Cepty Consultant - Ingénierie linguistique

Projet : API de gestion de base de données JSON

MANUEL D'UTILISATION

I/ Présentation de l'API

Dans le cadre de sa mission de conseil, Cepty Consultant propose une application Back-End hébergée sur un nouveau serveur, afin de manipuler la base de données de son client dans le cadre de projet d'ingénierie linguistique.

Pour cela, une REST API a été mise en place grâce au module python Flask, qui a permis de développer une application efficace et fonctionnelle reliée à la base de données JSON et d'effectuer diverses actions sur ces dernières au moyen de requêtes du protocole HTTP. Elles sont facilement faites au moyen du logiciel Postman, qui fait ici fonction de client.

La partie serveur est contrôlée par deux systèmes: Gunicorn (serveur HTTP WSGI) sert d'intermédiaire entre l'application et le serveur Nginx en s'assurant du bon traitement des requêtes, tandis que Nginx assume la fonction de serveur HTTP et de proxy inverse, s'occupant entre autres de la répartition de charge, de la gestion du cache et du chiffrement SSL. Supervisor est également utilisé, pour opérer un contrôle et un suivi des processus applicatifs sur les systèmes de type UNIX.

II/ Pré-requis et lancement de l'API

a) Arborescence de l'application

Vous trouverez divers fichiers et exécutables dans le livrable que vous aurez reçu. En dehors des dossiers se trouvent ce manuel (`Manuel.pdf`), la liste des modules de l'environnement virtuel utilisé (`requirementsAPI.txt`) et un fichier contenant la liste des commandes à exécuter pour configurer et exécuter les différents acteurs contribuant au bon fonctionnement de l'application (`commandes.sh`).

Le dossier "Data" contient les données du projet linguistique et la liste des cinq contributeurs autorisés à utiliser l'application: respectivement `DONNEES_CLIENT.json` et `LISTE_COLLABORATEURS.json`

Le dossier "Configuration" contient les fichiers nécessaires au bon fonctionnement du serveur, qui règlent la configuration de chacun des composants du Back-end:

- le fichier conf pour l'exécution de l'application avec supervisor (`/home/`*`user`*`/environnements/virtenvTW/lib/python3.7/site-packages/supervisor/conf.d/`**`api_cepty.conf`**),
- Supervisor (`supervisord.conf`) ,
- le serveur NGINX (`/etc/nginx/sites-available/`**`nginx_api_cepty.conf`**),
- et les paramètres de sécurité du serveur (`/etc/nginx/snippets/`**`secu-params.conf`**)

Le dossier "Scripts" contient le script à exécuter pour lancer l'application (`requests_api_run.py`), un script de traitement des données JSON (`wrangling_json_data.py`).

Le dossier “environnement” contient les fichiers qui paramètrent l’environnement virtuel de développement que vous créez en suivant les instructions de `commandes.sh`.

b) Lancer l’application

Nous vous proposons, dans un souci de rapidité et de simplicité, de suivre les étapes détaillées dans le fichier `commandes.sh`. Cela vous permettra d’installer les logiciels et modules nécessaires à l’exécution de l’application, et s’occupera de la configuration des serveurs. Attention! il est nécessaire de modifier les chemins par rapport à l’emplacement de l’application dans votre machine! En cas de besoin vous pouvez consulter le fichier `requirementsAPI.txt` pour compléter votre environnement de développement.

Ensuite, exécutez le script python `requests_api_run.py` en le précédant du chemin relatif au dossier dans lequel vous vous trouverez. L’application est lancée! Vous pouvez à présent utiliser Postman pour manipuler les données.

Vous devriez voir la sortie ci-dessous dans votre terminal:

```
 * Serving Flask app "requests_api_run" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:5000 (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 712-472-159
```

III/ Requêtes HTTP

a) Utiliser Postman

Il existe beaucoup de solutions pour appeler/tester une API web: Postman fait partie de ces solutions. Le logiciel Postman permet de construire et d’exécuter des requêtes HTTP, de tester des API et aussi de les enregistrer.

Pour l’utiliser, il faut entrer l’URL de l’API dans le cadre situé au milieu de l’interface, et cliquer sur “Send”. Si un processus d’authentification est nécessaire, le message “Unauthorized access” s’affiche dans le cadre en bas de l’interface. Il faut alors se rendre dans l’onglet “Authorization”, sélectionner “Basic Auth” sur le menu déroulant qui apparaît alors, entrer les identifiants dans les cadres prévus à cet effet et cliquer à nouveau sur “Send”. L’API est donc disponible, et vous pouvez à présent écrire vos requêtes, dont la syntaxe est expliquée dans la partie ci-dessous.

Choisissez dans le menu déroulant à gauche de l’URL le type de requête que vous souhaitez effectuer. Pour modifier le corps et le format de votre requête, il suffit de cliquer sur l’onglet “Body”, puis le bouton radio “raw” et enfin “JSON” à droite.

b) Syntaxe et sémantique des requêtes

Lire les données : requête GET

→ Modifier l’URL pour accéder aux données qui vous intéressent :

-pour un numéro de contribution x (index commençant à 0) :
ceptyconsultant.local/get/x
-pour un champ donné (ex : contrib_name) :
ceptyconsultant.local/get/contrib_name
-pour un champ donné dans une contribution x :
ceptyconsultant.local/get/x/contrib_name

Exemple :
Si je veux lire la valeur du champ validate dans la contribution 2 (donc la troisième), je vais écrire l'URL *ceptyconsultant.local/get/2/validate*

Ajouter de nouvelles données : requête POST

→ Vérifier que les données à ajouter ont bien le format requis et comportent les champs requis
→ Écrire le corps de la requête en format JSON

Exemple :
Si vous écrivez la requête ci-dessous, ces données seront ajoutées à la fin du document JSON, dans la clé « Data ». :


```
{
    "public_id": "ab317597-7c66-4af6-9037-ffa862c9178a",
    "dico_id": "yb_fr_3031",
    "user_id": "b42e96a8-7b0b-8b45-ae69-7c2efd472e1d",
    "user_name": "Bergier",
    "article_id": "6777b70e-02d1-497f-ab63-433f2205978f",
    "contrib_type": "sound",
    "contrib_data": "amm-_sound_2-2019-09-23T105941.149Z-.wav",
    "contrib_path": "https://ntealan.net/soundcontrib/",
    "contrib_name": "amm\u025b",
    "ntealan": true,
    "validate": false,
    "last_update": "2019-09-23 10:59:43.135000"
}
```


Modifier des données existantes : requête PUT

→ Cette requête vous permet de modifier des données :
- un champ dans une contribution
- un champ dans toutes les contributions

→ La requête doit être en format JSON

Exemple :
La requête ci-dessous écrit « sound » comme valeur dans la clé/champ « contrib_type » dans toutes les contributions (car data_number = None)
→ Si data_number = x, modifie le champ dans la contribution numéro x

```
{
        "field": "contrib_type"
                "data_number": None,
        "new_data": "sound"
}
```

Supprimer des données existantes : requête DELETE

→ Cette requête vous permet de supprimer **une contribution** du fichier de données.
→ La requête doit être en format JSON
→ On ne peut pas supprimer un seul champ d'une contribution, puisque cela pourrait altérer la validité du formatage des données. Cependant, vous pouvez modifier la valeur d'un champ par Null ou Nan avec la requête PUT.

Exemple :
La requête ci-dessous supprime la dernière contribution inscrite dans la base de données JSON, avec tous ses champs.

```
{
                "data_number":  -1
}
```

c) Index des erreurs

- `400 bad request`: dans les deux cas présentés ci-dessous, le problème vient du formatage de la requête. Elle doit absolument être en format JSON et contenir tous les champs nécessaires (indiqués dans la sortie erreur).

```
{
 "message": "Request body must be JSON"
}
```

```
{
 "message": "Wrong data structure, check these fields exist in your request: ['field', 'data_number', 'new_data']"
}
```

- `405 Method not allowed`: consultez la partie III/ du manuel pour vous assurer que vous êtes sur le chemin (URL) correspondant à la requête souhaitée.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

```
<title>405 Method Not Allowed</title>
<h1>Method Not Allowed</h1>
<p>The method is not allowed for the requested URL.</p>
```

- `401 Unauthorized access`: problème dans l'authentification. Voir III/a) et IV/a) dans le manuel le protocole de sécurité utilisateur.

```
Unauthorized Access
```

IV / Questions de sécurité

a) L'authentification

Pour plus de sécurité dans notre application, une authentification est requise, que ce soit via un navigateur web (Mozilla, Chrome..) ou via un client (Postman). Cela sera visible lors de l'ouverture via le navigateur par l'apparition d'une fenêtre pop-up vous demandant vos identifiants. Sur Postman, il faudra aller sur l'onglet "Authorization", choisir l'option "Basic Auth" dans le menu déroulant "TYPE", écrire les identifiants dans les cases "Username" et "Password", et cliquer sur le bouton "Send" pour accéder aux données. Si l'identification n'est pas bonne, ou si vous ne rentrez pas les identifiants, le message "Unauthorized access" restera visible (cf.index des erreurs).

Les identifiants nécessaires sont constitués du nom des différents clients pour l'username, et du mot "password" pour le mot de passe.
Il y a donc 5 identifiants possibles: "Mikolov", "Bergier", "Taken", "Gate" et "Fokwe".

b) Sécurisation Nginx

Afin que le côté serveur soit sécurisé, nous avons désactivé les requêtes HTTP qui ne sont pas exposées dans le manuel. A travers la configuration de notre serveur, nous avons également pris soin de masquer la version de NGINX utilisée, d'empêcher le détournement de clic, le téléchargement furtif et le *cross-site scripting*.

c) Poursuite du projet

Pour perfectionner la sécurité de l'application, nous avons prévu de travailler sur l'usage de certificats SSL/TSL, ainsi que sur le cryptage des données des utilisateurs.