Tutoriel Web Services REST

PARTIE 3

6. Mise en œuvre de requêtes http utilisant la méthode DELETE pour l'effacement du contenu des ressources et implémentation d'une procédure d'authentification des utilisateur.

7. Mise en œuvre des requêtes POST et PUT pour la création et la mise à jour du contenu des ressources.

6 – Mise en œuvre de requêtes http utilisant la méthode DELETE pour l'effacement du contenu des ressources et implémentation d'une procédure d'authentification des utilisateur.

Jusqu'à présent, la seule opération que vous avez réalisée consistait à consommer la représentation de ressources distantes à l'aide de requêtes http de type GET. Dans les deux parties restantes du tuturiel, vous allez apprendre comment modifier le contenu d'une collection à l'aide des 3 méthodes DELETE, POST et PUT. Il est important de se rappeler que, selon les spécifications de l'architecture REST, chacune des méthodes ne peut être utilisée que dans un objectif unique :

- une requête utilisant la méthode **GET** devra être utilisée **seulement pour consommer** le contenu d'une ressource individuelle ou d'une collection ;

- une requête utilisant la méthode **DELETE** devra être utilisée **seulement pour supprimer** le contenu d'une des ressources individuelles de la collection ;

- une requête utilisant la méthode **POST** devra être utilisée **seulement pour créer** (ajouter) le contenu d'une nouvelle ressource individuelle à la collection ;

- une requête utilisant la méthode **PUT** devra être utilisée **seulement pour mettre à jour** une ressource individuelle existante.

Une fois ce cadre posé, on peut séparer les 4 méthodes en deux catégories distinctes : les méthodes GET et DELETE d'un côté et les méthodes POST et PUT de l'autre. En effet, pour agir sur une ressource distante avec les deux premières méthodes (que ce soit pour en récupérer ou en effacer le contenu), on a seulement besoin de préciser le terme identifiant la ressource en le rajoutant à la fin de l'URI. Par exemple une requête http GET sur l'URI :

https://hwww.gaalactic.fr/ws/~login/ws/messages/**women**

permet de récupérer la représentation json du message destiné aux femmes. Si l'on souhaite supprimer cette même ressource, il suffira de lancer une requête http utilisant la méthode DELETE en utilisant le même URI sans avoir à préciser quelqu' autre information que ce soit. Nous verrons comment on doit traiter le cas des méthodes POST et PUT dans la dernière partie du tutoriel.

Concentrons nous donc pour l'instant sur la méthode DELETE et notons qu'elle ne devra pas permettre de supprimer plus d'une ressource individuelle à la fois.

Supposons que l'on veuille supprimer la ressource contenant le message destiné aux hommes. L'URI que doit utiliser l'agent client pour demander au serveur d'accéder au contenu de cette ressource est :

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

La réception d'une requête http de type DELETE par l'agent serveur doit déclencher l'exécution d'une requête SQL dont l'expression est :

DELETE FROM `messages` WHERE `dest` = 'men'

Attention : pour que la suppression demandée par une requête SQL DELETE soit effective, elle doit obligatoirement être confirmée par l'envoi d'une instruction ***COMMIT*** (voir la note page 29).

Jusqu'à présent, le script Python messages.py n'avait eu à servir que des requêtes http de type GET. Il doit à présent être capable de différentier les deux types de requêtes (GET ou DELETE) et choisir en conséquence le type de requête SQL (SELECT ou DELETE) à lancer vers le SGBD. Par ailleurs il est d'usage de retourner un compte-rendu sur l'opération exécutée par le serveur (la suppression de la ressource a-t-elle été réalisée ou pas) ainsi que l'état de la collection après suppression du contenu.

De son côté, l'agent client doit pouvoir lancer des requêtes http de type DELETE. Dans un premier temps, le test des requêtes http sera fait à l'aide de l'extension RESTED. Pour réaliser cela à partir du script client Python, il suffira de lancer la requête en utilisant la méthode *requests.delete()* à la place de *requests.get()*.

Attention : l'envoi de requêtes de type DELETE nécessite que le client envoie une **directive d'en-tête " Content-type "** qui sert à préciser sous quel format les données http sont envoyées (voir la façon de faire dans la note page 29).

★★ Travail à faire :

- faites une copie de sauvegarde de la version précédente du script " messages.py "
- en vous inspirant de la structure de la fonction *readResource()* exécutant des requêtes SQL de type SELECT écrite dans la partie 5, ajoutez une fonction *deleteRessource()* dans le script Python de l'agent " messages.py " (*ou dans la librairie si vous en avez créé une*) qui lance des requête SQL de type DELETE pour effacer les données de la table " messages " correspondant à une ressource individuelle ;
- ajoutez dans le programme principal un test pour empêcher de lancer l'exécution de la fonction deleteResource() si un nom de ressource individuelle n'a pas été précisé dans l'URI par le client ; dans ce cas, le programme doit renvoyer un message rappelant qu'il n'est pas possible d'effacer toutes les ressources à la fois ;
- ajoutez dans le programme principal une structure de test permettant d'appeler soit la fonction *readResource()*, soit la fonction *deleteResource()* en fonction du type de la méthode (GET ou DELETE) de la requête http envoyée par le client ;
- faites précéder la requêtes SQL d'effacement du contenu dans la table " messages " d'un test permettant de vérifier que ce contenu existe bien dans la base de données *(utiliser la fonction readResource() pour réaliser ce test) ;*
- faites suivre l'effacement du contenu de la ressource par une récupération de la nouvelle collection de messages afin de la renvoyer au client ;
- testez le bon fonctionnement de l'agent serveur à l'aide du client RESTED, en lançant des requêtes GET et DELETE sur des ressources individuelles correspondant à des messages dont les groupes destinataires existent et n'existent pas dans la base de données ;
- faites une copie de sauvegarde et étendez les fonctionnalités du script de l'agent client écrit dans la partie 3 du tutoriel pour qu'il sache envoyer des requêtes de type DELETE ou des requêtes de type GET en fonction d'un choix saisi par l'utilisateur. .../...

.../...
modifiez également le script index.py pour qu'il indique que la méthode DELETE fait partie des méthodes acceptées par l'agent serveur.

- ajoutez au programme les structures de test permettant de traiter les exceptions correspondant aux opérations invalides ou interdites et de renvoyer, selon les cas, des compte-rendus d'erreur ou des compte-rendus d'opération explicites.

Note :
comme indiqué dans le chapitre 4 du tutoriel, le type de méthode de la requête http lancée par le client se récupère dans le dictionnaire des variables d'environnement :

```
httpMethod = os.environ['REQUEST_METHOD']
```

une instruction COMMIT doit confirmer la requête SQL d'effacement du contenu :

```
recipient = httpData['dest']
with connection.cursor() as cursor:
        sql = "DELETE FROM `messages` WHERE `dest` = '"+recipient+"'"
        print(sql)
        cursor.execute(sql)
connection.commit() # Suppression confirmée par un COMMIT (obligatoire)
```

Pour lancer une requête de type DELETE depuis le script de l'agent client il faut utiliser la méthode delete() de la libraire " requests " de la façon suivante :

```
customHeaders = {'Accept': 'application/json', 'Content-type' :
'application/x-www-form-urlencoded'}
httpReturn = requests.delete(uri, headers=customHeaders)
```

httpReturn *est l'objet contenant les données de la réponse http renvoyée par le serveur et* ***uri*** *est l'URI de la ressource dont le contenu doit être effacé*

Attention *: il est indispensable de* ***spécifier l'en-tête " Content-type*** *" avec la valeur " **application/x-www-form-urlencoded** " dans le cas du lancement d'une requête http utilisant la méthode DELETE pour que le module CGI reconnaisse correctement le format des données, sinon une erreur d'exécution sera renvoyée*

Exemples de contenus possibles pour la chaîne json retournée par le serveur dans le cas où une requête a produit une erreur ou dans le cas où l'opération s'est déroulée avec succès :

- *retour consécutif à une requête GET sur la collection intégrale des ressources*

```
{
        "code": "OPERATION_OK",
        "operation":"COLLECTION_READ",
        "text":"Collection read successfull",
        "content":[
                        {"dest":"men", "text":"Hello men !"},
                        {"dest":"women", "text":"Hello women !"},
                        {"dest":"girls", "text":"Hello girls !"}
                ]
}
```

.../...

.../...

- *retour consécutif à une requête GET sur une ressource individuelle qui n'existe pas (contenu inexistant dans la BdD)*

```
Le contenu renvoyé est celui de la collection existante. Il permet
de voir que la ressource recherchée n'existe pas.
```

```
{
    "code": "OPERATION_KO",
    "operation":"RESOURCE_READ",
    "text":"Resource not found",
    "content":[
                {"dest":"men", "text":"Hello men !"},
                {"dest":"women", "text":"Hello women !"},
                {"dest":"girls", "text":"Hello girls !"}
            ]
}
```

- *retour consécutif à une requête DELETE sur une ressource individuelle qui existait (on a par exemple effacé le message destiné aux filles)*

```
Le contenu renvoyé est celui de la collection après l'opération.
```

```
{
    "code": "OPERATION_OK",
    "operation":"RESOURCE_DELETE",
    "text":"Resource delete successfull",
    "content":[
                {"dest":"men", "text":"Hello men !"},
                {"dest":"women", "text":"Hello women !"}
             ]
}
```

- *retour consécutif à une requête DELETE pour effacer une ressource individuelle qui n'existe pas (on a par exemple essayé d'effacé le message destiné aux chats)*

```
{
    "code": "DELETE_KO",
    "operation":"RESOURCE_DELETE",
    "text":"Resource not found",
    "content":[
                {"dest":"men", "text":"Hello men !"},
                {"dest":"women", "text":"Hello women !"}
              ]
}
```

- *retour consécutif à une requête DELETE réalisée par un utilisateur non autorisé*

```
{
    "code":"USER_RIGHTS_KO",
    "operation":"RESOURCE_DELETE",
    "text":"User not allowed",
    "content":[]
}
```

- *retour consécutif à une requête POST pour laquelle le contenu de la ressource n'a pas été passé dans le corps de la requête*

```
{
    "code":"MISSING_CONTENT",
    "operation":"RESOURCE_CREATE"
    "text":"Resource content missing",
    "content":[
                {"dest":"men", "text":"Hello men !"},
                {"dest":"women", "text":"Hello women !"}
              ]
}
```

- *etc,…*

Vous pouvez choisir un autre formalise en veillant toutefois à ce que la structure et la terminologie des retours reste cohérente sur l'ensemble des cas.

Mise en place d'une procédure d'authentification des utilisateurs et de contrôle des droits d'accès aux ressources.

A ce stade, le Web Service expose publiquement des ressources dont le contenu peut être aussi bien consommé qu'effacé par n'importe quel client connaissant la procédure d'accès au script " messages.py " de l'agent serveur. Nous allons voir comment mettre en place une procédure de contrôle permettant de limiter les accès aux seuls utilisateurs identifiés et présents dans la table " users " de la base de données. Lorsque vous listez le contenu de la table " users " en utilisant phpmyadmin, vous constatez qu'elle contient 2 utilisateurs dont les login sont ***Joan*** et ***Mary*** disposant respectivement des droits ***read*** et ***all*** indiquant que Joan dispose seulement du droit de récupérer (lire) le contenu des ressources se trouvant dans la table " messages " et que Mary dispose de son côté de tous les droits (créer, lire, mettre à jour et effacer).

idUser	loginUser	pwdUser	rights
1	Joan	*C4B90000B75BB7AAB97655238B97325CDB66F96A	read
2	Mary	*3A23E1093E954158272BD54EFCE2E8EC01356EB0	all

Vous noterez que la valeur du contenu de l'attribut " pwdUser " n'est pas égale à ***la valeur du mot de passe en clair qui est Andromède432 pour Joan et Passiflore812 pour Mary***. En effet, pour des raisons de sécurité, les mots de passe sont toujours enregistrés sous la forme d'un ***hashcode*** calculé en temps réel au moment de l'enregistrement dans la base. Dans le cas présent, le calcul de hashcode a été fait à l'aide de la fonction ***PASSWORD()*** interne au SGBD et il faudra utiliser cette fonction lors de l'implémentation de la requête SQL permettant de vérifier la validité du mot de passe des utilisateurs. Les requêtes ci-dessous permettent de comprendre comment la fonction PASSWORD() se met en oeuvre. Vous pouvez tester leur résultat en les exécutant dans la console SQL de phpmyadmin :

- ***SELECT PASSWORD('Andromède432')***
- ***SELECT PASSWORD('Passiflore812')***

Afin de réserver l'accès aux ressources aux seuls utilisateurs présents dans la table " users ", vous allez ajouter une procédure d'authentification qui consiste à faire vérifier, par le script " messages.py " de l'agent serveur, la validité des identifiants et des droits associés.

L'envoi des identifiants de l'utilisateur est fait via les en-têtes de la requête http lancée par le client. Pour cela vous devez renseigner le contenu d'une directive d'en-tête spécifique nommé " X-Auth " qui recevra le contenu d'une chaîne de caractères composée de la concaténation du login et du mot de passe de l'utilisateur séparés par le caractère **:** (deux points) :

X-Auth reçoit la chaîne login:password

A la réception de la requête http, le script " messages.py " doit :

- ✔ vérifier si l'en-tête X-Auth est bien présent en testant l'existence de l'élément ***os.environ('HTTP_X_AUTH')***
  - ➔ si c'est le cas en extraire le login et le mot de passe de l'utilisateur
  - ➔ sinon renvoyer un compte-rendu d'erreur au client et quitter

- ✔ exécuter une fonction qui contrôle la validité des identifiants (le login correspond-il à un des utilisateurs déclarés dans la base et le mot de passe est-il valide ?) et retourne les droits d'accès aux ressources de l'utilisateur. Cette fonction doit :

  - ➔ vérifier si les identifiants correspondent bien à ceux d'un utilisateur présent dans la table " users "

    - si c'est le cas récupérer ses droits d'accès sur les contenus de la table " messages " et les retourner au programme principal

    - sinon retourner au programme principal une information indiquant que l'utilisateur n'est pas autorisé à accéder au contenu de la ressource

- ✔ évaluer le retour de la fonction de vérification

  - ➔ si l'utilisateur n'existe pas ou si les droits d'accès ne permettent pas l'exécution de l'opération demandée, retourner un compte-rendu au client indiquant que l'opération n'a pas été autorisée, puis quitter

  - ➔ sinon lancer la lecture ou l'effacement du contenu de la ressource correspondant à la méthode GET ou DELETE de la requête http lancée par le client, puis retourner au client la réponse accompagnée du compte-rendu d'opération

★★ Travail à faire :

- faites une copie de sauvegarde de la version précédente du script " messages.py "

- rajoutez au script " messages.py " une procédure qui vérifie la présence puis récupère le login et le mot de passe de l'utilisateur utilisateur dans l'en-tête d'authentification

- créez une fonction testant l'existence, dans la table " users ", de l'utilisateur dont les identifiants ont été envoyés et récupérant ses droits d'accès si celui-ci a été trouvé *(cette fonction doit retourner, soit les droits d'accès, soit une indication signifiant que l'utilisateur n'a pas été trouvé)*

- créez la structure de test permettant d'appeler la fonction de lecture ou d'effacement du contenu de la ressource en fonction de la méthode utilisée dans la requête http lancée par le client (GET ou DELETE) ou retournant le compte-rendu d'exécution " opération non autorisée " en envisageant tous les cas possibles

  Nota : il est nécessaire d'étendre la structure de test pour prendre en compte le cas où le client aurait lancé une requête http de type DELETE sans avoir précisé le nom de la ressource individuelle en fin d'URI puisqu'il a été dit précédemment qu'il ne devait pas être possible d'effacer l'intégralité de la collection en une seule opération. Le tableau fourni en annexe vous guidera dans le travail de recensement et d'identification des cas d'erreurs que vous devez envisager.

- testez le fonctionnement du script serveur " messages.py " à l'aide du client RESTED

- complétez le script du client Python pour qu'il :
  - demande à l'utilisateur quelle opération il souhaite réaliser sur le contenu de la ressource (lire ou effacer) afin de choisir si il faut envoyer une requête http de type GET ou DELETE ;
  - demande à l'utilisateur de saisir son login et son mot de passe afin d'envoyer, dans la requête http, un en-tête nommé " X-Auth " contenant la chaîne de caractères constituée de la concaténation du login et du mot de passe selon le format spécifié précédemment
- testez le fonctionnement de l'ensemble ***client python/serveur python***

Note :

pour récupérer l'en-tête d'authentification dans le script " messages.py " :

```
loginMdp = os.environ['HTTP_X_AUTH'].split(':') # loginMdp : liste contenant le login et
                                                # le mot de passe de l'utilisateur
                                                 # respecter scrupuleusement le nom
                                                 # " HTTP_X_AUTH "
```

pour vérifier l'existence d'un utilisateur dans la table " users " et récupérer les droits de l'utilisateur, lancer la requête :

```
with databaseConnection.cursor() as cursor:
               sql = "SELECT `rights` FROM `users`
               WHERE `loginUser` = '" + loginMdp[0] + "' AND `pwdUser` = PASSWORD('" +
               loginMdp[1] + "')"
               cursor.execute(sql)
               rights = cursor.fetchone()
```

*si le couple des identifiants de l'utilisateur a été trouvé dans la table " users ", le contenu de " rights " est un dictionnaire de la forme {'rights': 'all'} sinon le contenu de " rights " est non défini (sa valeur est égale à **None**)*

pour insérer la directive " X-Auth " dans l'en-tête de la requête http lancée par le script client :

```
loginPwd = 'Mary:Passiflore812'
customHeaders = {'Accept' : 'application/json', 'X-Auth' : loginPwd, 'Content-type' :
'application/x-www-form-urlencoded'}
httpReturn = requests.get(uri, headers=customHeaders)  # lire le contenu d'une ressource
# ou
httpReturn = requests.delete(uri, headers=customHeaders) # effacer le contenu d'une ressource
```

7 - Mise en œuvre des requêtes POST et PUT pour la création et la mise à jour du contenu des ressources.

Dans cette dernière partie du tutoriel, nous allons voir comment créer ou mettre à jour le contenu d'une ressource. Lorsque l'on met en œuvre les deux méthodes POST et PUT, il faut non seulement préciser l'identifiant de la ressource à la fin de l'URI, comme on l' a fait jusque là lors de l'utilisation des méthodes GET et DELETE, mais il faut aussi envoyer le contenu (ici le texte du message) qui doit être enregistré sur l'hôte d'hébergement. Si l'on veut par exemple créer un nouveau message destiné aux éléphants dans la base de données il faut fournir à la fois le nom du groupe destinataire

" elephants " et le texte du message " Hello elephants ! ". Le client doit donc transmettre non plus 1 mais 2 informations via la requête http.

Puisqu'on peut rajouter des paramètres supplémentaires en fin d'URI, il serait possible de lancer des requêtes POST et PUT en utilisant des expressions de la forme :

https://www.gaalactic.fr/ws/~login/ws/messages/**elephants?text=Hello elephants !**

*Dans une telle expression, le terme **elephants** identifierait la ressource à créer ou à mettre à jour et correspondrait à l'attribut " dest " de la table " messages ", le paramètre **text** portant quant-à lui le contenu de la ressource correspondant à l'attribut " text " de la table " messages ".*

Mais on utilise préférentiellement une autre façon de procéder avec ces deux méthodes, qui consiste à faire passer le contenu de la ressource non pas via un paramètre d'URI mais en plaçant les données dans le corps de la requête http. Ce mode opératoire présente un gros avantage car la taille du corps de la requête http n'est pas limitée (contrairement à la zone des arguments d'URL), ce qui permet d'envoyer des contenus de toutes tailles et de tous types (textes, binaires, sons, images, etc.), à condition d'en avoir indiqué la nature avec la directive d'en-tête "***Content-type*** ".

A retenir : *on réservera plutôt l'utilisation des paramètres d'URI à la formulation de filtres en conjonction avec l'utilisation d'une requête de type GET, à l'image de l'exemple donné en cours, qui illustre la façon dont on pourrait récupérer, via un Web Service, la liste de tous les antibiotiques de la classe macrolides disposant d'une forme pharmaceutique de type sirop (cf. ci-desous) :*

*https://www.pharmacy.com/api/v1/medicaments/antibiotics/macrolids**?pharmaceutical_form=sirup***

Une fois présenté le principe général de mise en œuvre des deux méthodes POST et PUT, voyons comment on insère le contenu de la ressource dans le corps de la requête côté client et comment on on le récupère côté serveur.

★★ Travail à faire :

Dans un premier temps vous allez visualiser la façon dont les données sont récupérées côté serveur lorsque l'on insère à la fois le nom identifiant la ressource en fin d'URI et son contenu dans le corps de la requête lors de l'utilisation des méthodes POST et PUT :

- faites une copie de sauvegarde du fichier " messages.py " et copiez-y le script ci-après (page 35) puis déposez le sur le serveur ;
- ouvrez l'extension RESTED de votre navigateur puis lancez des requêtes utilisant les quatre méthodes GET, DELETE, POST et PUT pour lesquelles sont fournis ou pas le nom d'une ressource (men, women,…), et son contenu (Hello world !, Hello women,…), le premier étant précisé dans l'URI, le second à l'aide d'une donnée http (nommé paramètre dans RESTED) dont le nom est " text " (qui correspondre à l'attribut " text " de la table " messages " ;
- *(n'oubliez pas de fournir l'en-tête **Content-type** valant **application/x-www-form-urlencoded** sans quoi le script serveur messages.py ne saura pas interpréter le format des données reçues et renverra une erreur d'exécution*).

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import os
import json
import cgi
import cgitb
cgitb.enable()
print('Content-type: application/json\n')

def returnHttpData():
    formData = cgi.FieldStorage()
    httpData = {}
    httpDataKeys = []
    httpDataKeys = list(formData)
    for key in httpDataKeys:
        httpData[key] = (formData[key].value)
    return httpData

httpMethod = os.environ['REQUEST_METHOD']
httpData = returnHttpData()
dataset = {'method' : httpMethod, 'data' : httpData}
print(json.dumps(dataset))
```

Nota *: le but de ce travail préliminaire, au-delà de la possibilité de visualiser les données reçues par l'agent serveur, est de vous faire interroger sur la validité des différentes opérations que peut demander le client en fonction de la méthode http utilisée et selon que les données caractérisant les ressources (identifiant " dest " via l'URI et contenu " text "dans le corps de la requête) sont envoyées ou pas. Le tableau fourni en annexe vous aidera à identifier de façon exhaustive l'ensemble des cas possibles.*

Lorsque l'on développe le programme d'un agent serveur, il est nécessaire de prévoir les structures de contrôle qui permettront de rejeter toutes les requêtes qui son susceptibles de produire des opérations invalides ou interdites, afin d'éliminer les retours de messages abscons et les risques de corruption de la base de données. Vous n'aurez peut-être pas le temps de les implémenter toutes dans le cadre de ce tutoriel. En effet, le tableau montre que les cas pour lesquels l'envoi d'une requête http correspond à une opération invalide ou interdite sont très largement majoritaires (11 cas sur 16).

Il est important de retenir que la conception et l'implémentation d'un agent serveur acceptant l'ensemble des méthodes http doit se faire de façon très rigoureuse et inclure les traitements et les tests nécessaires à la levée de ces exceptions.

Implémentation des requêtes http de type POST.

On suppose que l'on veut ajouter une nouvelle ressource constituée d'un message destiné aux éléphants. Il faut donc lancer une requête POST qui va être traduite par le script serveur messages.py dans la requête SQL suivante :

INSERT INTO `messages` VALUES ('elephants', 'Hello elephants !')

Mais avant de pouvoir lancer la demande de création au SGBD il faut au préalable s'assurer que le contenu correspondant à la ressource à créer n'existe pas déjà dans la base de données. Toute requête SQL de type INSERT doit donc être précédée d'une requête de type SELECT qui doit permettre de vérifier qu'aucune ressource du même nom n'est déjà présente. Dans le cas présent, la recherche va être faite sur l'attribut " dest " de la table message, puisque c'est cet attribut qui constitue l'identifiant de la ressource :

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

Si et seulement si le retour de cette requête est vide, il est alors possible de lancer la requête de création. Sinon, il faudra renvoyer un message au client pour l'informer que la ressource n'a pas pu être créée car son contenu est déjà présent dans la base de données.

Du côté du script de l'agent client, il est nécessaire d'ajouter un bloc de programme permettant de lancer des requêtes de type POST en utilisant une nouvelle méthode de la librairie *requests* comme indiqué ci-dessous :

httpReturn = requests.post(uri, httpHeaders, httpData)

Des précisions sur l'implémentation de la méthode requests.post() sont fournies dans la note de la page 36.

★★ Travail à faire :

- reprenez la copie du script " messages.py " contenant l'ensemble de votre script serveur développé jusqu'à la partie 6 ;
- à partir des modèles déjà élaborés pour les fonctions *readResource()* et *deleteResource()* rajoutez une fonction de création *createResource()* dans le script principal ou dans la librairie qui lance une requête de type INSERT pour créer le contenu de la ressource dans la table " messages " ;
- ajoutez dans le programme principal un test pour empêcher de lancer l'exécution de la fonction *createResource()* si un nom de ressource individuelle n'a pas été précisé dans l'URI et/ou si un contenu n'a pas été envoyé via le corps de la requête http par le client ;
- ajoutez dans le programme principal une structure de test permettant d'appeler soit la fonction *readResource()*, soit la fonction *deleteResource()*, soit la fonction *createResource()* en fonction du type de la méthode (GET, DELETE ou POST) de la requête http envoyée par le client ; complétez la structure de test pour déterminer si l'opération de création est autorisée ou pas en fonction des identifiants utilisateurs qui ont été envoyés dans l'en-tête d'authentification ;
- faites précéder la requêtes SQL de création du contenu dans la table " messages " d'un test permettant de vérifier que ce contenu n'existe pas déjà dans la base de données *(utiliser la fonction readResource() pour réaliser ce test) ;*
- faites suivre la création du contenu de la ressource par une récupération de la nouvelle collection de messages afin de la renvoyer au client ;
- ouvrez l'extension RESTED de votre navigateur puis lancez comme précédemment plusieurs requêtes de type POST pour lesquelles d'une ressource individuelle et son contenu sont fournis ou pas, afin de tester le comportement du script dans les différents cas, et améliorez si nécessaire votre structure de test ;
- faites une copie de sauvegarde et étendez les fonctionnalités du script de l'agent client écrit dans la partie 6 du tutoriel pour qu'il sache envoyer des requêtes de type POST, DELETE ou GET en fonction de choix saisis par l'utilisateur (type de requête à lancer, valeurs à saisir pour les noms et les contenus des ressources).

Note :

De la même manière que vous l'avez fait pour la requête SQL DELETE, vous devez faire suivre la requête CREATE d'un instruction COMMIT

Pour mettre en œuvre la méthode request.post() côté client :

```
loginPwd = 'Mary:Passiflore812'
uri = 'https://www.gaalactic.fr/~login/ws/messages/elephants'
customHeaders = {'Accept': 'application/json', 'X-Auth': loginPwd, 'Content-type' :
'application/x-www-form-urlencoded'}
httpReturn = requests.get(uri, headers=customHeaders)  # Lecture
# or
httpReturn = requests.delete(uri, headers=customHeaders) # Suppression
# or
httpData = {'text': 'Hello elephant !'}
httpReturn = requests.post(uri, headers=customHeaders, data=httpData) # Création
```

Implémentation des requêtes http de type PUT.

On suppose que l'on veut mettre à jour la ressource contenant le message destiné aux éléphants avec un nouveau contenu de message valant ***"Hello big elephants !"***. Pour cela il faut lancer une requête http PUT qui va être traduite par le script serveur messages.py dans une requête SQL qui a la forme suivante :

UPDATE `messages` SET `text` = 'Hello big elephants !' WHERE `dest` = 'elephants'

De la même manière que dans le cas précédent (requête http POST) il est nécessaire, avant de lancer la requête SQL de mise à jour de la ressource, de procéder préalablement à l'interrogation de la base de données, mais pour cette fois-ci s'assurer que la ressource et son contenu sont bien présents. Cette vérification sera à nouveau effectuée par la requête SQL :

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

à la différence près que la mise à jour ne sera lancée que si et seulement si la requête renvoie un résultat non vide.

★★ Travail à faire :

- procédez de la même façon que dans le cas de la méthode POST. Le travail consiste à étendre le script serveur messages.py en ajoutant une fonction *updateResource()* qui lance une requête de mise à jour du contenu de la table messages à partir des données récupérées dans la requête http et en complétant la structure de tests qui doit contrôler le respect des conditions d'exécution (tests par rapport aux données reçues et aux droits de l'utilisateur sur la table " messages ") ;

- procédez au test et au debogage du script serveur à l'aide du client RESTED puis étendez le script client Python pour qu'il puisse lancer des requêtes de type PUT à l'aide de la méthode *requests.put()* (cf. ci-dessous)

```
loginPwd = 'Mary:Passiflore812'
 uri = 'https://www.gaalactic.fr/~login/ws/messages/elephants'
 customHeaders = {'Accept': 'application/json', 'X-Auth': loginPwd, 'Content-type' :
'application/x-www-form-urlencoded'}
 httpData = {'text': 'Hello big elephants !'}
 httpReturn = requests.put(uri, headers=customHeaders, data=httpData) # Mise à jour
```

Annexe

Tableau d'évaluation de la validité des requêtes http

Le tableau ci-après présente une synthèse des différentes possibilités de formulation d'une requête http par un client de Web Service. Chaque requête est évaluée du point de vue de sa consistance et de la validité de l'opération demandée à l'agent serveur, en fonction de la méthode http et selon les données qui sont envoyées par le client via l'URI ou le corps de la requête. Tous les cas qui se traduisent par une une opération invalide correspondent à des exceptions susceptibles de provoquer une erreur d'exécution ou un comportement inattendu ou non souhaité du côté serveur, qui pourrait aboutir à une corruption de la base de données ou à la divulgation de contenus confidentiels. Le script de l'agent serveur devra donc intégrer des structures de test suffisantes pour identifier et traiter l'ensemble des cas possibles et refuser l'exécution des opérations invalides. Il devra également renvoyer au client une information documentant l'erreur constatée à l'aide d'un code d'erreur accompagné d'un texte explicatif suffisamment explicite.

Méthode http	Nom de ressource précisé dans l'URL	Contenu de ressource envoyé dans le corps de la requête	Signification de l'opération demandée	Statut de l'opération	Opération SQL correspondante
GET	Non	Non	Récupérer la représentation de la collection complète	**Valide**	SELECT
	Non	***Oui***	***Inconsistante***	***Invalide***	
	Oui	Non	Récupérer la représentation d'une ressource individuelle	**Valide**	SELECT + restriction (WHERE)
	Oui	***Oui***	***Inconstante***	***Invalide***	
DELETE	***Non***	Non	Effacer le contenu de l'intégralité de la collection	***Interdite***	
	Non	***Oui***	***Inconsistante***	***Interdite***	
	Oui	Non	Effacer le contenu d'une ressource individuelle	**Valide**	DELETE + restriction (WHERE)
	Oui	***Oui***	***Inconsistante***	***Invalide***	
POST	***Non***	***Non***	***Inconsistante***	***Invalide***	
	Non	Oui	***Inconsistante***	***Interdite***	
	Oui	***Non***	Créer une ressource individuelle sans contenu	***Invalide***	
	Oui	Oui	Créer une ressource individuelle et son contenu	**Valide**	INSERT
PUT	Non	***Non***	***Inconsistante***	***Invalide***	
	Non	Oui	Modifier le contenu de la collection complète	***Interdite***	
	Oui	***Non***	***Inconsistante***	***Invalide***	
	Oui	Oui	Mettre à jour le contenu d'une ressource individuelle	**Valide**	UPDATE + restriction (WHERE)

Nota Bene *: ce tableau montre que les cas qui correspondent à une opération invalide sont très largement majoritaires (11 cas sur 16). Vous n'aurez peut-être pas le temps de mettre en place tous les tests qui seraient nécessaires dans le cadre du tutoriel. Cependant, vous devez retenir que cette partie de la conception et de l'implémentation d'un Web Service est indispensable et qu'elle nécessite la plus grande attention ainsi qu'un temps d'analyse, de développement et de test important.*