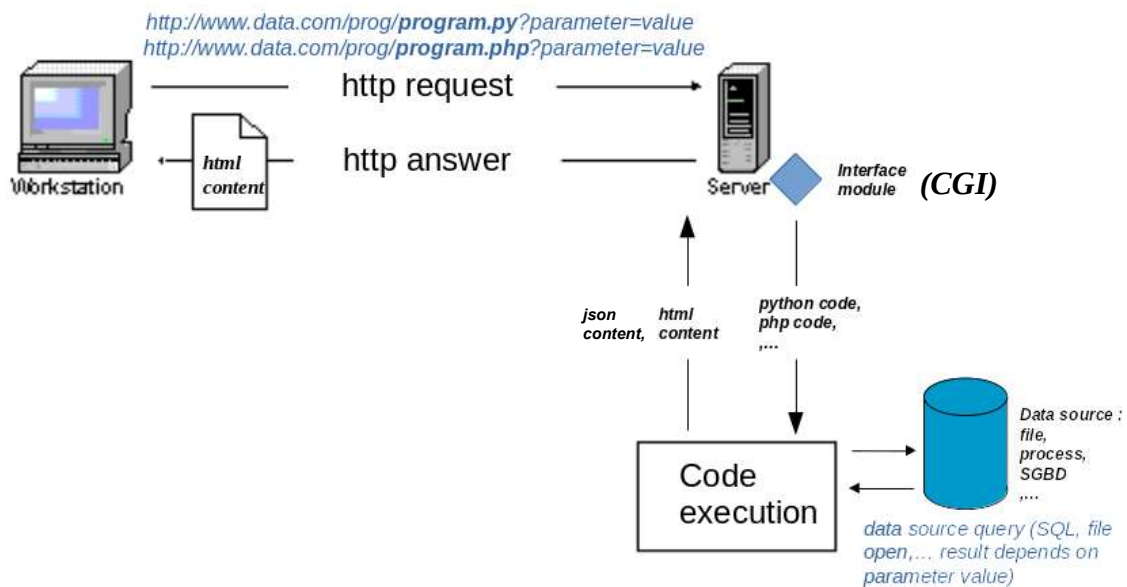
- la structure et le contenu de l' ensemble de données à produire sont les mêmes que ceux donnés au paragraphe 2.1 (page 5). Certaines modifications doivent évidemment être apportées dans l' élément " 01-info ". Vous devez également tenir compte du fait que l'agent serveur que vous allez développer ne produira que des représentations de ressources au format json, ce qui implique que vous devez adapter le contenu des en-têtes spécifiés dans les éléments " produce " et " consume ". Vous devez par contre laisser telle quelle la valeur de l'en-tête " Content type " se trouvant dans l'élément " consume " (application / x-www-form-urlencoded). Vous verrez lors de la réalisation des parties 5 et 6 du tutoriel que cette spécification est obligatoire si l' agent serveur doit recevoir des données http envoyées par les méthodes POST ou PUT.
- vous devez changer les attributs " permissions " du fichier index.py téléchargé sur le serveur gaalactic.fr pour permettre au serveur d' exécuter le programme. Cette modification peut être effectuée dans filezilla ou WinScp (clic droit sur index.py dans la liste des fichiers du serveur, puis ajoutez une autorisation d'exécution (x) à tous les utilisateurs). **N' oubliez pas que les permissions de tous les scripts python placés dans le répertoire ws devront être modifiées pour permettre leur exécution.**
- un chaîne " Content-type: application/json " doit être envoyé par le script index.py, **avant toute autre sortie**, pour remplir l' en-tête de la réponse http afin que l' agent client sache quel type de représentation il reçoit. Cela peut être fait avec une simple commande d' impression:

`print('Content-type: application/json\n')`

(n' oubliez pas le caractère \n car il marque la fin de la définition des en-têtes http)

A ce stade, et avant de vous lancer dans l'écriture du script index.py, il est important de comprendre comment le résultat de l' exécution d'un script est transformé en une réponse http qui sera finalement

renvoyée par le serveur Web à l'agent client. Ceci vous permettra notamment de saisir l'importance de la **recommandation en rouge**. Pour aborder cette question, il faut faire un retour sur l'architecture du Web dynamique qui est rappelée dans la figure ci-dessous. Comme vous pouvez le voir, le script python est exécuté par l'interpréteur python et ses sorties sont transmises au serveur http via un module d'interface. Plusieurs modules d'interface existent, en fonction du type de serveur http et du langage de programmation utilisé pour développer les agents serveurs. Le module utilisé dans notre cas est nommé CGI (pour **Common Gateway Interface**).



La

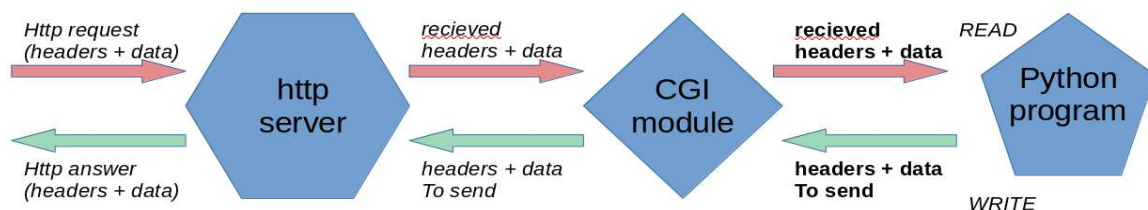


figure ci-dessus précise la façon dont le module intervient dans le transfert des données entre le programme python et le serveur http. Le module CGI possède deux paires d'entrées / sorties, l'une connectée à la paire d'entrée / sortie droite du serveur http et l'autre à la paire d'entrée / sortie du programme Python. Cela signifie que lorsque le programme Python délivre des données sur sa sortie (avec la commande **print()** par exemple), ces données sont transmises par le module CGI au serveur http qui va les utiliser pour " forger " le contenu de la réponse http. Or nous avons vu en cours que l'unité de données de la réponse était structurée en 2 parties principales : l'en-tête et le corps. **Le programme python devra donc délivrer le contenu de spécification des en-têtes de la réponse http avant le contenu du corps de la réponse**, sachant que l'en-tête doit se terminer par deux sauts de ligne. La fonction Python `print()` génère automatiquement le premier saut de ligne mais pour matérialiser correctement la fin des en-têtes, il faut donc imprimer explicitement le deuxième (`print('\n')`) juste après l'impression des données d'en-tête.

Ecriture du script de l'agent serveur simple " messages.py "

Maintenant que vous avez écrit et testé le bon fonctionnement du script `index.py`, vous allez procéder à l'écriture d'une première version du script de l'agent serveur **messages.py**.

Avant de commencer la conception et l'écriture du programme, voyons comment l'agent serveur doit procéder pour récupérer, d'une part les en-têtes http et d'autre part les données, envoyées par l'agent client. Ces deux opérations s'appuient sur deux mécanismes distincts :

- la récupération des données d'en-tête se fait en explorant les **variables d'environnement** qui sont contenues dans un jeu de données accessible via une ressource spécifique du serveur Web nommée **environ**. Il est possible d'accéder à ce jeu de données à l'aide de la librairie **os**. Voici deux exemples de commandes qui impriment le contenu de deux variables d'environnement correspondant à l'en-tête " Accept " et à la méthode (GET, POST,...) de la requête http lancée par l'agent client :

```
print(os.environ['HTTP_ACCEPT']) # Imprime le contenu de l'en-tête
                                # "Accept" de la requête http

print(os.environ['REQUEST_METHOD']) # Imprime la valeur de la méthode
                                    # de la requête http
```

*Note : Pour pouvoir utiliser ces variables d'environnement, vous devez importer le module " os " dans votre script : **import os***

- le second utilise une méthode de la librairie **cgi** nommée **FieldStorage()** qui permet de récupérer simultanément l'ensemble des données envoyées via la requête http du client, qu'elles aient été passées en tant que paramètre d'URI, ou insérées dans le corps de la requête¹.

*Note : la méthode **FieldStorage()** retourne une ressource dont le contenu est un peu difficile à manipuler. Pour faciliter la récupération des données http, une fonction nommée **returnHttpData()** est mise à votre disposition pour vous aider dans cette tâche (cf. Code ci-dessous). Cette fonction analyse le contenu de la ressource et produit le dictionnaire **httpData** contenant les données http envoyées par l'agent client. La structure de ce dictionnaire est la suivante :*

{ "data1name" : "data1value", "data2name" : "data2value", .. "dataNname" : "dataNvalue" }

Dans le cas du tutoriel, le contenu du dictionnaire pour une requête http de type GET sera :

- { }* si l'URI était **https://www.gaalactic.fr/~lacombea/ws/messages** (dictionnaire vide)
- { 'dest' : 'boys' }* si l'URI était **https://www.gaalactic.fr/~lacombea/ws/messages/boys**

```
# Code à placer dans le script serveur messages.py pour récupérer les données http
import cgi
import cgitb # Librairie permettant d'avoir des retours cgi explicites en cas d'erreur
cgitb.enable() # Activation des retours d'erreur cgi

def returnHttpData(): # Fonction retournant le dictionnaire qui contient les données http
    formData = cgi.FieldStorage()
    httpData = {}
    httpDataKeys = []
    httpDataKeys = list(formData)
    for key in httpDataKeys:
        httpData[key] = (formData[key].value)
    return httpData
```

¹ Ce cas sera vu dans la partie N° 3 du tutoriel.

★★ Travail à faire :

- écrivez le script `messages.py` qui donne accès à la collection complète ou à des messages individuels, selon l'URI utilisé par un agent client qui envoie une requête GET. Ce programme doit :
 - renvoyer la collection complète des messages au format json lorsque l'URI est :
`https://www.gaalactic.fr/~login/ws/messages`
 - renvoyer un message individuel au format json lorsque l'URI est, par exemple :
`https://www.gaalactic.fr/~login/ws/messages/men`
 - vérifier la présence de l'en-tête `Accept` dans la requête envoyée par le client
 - vérifier si la valeur de l'en-tête `Accept` est bien égale à `application/json`
 - envoyer la chaîne `Content-type: application/json\n` afin de préciser à l'agent client que le format de représentation des ressources renvoyées est du json
 - vérifier la validité de la méthode de la requête http (GET uniquement accepté)
 - gérer les exceptions et renvoyer des messages d'erreur compréhensibles au client :
 - si la méthode http est interdite (autre que GET)
 - si l'en-tête http `Accept` n'a pas été envoyé par le client ou n'est pas celui attendu
 - si l'URI utilisé par le client ne correspond à aucune ressource individuelle disponible
 - etc...

Note : ce programme Python est un agent serveur simplifié. Le contenu des ressources n'est donc, dans un premier temps, pas enregistré dans une base de données. Pour simuler le fonctionnement du script final tel qu'il sera vu dans la partie N° 3, vous allez donc utiliser une structure de données construite à la demande à chaque exécution du script. Cette structure de données peut être constituée sous la forme d'une liste contenant autant de dictionnaires que de groupes destinataires (c'est-à-dire `men`, `women`, `girls`, `boys`...). Chacun de ces dictionnaires est composé de 2 éléments :

- 1 élément " `dest` " dont le contenu est le nom d'un groupe (ie. `men`, `women`,...)
- 1 élément " `text` " dont le contenu est le texte du message (c.-à-d. `Hello men !`, `Hello women !`...)

La liste contenant la collection complète des messages a donc la forme suivante :

`messages = [{ 'dest': 'men', 'text': 'Hello men !' }, { 'dest': 'women', 'text': 'Hello women !' }]`

1/ Pour renvoyer la représentation json de la liste complète des messages demandés avec une méthode GET utilisant l'URI:

`https://www.gaalactic.fr/~login/ws/messages`

le script Python doit seulement sérialiser la liste complète et renvoyer la chaîne json sérialisée.

2 / Pour récupérer un message individuel demandé avec une méthode GET à l' aide de l' URI:

<https://www.gaalactic.fr/~login/ws/messages/men>

le script Python doit parcourir la liste et regarder dans chacun des dictionnaires pour vérifier si la valeur de l'élément " dest " est égale à la valeur " men " de la donnée http " dest " récupérée dans le dictionnaire httpData. Il doit ensuite renvoyer la chaîne json obtenue par sérialisation de ce dictionnaire.

Nota : pour décider s'il doit renvoyer la représentation de la liste complète des messages ou rechercher le message correspondant au nom du groupe destinataire reçu, le script doit simplement vérifier la taille du dictionnaire contenant les données http (**httpData**)

- pas de donnée http reçue → taille de httpData == 0
- donnée http dest reçue → taille de httpData == 1


5 - Conception, développement et test d' un agent serveur avancé exposant des ressources enregistrées dans une base de données MySql et accessibles uniquement en mode lecture. Test de la consommation des ressources exposées par l' agent client simple (envoi d'une requête GET).

A ce stade, vous avez développé les deux parties d'un Web service avec, d'un côté, l'agent serveur qui expose un ensemble de messages dont on peut récupérer la représentation intégrale ou individuelle au format json et de l'autre un agent client qui consomme les ressources exposées par l'agent serveur à l'aide de requêtes http utilisant la méthode GET. Mais les données constitutives de ces ressources ont une durée de vie limitée à la durée d'exécution du script. Elles ne sont donc pas persistantes. Pour pouvoir les conserver dans le temps, il est nécessaire de les enregistrer, soit sous la forme de fichiers (au format json, csv, ou même binaire lorsqu'il s'agit d'images, sons ou autres types de médias) soit dans une base de données structurée.

Vous allez donc voir maintenant comment créer et enregistrer le contenu des ressources sous la forme de données structurées dans une base de données relationnelle et comment l'agent serveur interagit avec le serveur de bases de données pour exécuter les opérations spécifiées par le client à travers l'ensemble des requêtes permettant de créer, lire, mettre à jour et effacer ces contenus.

La figure ci-dessous présente le modèle relationnel de la base de données que vous allez utiliser à partir de maintenant. Celle-ci est constituée de deux tables :

- une table nommée " messages " qui contient les données correspondant aux messages destinés aux différents groupes d'individus (men, women, girls,...) ;
- une table nommée " users " qui contient une liste d'utilisateurs autorisés à accéder au contenu de la table " messages ", avec des droits d'accès différents selon la nature des opérations réalisées (lecture, écriture, mise à jour ou effacement).



tutoWebServices messages	
dest	: varchar(50)
text	: varchar(50)



tutoWebServices users	
idUser	: int(11)
loginUser	: varchar(50)
pwdUser	: varchar(50)
rights	: enum('read','all','','')

On retrouve dans la table " messages " les deux attributs " dest " et " text " correspondant aux données des ressources exposées par le web service. La table " users " contient quant-à elle les logins de deux utilisateurs valant respectivement " Joan " et " Mary ", le premier ayant seulement le droit de lire le contenu de la table messages (droit " read ") et le second ayant tous les droits (" all " qui correspond à " create " + " read " + " update " + " delete "). La première opération à réaliser va consister à créer et remplir les deux tables " messages " et " users " avec leurs contenus respectifs.

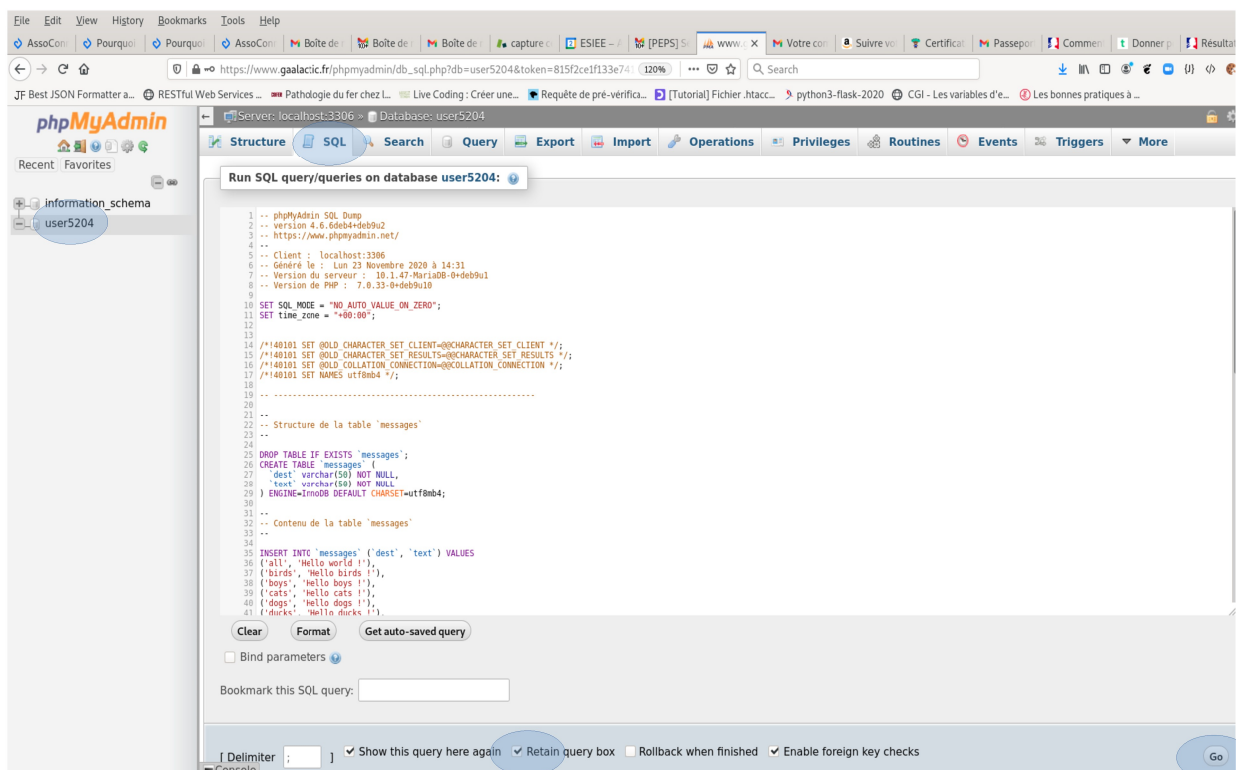
★★ Travail à faire :

- connectez-vous au serveur de bases de données hébergé sur www.gaalactic.fr à l'aide de l'outil " phpMyAdmin " qui permet de gérer un serveur MySQL à distance depuis le Web :

<https://www.gaalactic.fr/padmin>

(utilisez le login et le mot de passe qui vous ont été envoyés par mail)

- récupérez et ouvrez le fichier *WS_tutorial_tables_creation.sql* se trouvant dans le dossier TP 2 du cours sur BlackBoard, copiez puis collez le contenu (" CTR-V " ou " Pomme-V " sur Mac) dans la console SQL de phpMyAdmin puis exécutez le :



Attention :

- pensez à sélectionner la base de donnée portant votre nom en cliquant sur son nom dans la fenêtre de gauche ;
- cochez la case " Retain query box " pour pouvoir conserver le contenu des requêtes dans la fenêtre après exécution (très utile quand on veut récupérer le code par " copier/coller " après exécution).
- Cliquez à nouveau sur le nom de la base de données et observez le résultat de la création des 2 tables.

- Remplacez vous dans la console SQL et listez le contenu des 2 tables (attention, utiliser les quotes inversées pour entourer le nom des tables et des attributs (ALTGR-7 suivi d'un ESPACE sur clavier PC azerty) :

SELECT * FROM `messages` ;

SELECT * FROM `users` ;

Une fois ce travail préliminaire de création et de remplissage des tables réalisé, nous allons voir comment les données contenues dans la base peuvent être récupérées par le script python de l'agent serveur " messages.py ". Dans un premier temps nous ne nous intéresserons qu'au seul accès en lecture (Read) des ressources, à savoir celui qui, selon l'architecture REST est déclenché par une requête http utilisant la méthode GET.

Ce nouveau script est proche, dans son fonctionnement global, du script de l'agent serveur simple développé précédemment. La seule différence est qu'il doit lancer des requêtes SQL vers une base de données pour en extraire le contenu recherché au lieu de le récupérer dans une structure de données python créée à cet effet dans le script de l'agent serveur simple. Seule cette partie du programme sera donc à modifier.

Pour pouvoir lancer des requêtes SQL sur un serveur MySql depuis un script Python, il faut s'appuyer sur une librairie de classes et de fonctions spécifiques. Le package choisi et installé sur le serveur www.gaalactic.fr se nomme " pymysql " et il est donc nécessaire de l'importer dans le script de l'agent serveur " messages .py ".

Par ailleurs, de la même manière que vous avez dû vous connecter au serveur MySql et sélectionner la base de données sur laquelle vous souhaitiez travailler à partir du client phpMyAdmin, l'agent serveur messages.py devra, dans un premier temps, se connecter à la base de données. Ce n'est qu'après une connexion réalisée avec succès que les requêtes SQL pourront être exécutées. Nous verrons par la suite comment réaliser une telle connexion, puis comment la fermer une fois que les requêtes et la récupération des données ont été effectuées.

Pour faciliter l'écriture du code Python de l'agent serveur, il faut avoir préalablement listé et testé les différentes requêtes SQL qui seront lancées sur la base de données pour récupérer les ressources à partir des spécifications transmises par la requête http du client. Nous avons vu précédemment qu'il fallait pouvoir récupérer l'intégralité de la collection des messages ou bien un seul d'entre-eux selon qu'un groupe destinataire a été précisé ou pas. Pour réaliser ces deux types d'opérations, deux types de requêtes sont nécessaires. La première correspond à l'expression :

SELECT * FROM `messages`

La deuxième applique une restriction sur la réponse à l'aide de la clause WHERE portant sur l'attribut " dest " dont la valeur va servir à filtrer la liste des tuples pour ne garder que celui pour lequel la valeur de l'attribut " dest " correspond à une valeur passée en paramètre :

SELECT * FROM `messages` WHERE `dest` = 'valeur'

Attention : des " quotes simples " (touche 4 du clavier azerty sur PC) doivent entourer le terme **valeur** car le contenu de l'attribut " dest " est du type chaîne de caractères.

Une fois que le type des requêtes SQL et leur expression sont complètement définis il ne reste plus qu'à " forger " et exécuter ces requêtes à l'intérieur du script serveur " messages.py " et à récupérer le

contenu renvoyé par le SGBD pour finalement retourner la représentation json à l'agent client via la réponse http.

★★ Travail à faire :

faites une copie de sauvegarde du script " messages.py " existant sous un nom différent, de manière à conserver la première version de l'agent client.

- en vous aidant des indications données dans la note ci-après et de la documentation disponible à l'adresse :

<https://pymysql.readthedocs.io/en/latest/user/index.html>

- écrivez une première version de l'agent serveur messages.py qui n'accepte que des requêtes http de type GET et qui récupère la collection intégrale des messages dans la base de données et renvoie sa représentation au format json ;
- testez l'envoi de requêtes GET à l'aide du client RESTED dans un premier temps puis utilisez l'agent client simple écrit dans la partie 3

Pour rappel, l'intégralité de la collection des messages se récupère via l'URI :

<https://www.gaalactic.fr/~login/ws/messages>

- étendez les fonctionnalités du programme pour qu'il soit capable de renvoyer soit un message individuel, soit la collection intégrale des messages, selon que le groupe destinataire a été précisé ou pas dans l'URI de la requête :

<https://www.gaalactic.fr/~login/ws/messages/women> (par exemple)

ou

<https://www.gaalactic.fr/~login/ws/messages>

- faites en sorte que le programme renvoie un message indiquant qu'aucune ressource n'a été trouvée lorsque le nom du groupe destinataire n'a pas été trouvé dans la table des messages

Note :

ne pas oublier de vérifier les prérequis suivants :

- fin des lignes au format Linux (LF) obligatoire dans les scripts
- droits d'exécution pour tous les utilisateurs sur les scripts déposés sur le serveur
- lignes de directive (ligne N°1 : shebang line + ligne N°2 : spécification du jeu de caractères utf-8) obligatoires en début de script
- envoi de l'en-tête " Content-type: application/json " par le script serveur avant toute autre sortie

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Dec  2 17:34:57 2020
@author: auteur
"""
print('Content-type: application/json\n')
```

pour importer la librairie pymysql :

```
import pymysql
```

pour créer une ressource de connexion à votre base de données sur le serveur MySQL :

```
# Adaptez le contenu en bleu à votre cas particulier
connection = pymysql.connect(host='localhost',
                             user='your_login',
                             password='your_password',
                             db='your_login',
                             charset='utf8mb4',
                             port=3306,
                             cursorclass=pymysql.cursors.DictCursor)
```

pour lancer une requête de récupération de l'intégralité du contenu dans la table " messages " :

```
with connection.cursor() as cursor:
    sql = "SELECT * FROM `messages`"
    cursor.execute(sql)
    result = cursor.fetchall()
```

Crée la ressource de connexion
Syntaxe de la requête SQL
Exécution de la requête SQL
Réponse récupérée dans une liste

pour lancer une requête de récupération du message individuel destiné aux femmes dans la table " messages " (en supposant que la variable " recipient " a pour valeur " women ") :

```
with connection.cursor() as cursor:
    sql = "SELECT * FROM `messages` WHERE `dest` = '"+recipient+"'"
    cursor.execute(sql)
    result = cursor.fetchall()
```

on constate que la seule différence entre les deux requêtes réside dans la présence ou pas de la clause WHERE. On peut donc organiser le code pour n'avoir qu'une seule requête permettant de récupérer soit l'intégralité des messages, soit un message individuel selon qu'un groupe destinataire a été précisé dans la requête http du client. En effet, si un groupe destinataire a été précisé par le client, le dictionnaire " httpData " construit à l'aide de la fonction returnHttpData() contiendra 1 élément dont la clé est " dest " et la valeur est égale au nom du groupe (" women " par exemple). Par contre, si aucun groupe destinataire n'a été précisé, le dictionnaire " httpData " sera vide (taille = 0). Il suffit donc de faire un test sur la taille du dictionnaire pour savoir s'il faut rajouter ou pas la cause WHERE dans l'expression de la requête SQL. Notez que si le groupe destinataire spécifié dans la requête http n'existe pas dans la table, la liste result sera vide (taille = 0).

Implémentation possible :

```
if len(httpData) == 0:
    clauseWhere = "" # Chaîne vide
else:
    recipient = httpData['dest'] # Récupération du groupe destinataire
    clauseWhere = "WHERE `dest` = '"+recipient+"'"
    with connection.cursor() as cursor:
        sql = "SELECT * FROM `messages` "+clauseWhere
        cursor.execute(sql)
    result = cursor.fetchall()
```

Pour fermer la connexion à la base MySQL :

```
connection.close()
```

Attention : la fermeture de la connexion après exécution de la requête est nécessaire car elle permet de libérer la ressource. Mais si plusieurs requêtes sont lancées successivement sur la même base, il ne faut fermer la connexion qu'après exécution de l'ensemble des requêtes. Nous aborderons ce cas dans la dernière partie du tutoriel.

Pour aller plus loin :

à ce stade vous constatez, bien que vous n'avez pour l'instant traité que le cas de la lecture et de la récupération de contenu dans la base de données déclenchée par une requête de type GET, que le script " messages.py " de l'agent serveur contient un nombre assez important de lignes et devient plus difficile à gérer.

Or il reste encore à ajouter les parties qui devront traiter les opérations de création, mise à jour et suppression déclenchées par les méthodes http de type POST, PUT et DELETE. Il est temps, si vous n'avez pas déjà commencé, d'aménager le programme en le découpant en un enchaînement d'opérations unitaires réalisées par des fonctions spécialisées dont le code peut être déporté dans une librairie spécifique. En plaçant le code de ces fonctions dans un fichier nommé **wslib.py** copié sur le serveur dans un sous-répertoire **lib** créé dans le répertoire **ws**, vous pourrez importer les fonctions qu'elle contient grâce aux instructions suivantes :

```
sys.path.insert(1, '/home/votre_login/public_html/ws/lib') # Remplacer par votre login
import wslib
```

puis les appeler dans le script messages.py en les préfixant avec le nom de la librairie :

```
wslib.nom_de_fonction(...)
```

La librairie peut contenir à ce stade les fonctions suivantes :

- une fonction ou un ensemble de fonctions chargées de tester la validité de la requête http (contrôle du contenu de l'en-tête et de la méthode http par rapport à ce qui est attendu et autorisé), renvoyant des messages personnalisés au client puis stoppant l'exécution du script si les contrôles échouent (vous pouvez utiliser pour cela la méthode `sys.exit()` (importer le module `sys`) ;
- la fonction **returnHttpData()**, qui retourne les données envoyées par la requête http du client ;
- une fonction **connect()** réalisant la connexion à la base de données et retournant la ressource nécessaire au lancement des requêtes SQL ;
- une fonction **readResource()** lançant la requête SQL de lecture du contenu de la table " messages " et retournant le résultat sous la forme d'une liste qui contient soit l'intégralité des messages, soit un message individuel, soit une liste vide selon qu'un groupe destinataire a été spécifié ou non dans les données de la requête http et que ce groupe destinataire existe ou pas dans la table " messages ".