65, rue des Grands Moulins, 75013 PARIS

Création d'un back end permettant l'accès à des données avec une API REST

BOUZARD ARTHUR & AUFFRET FABIENNE

Institut National des Langues et Civilisations Orientales (I.NA.L.C.O)

Master 2 TAL IM- Février 2021

Sommaire

1. Création et tests en local

Nous avons tout d'abord développé notre application en local.

Nous avons utilisé Flask et SQLAlchemy pour créer une base de données sous PostgreSQL avec deux tables (Users et Data) qui seront remplies en lisant les fichiers users.json et les données FR.tsv.

Nous avons fait en sorte que le programme se connecte à cette base de données dont l'Uri est spécifié dans le fichier config.py (SQLALCHEMY_DATABASE_URI = 'postgresql+psycopg2: //tp :tp@localhost/geoname').

Le fichier api.py est ce qui servira de point d'entrée pour notre application. Cela indiquera à notre serveur Gunicorn comment interagir avec l'application. Nous y avons indiqué les « routes » des requêtes.

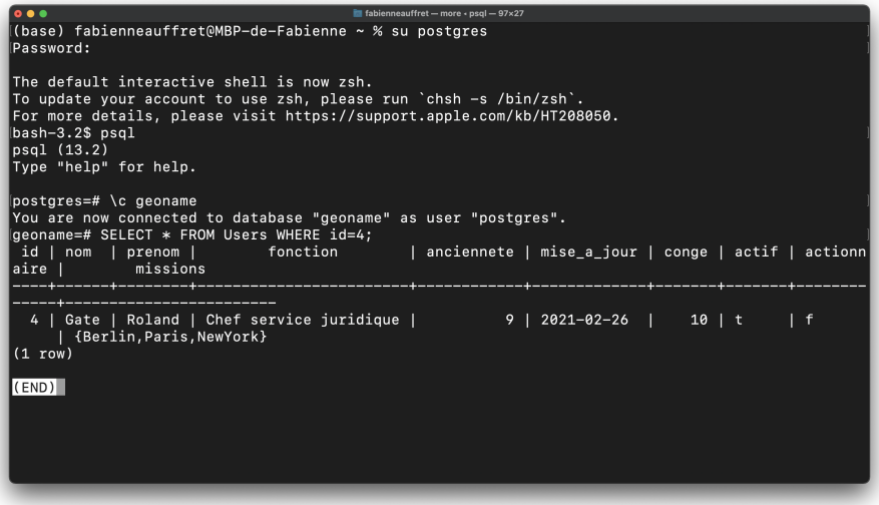
C'est le fichier models.py qui contient la structure des tables et les méthodes qui leur seront appliquées (lire, effacer, éditer, ajouter).

Le programme construit ensuite les tables users et data si elles n'existent pas dans la base de données puis les peuple avec les données fournies.

Nous avons ensuite écrit les fichiers DataService.py et UsersService.py (dans un répertoire services) qui traitent les requêtes des utilisateurs en utilisant les méthodes définies dans chaque classe Users ou Data.

Nous avons ajouté les méthodes peu à peu, les testant au fur et à mesure (en regardant d'abord dans la base en ligne de commande, puis avec Postman).

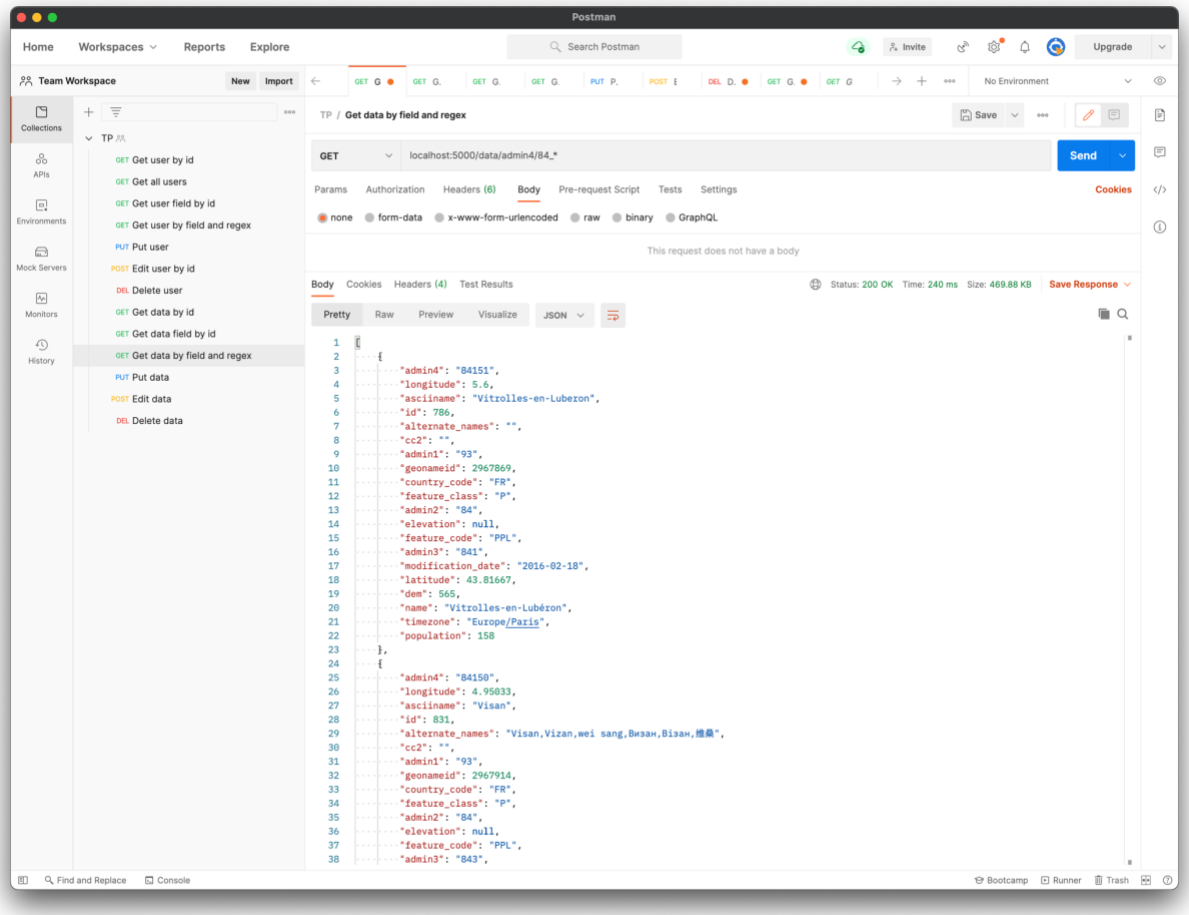
Nous avons fait des requêtes en local en ligne de commande (on se connecte avec le bon utilisateur, puis on lance Postgres – commande psql- puis on se connecte à la table\c nom_de_la_table et on peut faire des requêtes) pour vérifier que notre base de données avait été bien initialisée.



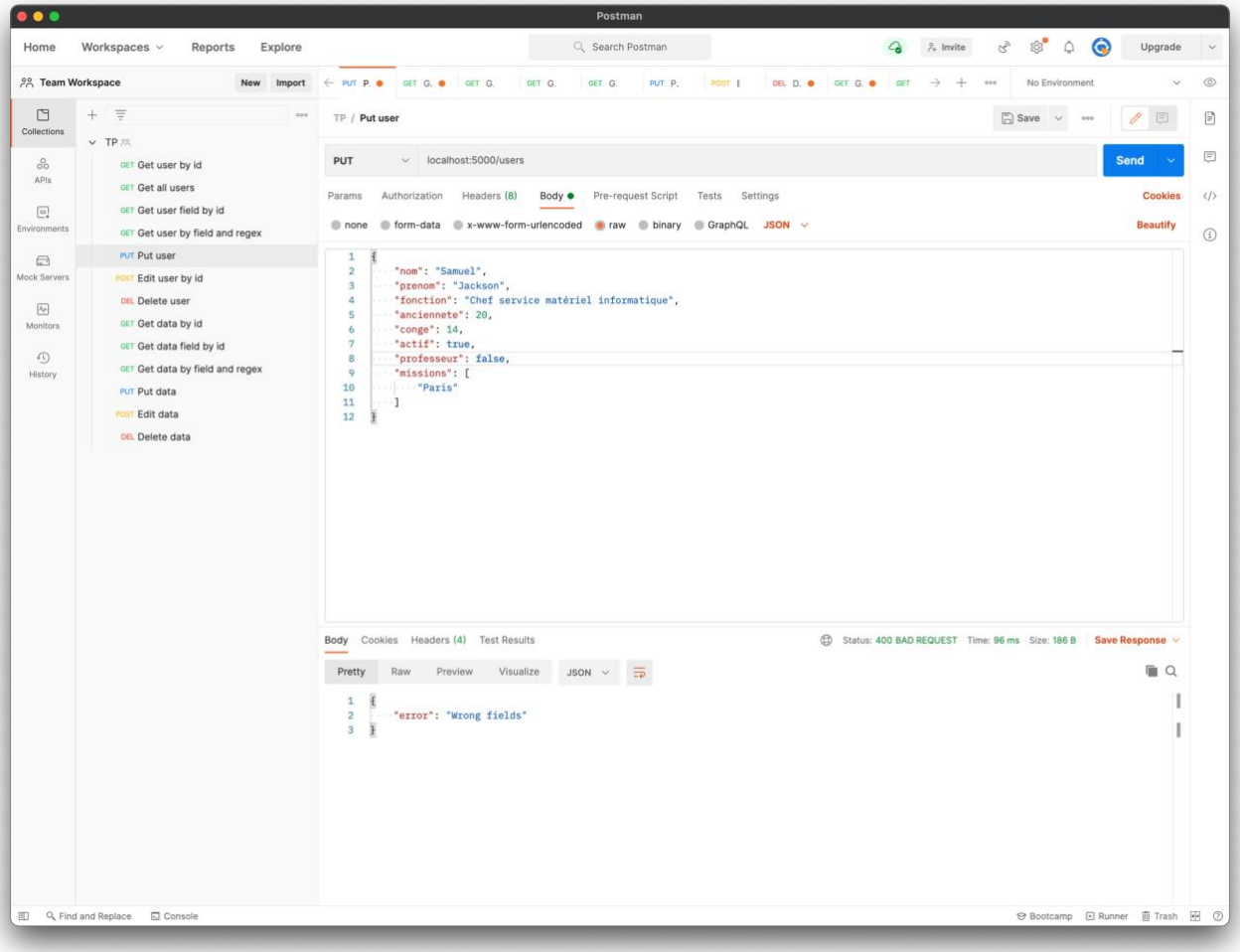
Nous avons traité divers cas d'erreurs (mauvais champ, données absentes, etc.) et renvoyé un message indiquant à l'utilisateur le statut de la requête traitée.)

Nous avons aussi proposé une requête avec regex (sachant que pour alchemy, les _ remplacent le point).

Nous avons testé toutes les sortes de requêtes et nos messages d'erreurs avec Postman en local :



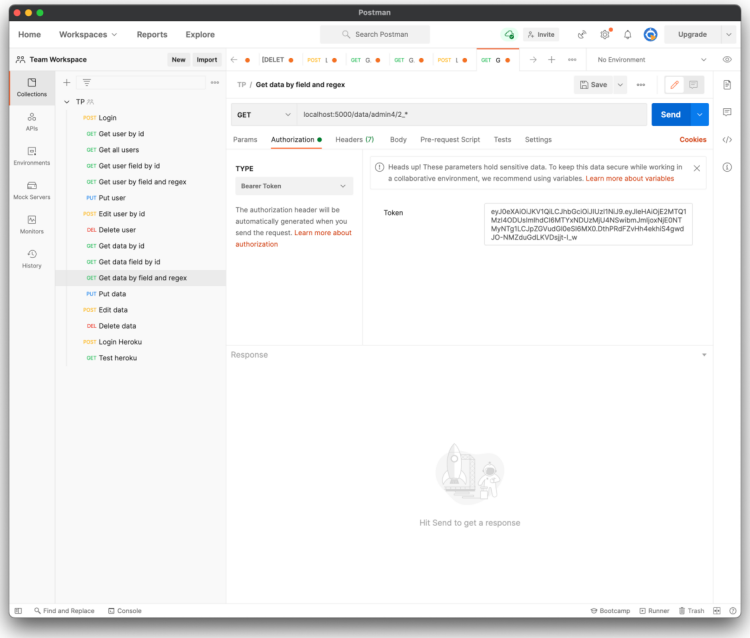
Et ici avec le champ « professeur » au lieu de « actionnaire » dans le body de la requête :

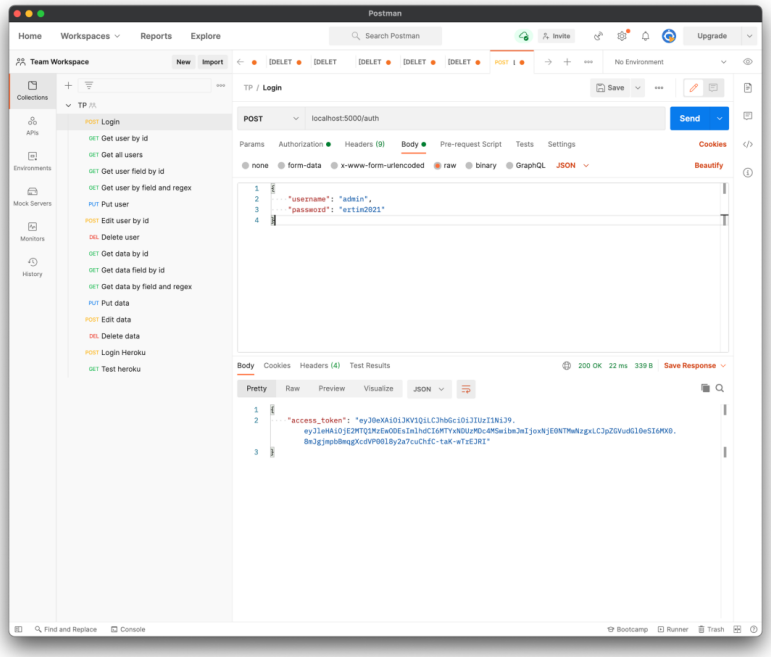


Nous avons ensuite implémenté Flask-JWT dans AuthService.py pour limiter les connexions (sans créer de nouveaux champs password et uuid dans la table user, ce qui avait été la première piste suivie, avec l'utilisation de tokens), en adaptant les choix de ce tutoriel : https://pythonhosted.org/Flask-JWT/.

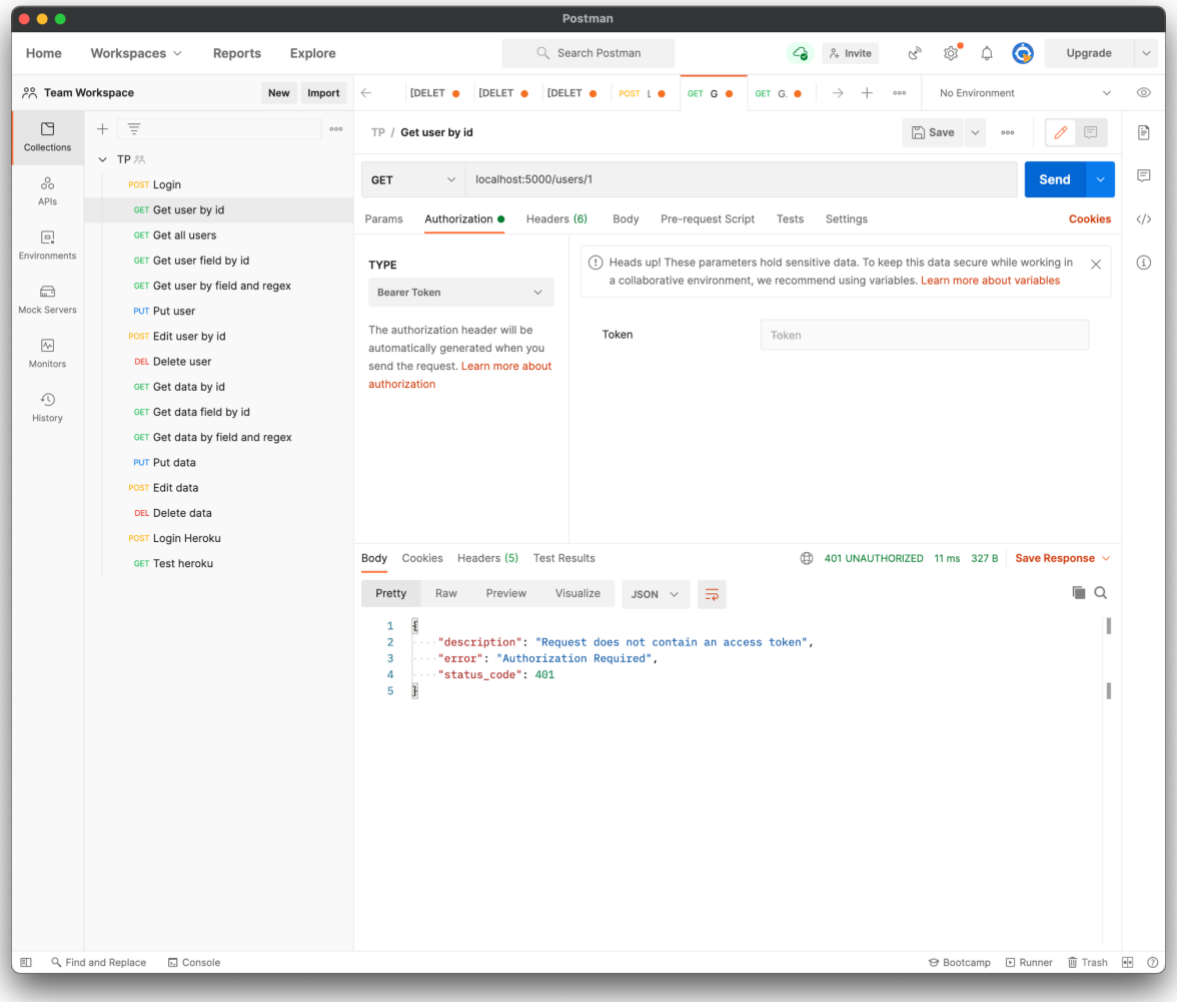
.

Désormais il faut s'identifier dans Postman pour se connecter à l'api. Il faut envoyer un body sur l'url du serveur /auth, contenant les champs username et password correctement renseignés pour recevoir un token qu'il faut coller dans l'onglet 'Authorization' de postman :

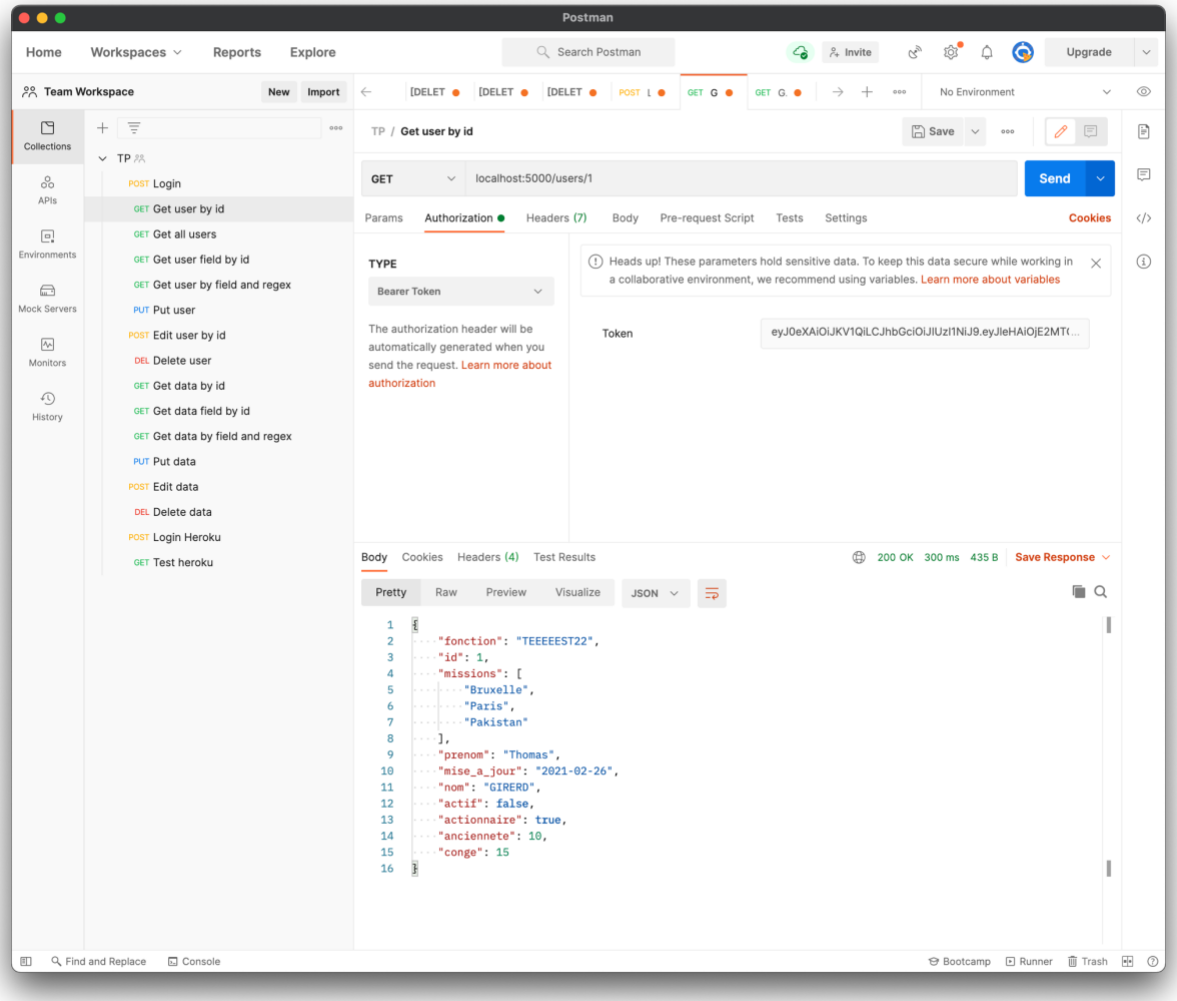




Nous récupérons ce token, et si nous ne copions pas le token dans l'entête de la requête nous avons un message d'erreur :



Avec le token on obtient l'affichage suivant :



2. Déploiement sur Heroku

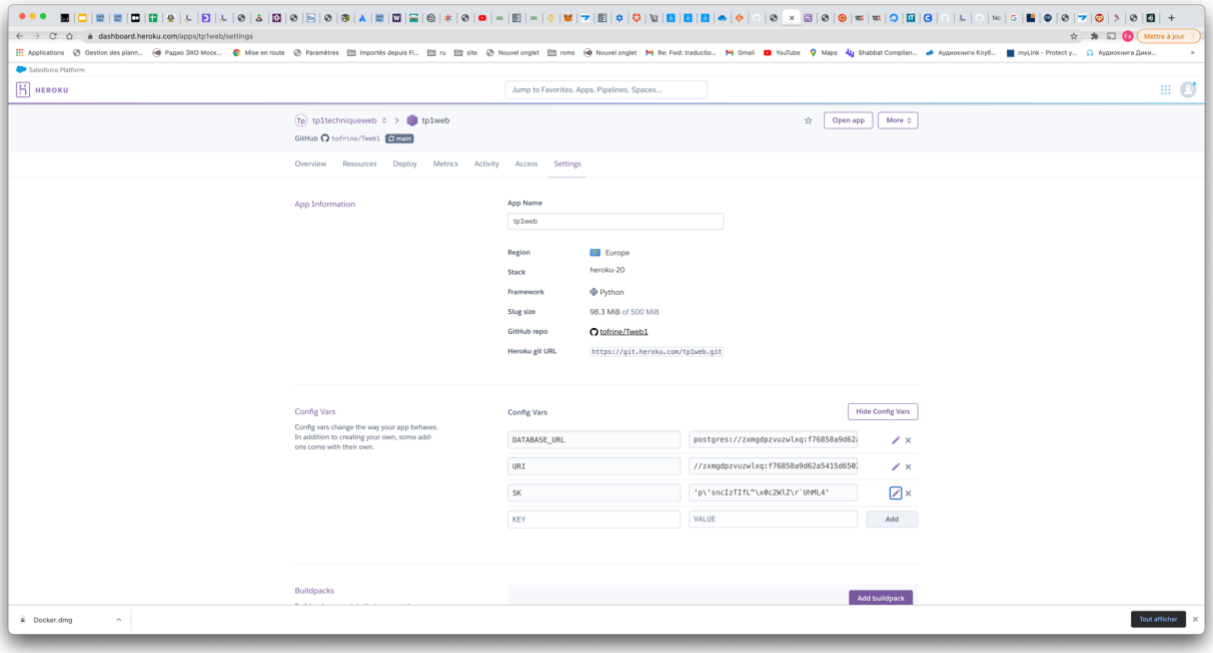
2.1 Préparation et upload

Nous avons créé un fichier requirements.txt listant modules python nécessaires à Heroku et avons généré un fichier pipenv avec la commande 'pipenv install –r requirements.txt' utilisé par Heroku pour installer les librairies python.

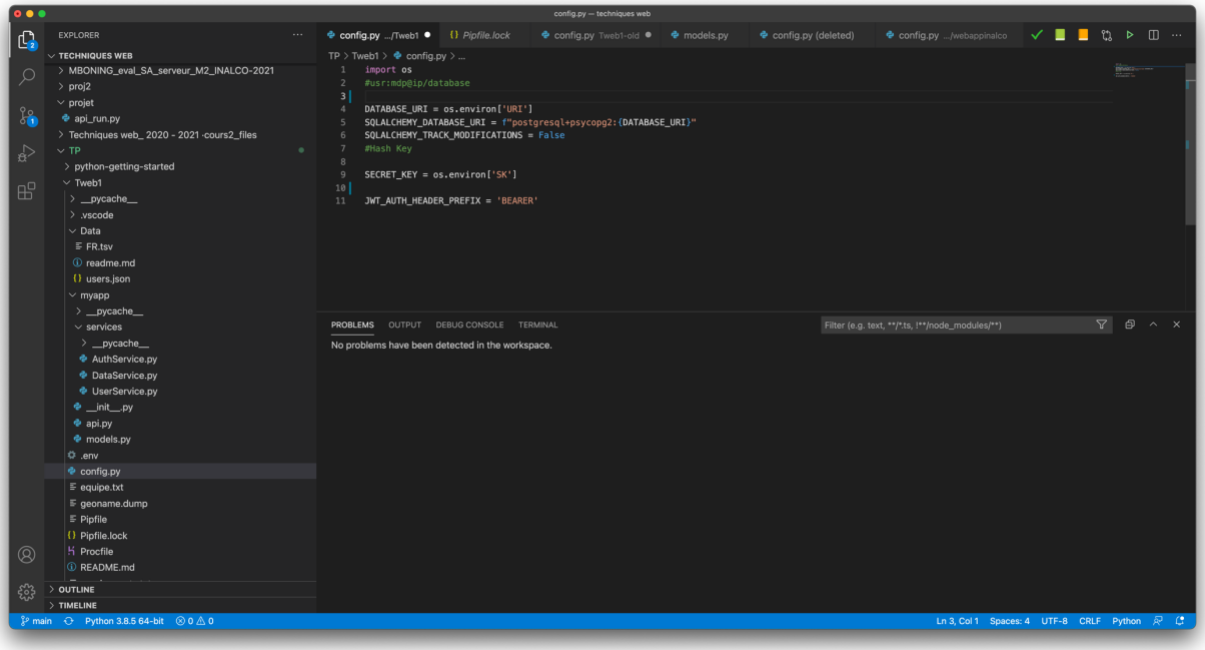
Puis un fichier Procfile, qui définit les processus/commandes qui doivent être lancés pour que Heroku puisse faire tourner notre application, où nous avons mis la commande web pour indiquer à Heroku de lancer gunicorn (wsgi http serveur pour notre application).
Ensuite, nous avons ajouté une base PostgreSQL dont nous avons récupéré les variables (mot de passe, hôte, nom de la db). La limite pour un compte gratuit étant de 10,000 lignes et notre fichier de lieux en faisant 148539, nous l'avons d'abord réduit à 8000 lignes (cf. l'application ici, par exemple avec cette requête : https://tp1web.herokuapp.com/data/asciiname/Bi_* qui renvoie des lieux dont le nom ASCII commence par Bi).

Par la suite, nous avons connecté le programme à une base de données externe (stockage Amazon WS gratuit pour une année, cf. https://aws.amazon.com/fr/products/databases/ et https://aws.amazon.com/fr/getting-started/hands-on/create-connect-postgresql-db/) qui permet d'exploiter les fichiers de données en entier, voir ici : https://inappco.herokuapp.com/data/56991.

Dans les deux cas, nous avons stocké les identifiants de connexion dans une variable locale (utilisée dans le fichier config.py) :

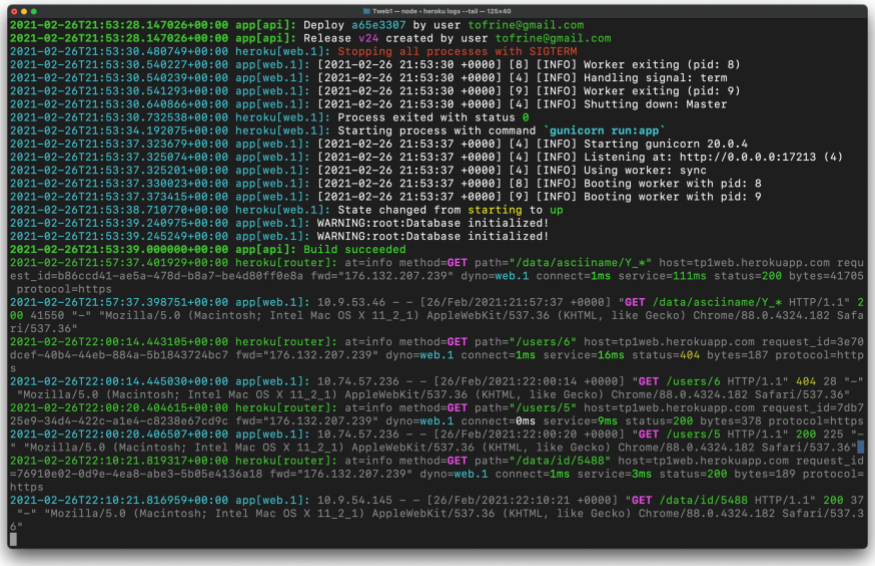


Ce qui s'écrit dans le config.py sous la forme :



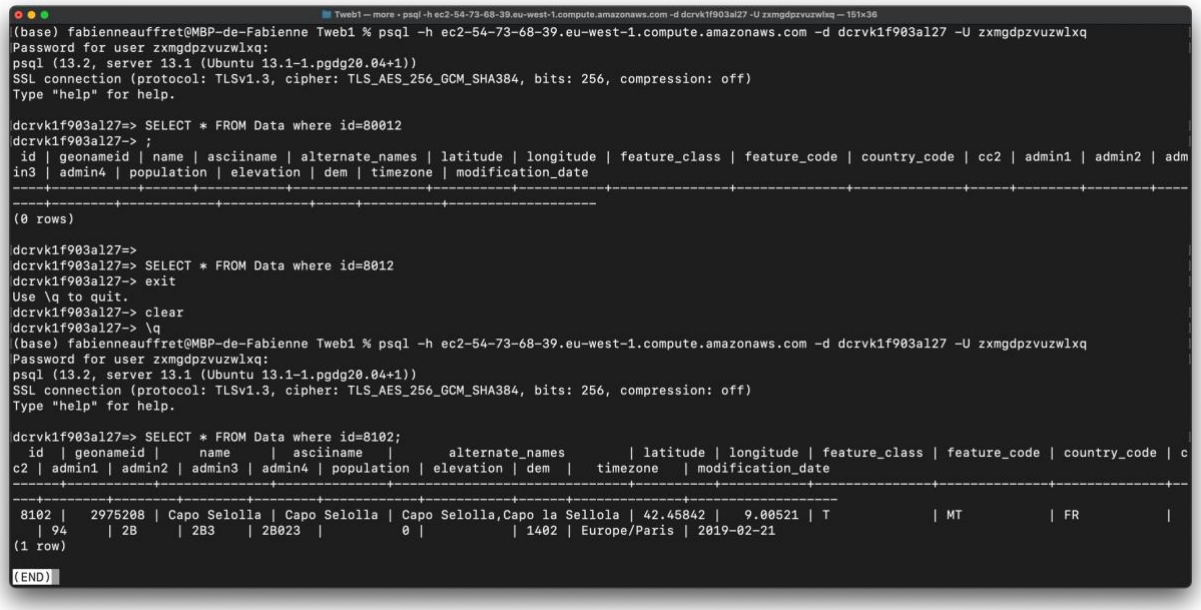
Heroku a le SSL (encryption) activé par défaut, ce qui garantit une certaine sécurité.

Une fois ces préparations faites nous avons connecté notre git à Heroku et pushé l'application pour la tester.
La commande heroku logs -–tail permet de suivre les logs et de voir que les requêtes passent bien :

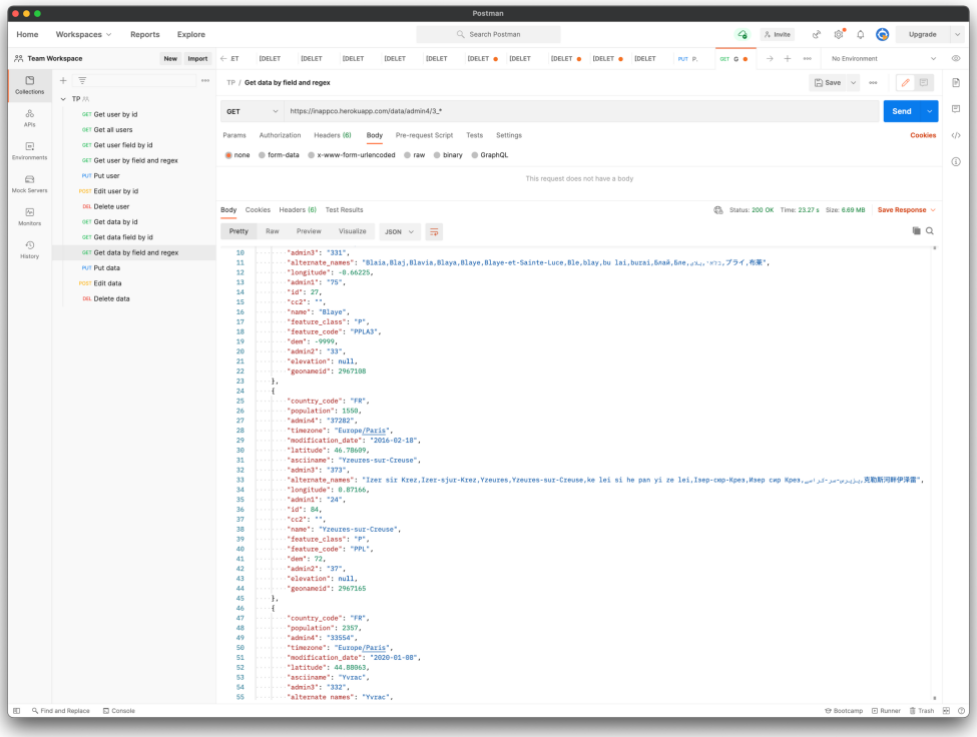


2.2 Tests sur Heroku

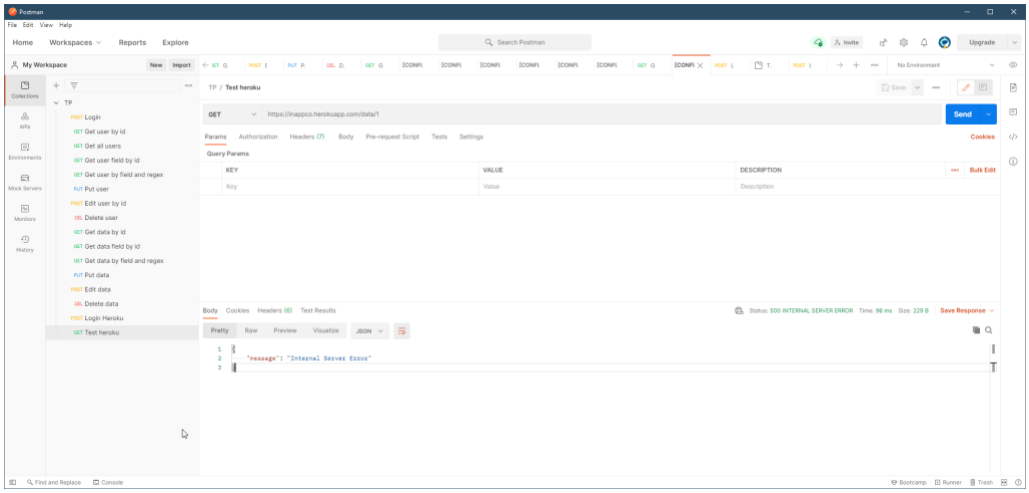
Comme en local nous avons d'abord vérifié la création de la base et son initialisation correcte en faisant des requêtes en ligne de commande, en utilisant : **psql -h hostname -d databasename -U username :**



Nous avons utilisé Postman (comme en local) pour effectuer des tests (nous avons repris le fichier des « templates de test » utilisés pour les tests en local) sur les deux tables :



Note : en cas d'erreur d'authentification, sur heroku le message d'erreur est moins détaillé qu'en local, il faut lire les logs pour savoir qu'il s'agit d'un souci d'authentification :



3. Modes d'emploi

3.1 Installation

Prérequis : une machine avec Python et Postman (https://www.postman.com/downloads/) installés.

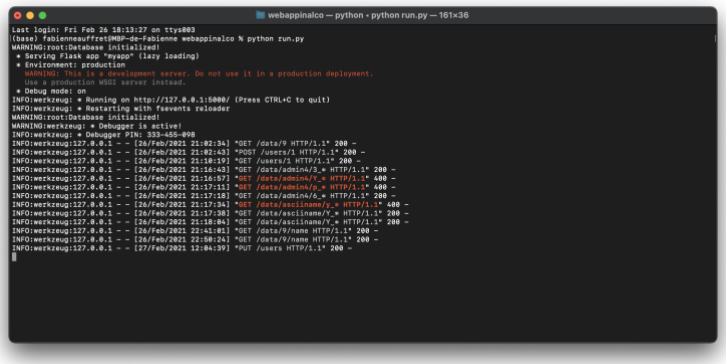
En ligne de commande :

- Installation des modules python mentionnés dans le fichier requirements.txt (pip install <nom du module>)
- Installation de PostgreSQL et création d'une base de données locale

(cf. https://www.postgresqltutorial.com/install-postgresql-linux/)

- Lancement du serveur de base de données (Postgres, avec la commande 'service postgresql start')
- Création de la base de données
- Création d'un utilisateur et de son mot de passe (Usr et pwd)
- Modification du fichier de connexion : remplacer SQLALCHEMY_DATABASE_URI = f"postgresql+psycopg2:{DATABASE_URI}" par SQLALCHEMY_DATABASE_URI = postgresql+psycopg2://usr:pwd@localhost/nom_table'
- Remplir la variable SECRET_KEY du fichier config.py par une clé que vous générerez en lançant le script keygen.py
- Lancer python run.py. Vous deviez avoir l'affichage suivant : (Le remplissage de la base de données créée en local peut prendre entre 1 et 3 minutes selon la vitesse de votre machine et retarder l'affichage) :

```
WARNING:root:Database initialized!
 * Serving Flask app "myapp" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
INFO: werkzeug: * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
INFO: werkzeug: * Restarting with fsevents reloader
WARNING: root:Database initialized!
WARNING:werkzeug: * Debugger is active!
INFO:werkzeug: * Debugger PIN: 333-455-098
```



Vous pouvez néanmoins utiliser le fichier *geoname.dump* pour peupler rapidement la nouvelle base de données à partir du dump des données non modifiées. Pour cela utilisez la commande ’`pg_restore -d nom_de_votre_db geoname.dump`’.

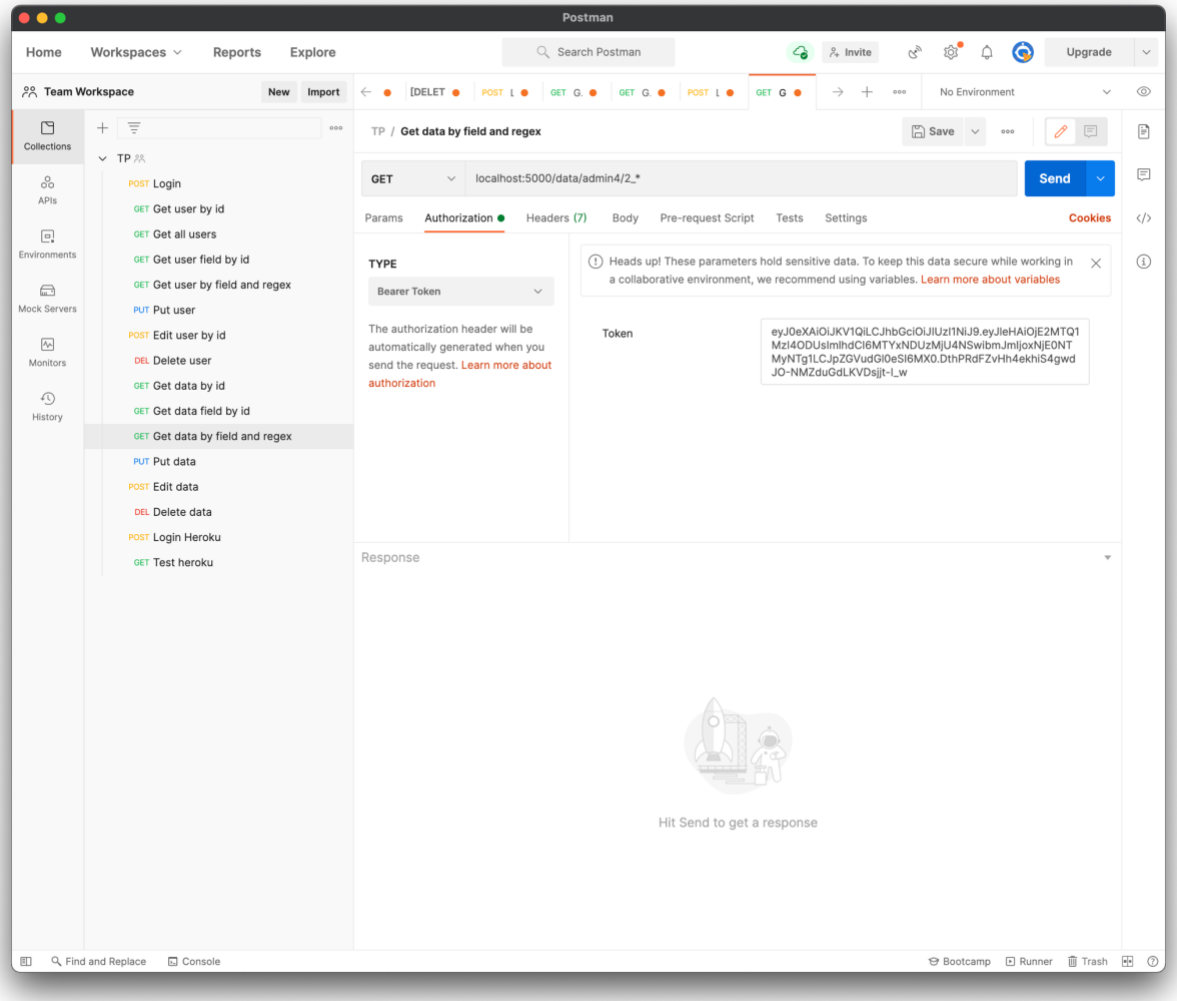
3.2 Utilisation

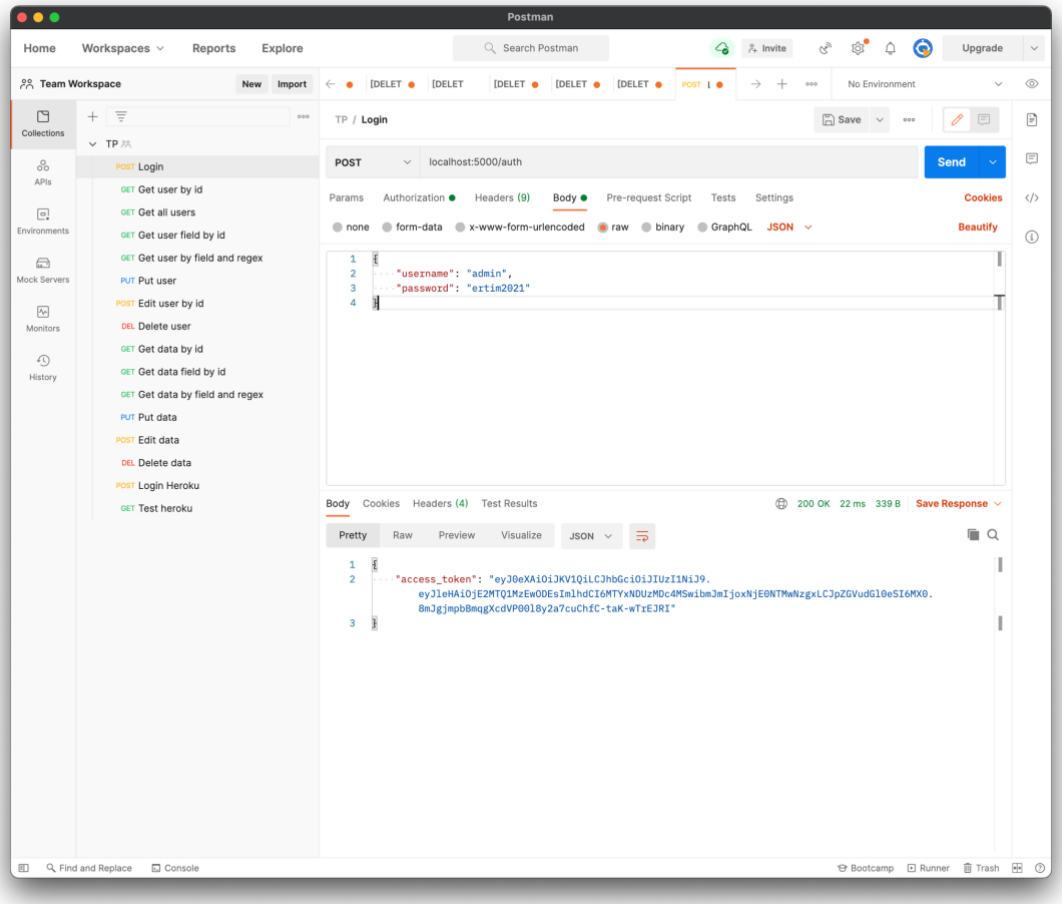
Nous avons décidé de sécuriser notre API en utilisant le module Flask-JWT. Le cahier des charges demandant simplement que les utilisateurs devaient se “connecter” et non “s’identifier” et la table des utilisateurs ne possédant pas de champ mot de passe, nous avons opté pour la mise en place d’un mot de passe administrateur.

Vous pouvez ouvrir Postman, importer le fichier *TP.postman_collection.json.* Ces templates sont là pour vous aider à former des requêtes avec Postman.

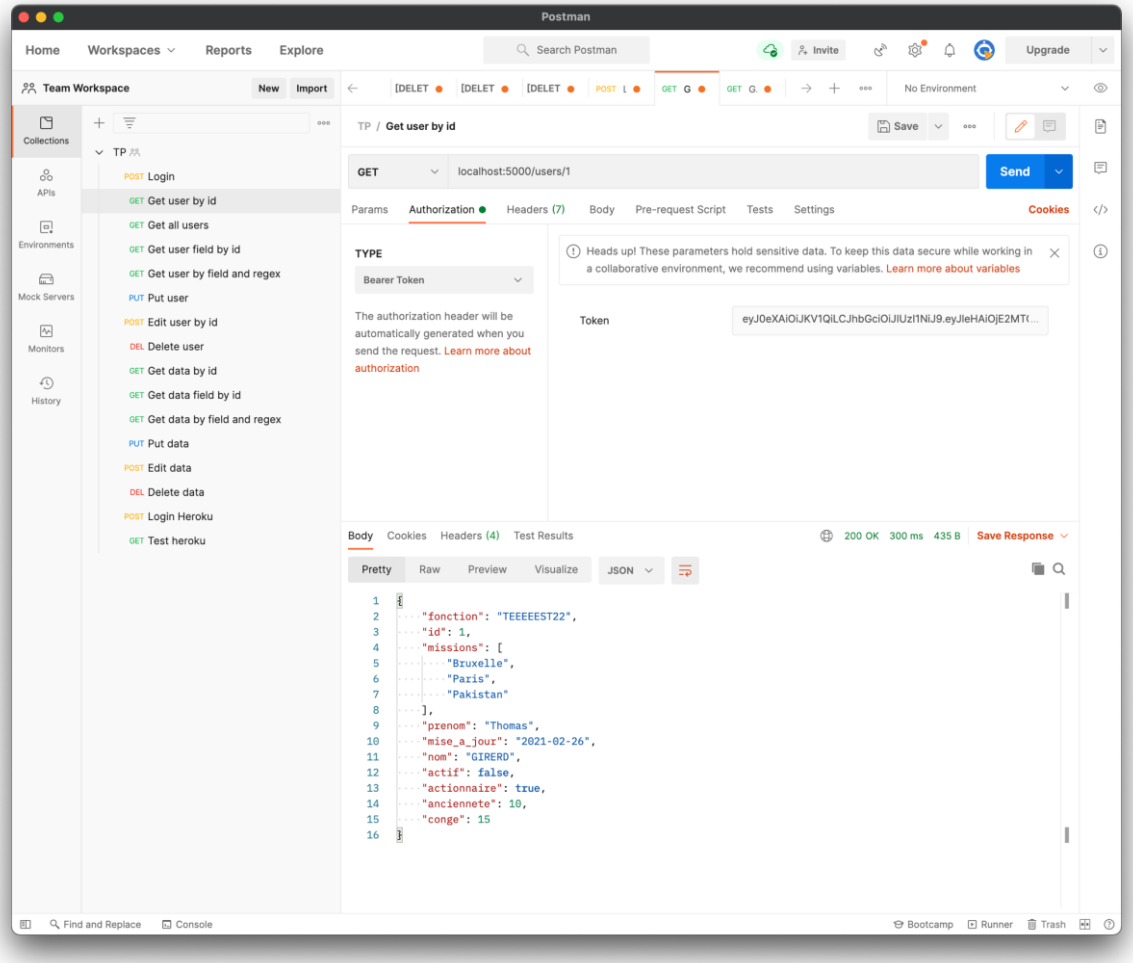
L’onglet de sortie « pretty » donne un affichage plus lisible.

Les utilisateurs doivent aller à l'adresse de l'api suivie de\auth et entrer le login et mot de passe nécessaires pour pouvoir faire des requêtes :

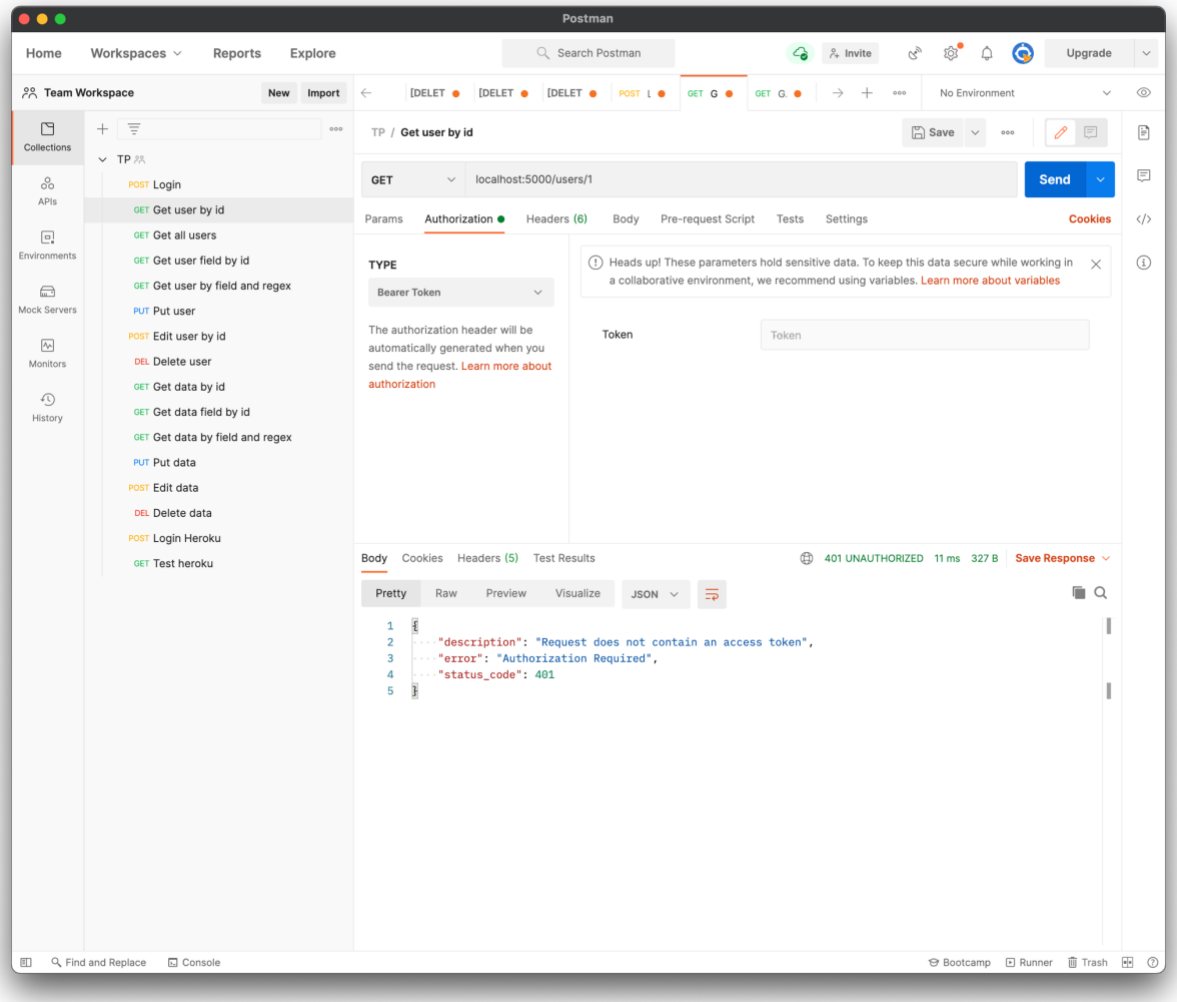



Postman
Home
Workspaces
Reports
Explore
Search Postman
Invite
Upgrade
Team Workspace
New
Import
No Environment
Collections
APIs
Environments
Mock Servers
Monitors
History
TP
POST Login
GET Get user by id
GET Get all users
GET Get user field by id
GET Get user by field and regex
PUT Put user
POST Edit user by id
DEL Delete user
GET Get data by id
GET Get data field by id
GET Get data by field and regex
PUT Put data
POST Edit data
DEL Delete data
POST Login Heroku
GET Test heroku
TP / Login
Save
POST
localhost:5000/auth
Send
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"username": "admin",
"password": "ertim2021"
}
Body
Cookies
Headers (4)
Test Results
200 OK
22 ms
339 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJleHAiOjE2MTQ1MzEwODEsImlhdCI6MTYxNDUzMDc4MSwibmJmIjoxNjE0NTMwNzgxLCJpZGVudGl0eSI6MX0.
8mJgjmpbBmqgXcdVP00l8y2a7cuChfC-taK-wTrEJRI"
}
Find and Replace
Console
Bootcamp
Runner
Trash

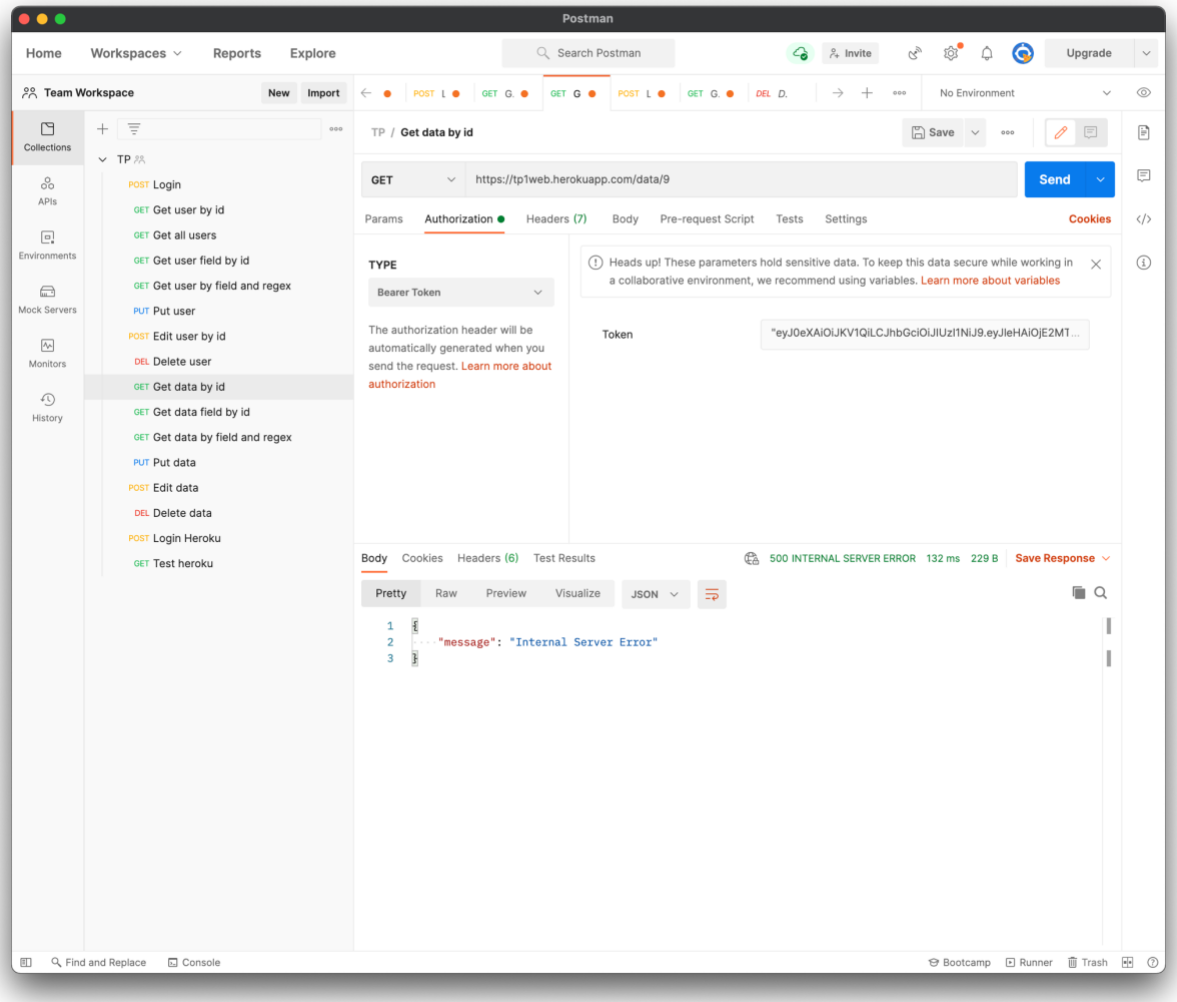
Ils obtiennent un token qu'il faudra recopier dans les autorisations (type : bearer) pour pouvoir faire les requêtes :



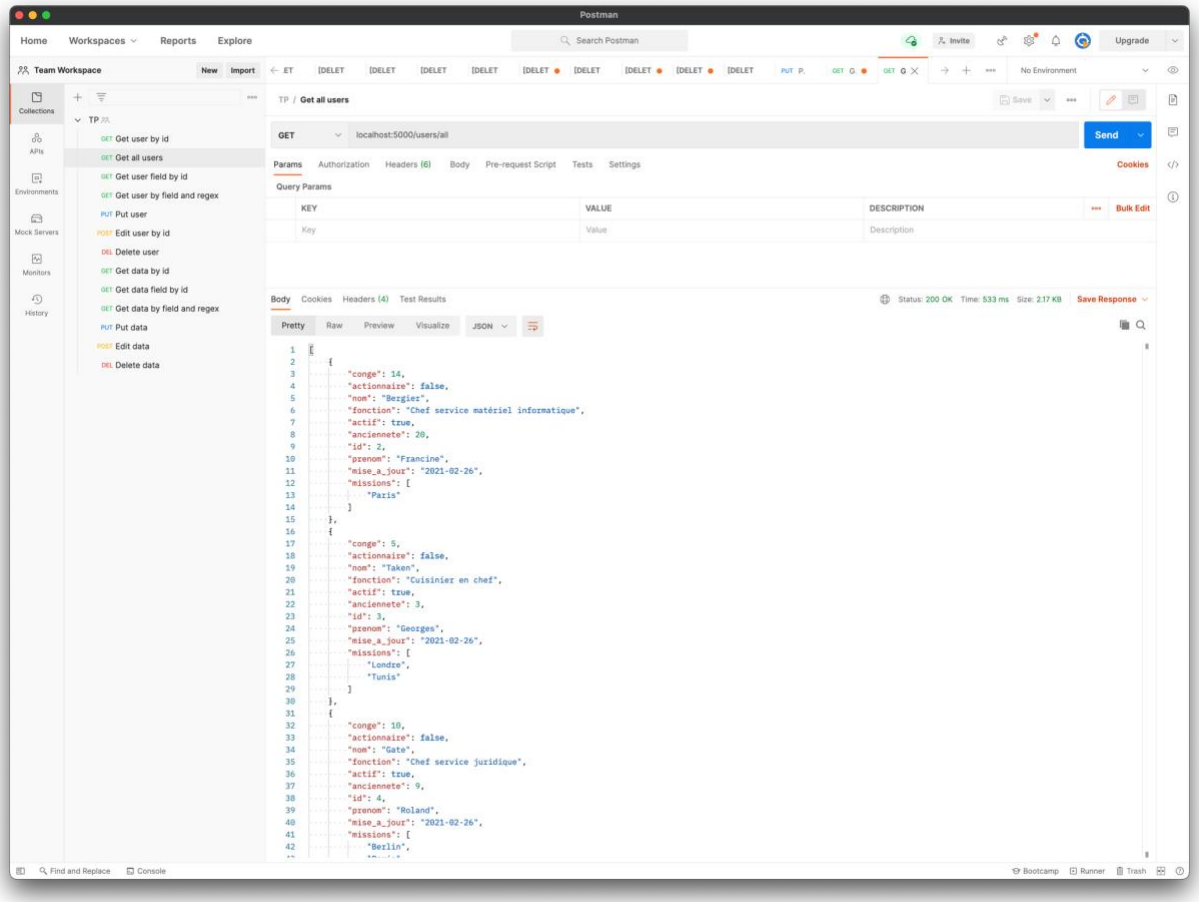
Sans celui-ci, on a :



Si on est sur une application déployée, il est possible que les messages d'erreur soient plus succincts, mais les logs server side indiquent l'erreur complète : :



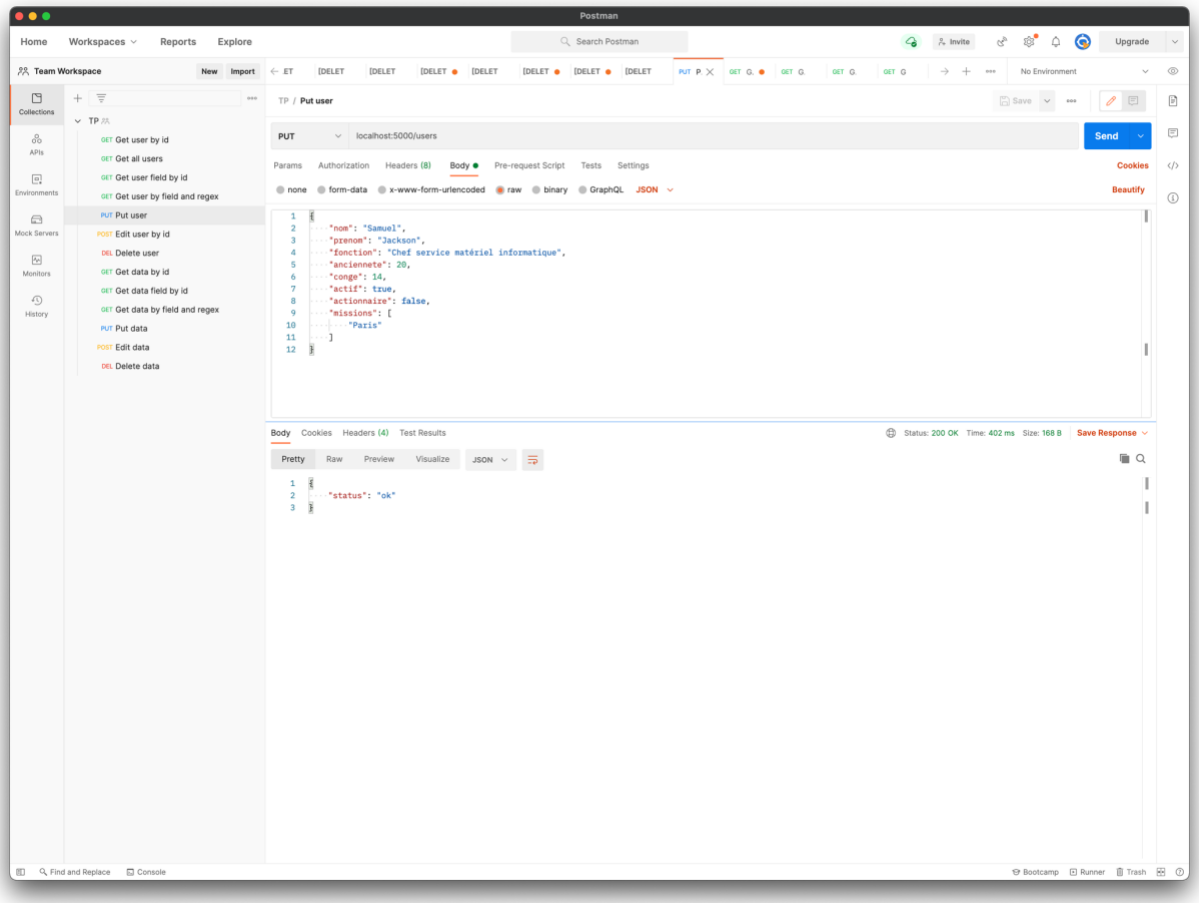
Ci-dessous, vous affichez tous les utilisateurs :



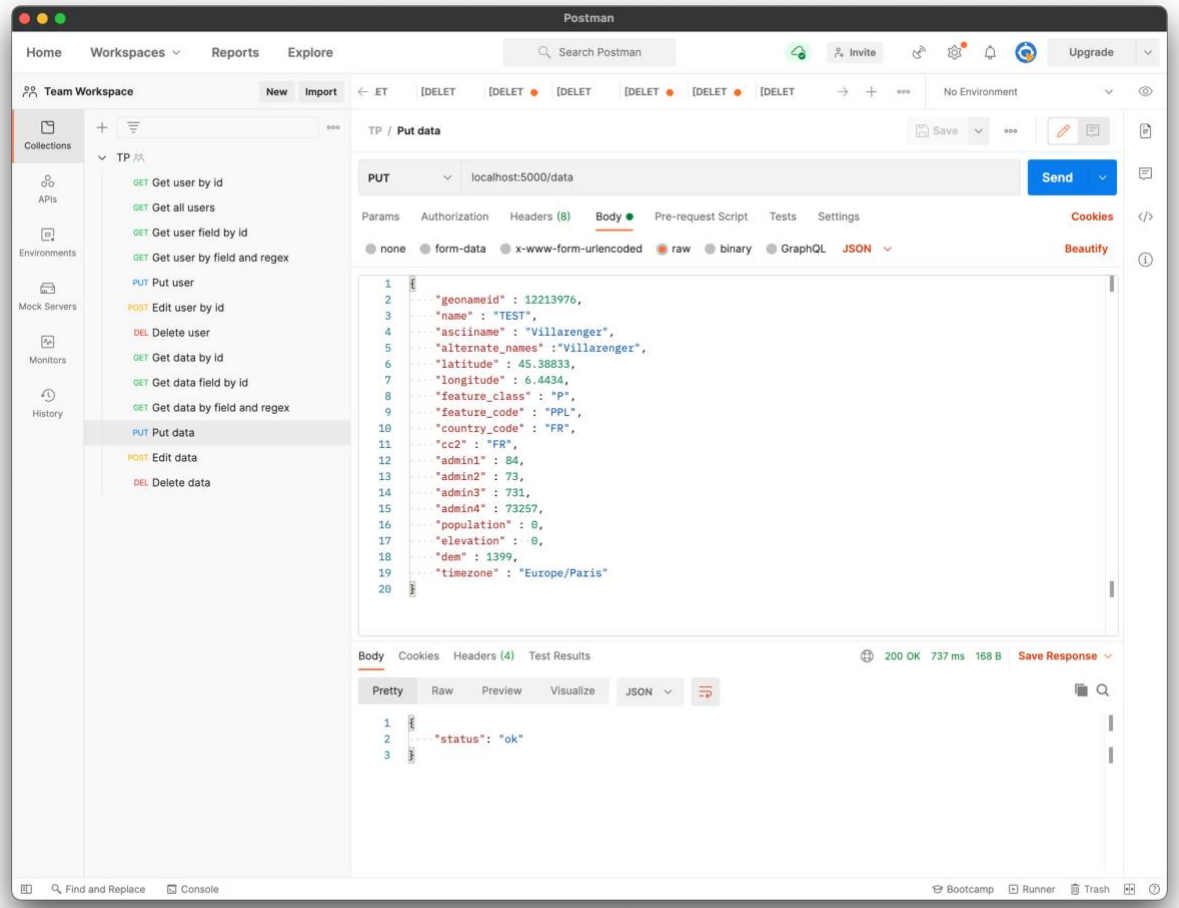
Vous pouvez ajouter un utilisateur (en utilisant la reqête « put ») :

Cliquez sur l'onglet intitulé Body. Sélectionnez ensuite le bouton brut et choisissez le format JSON.

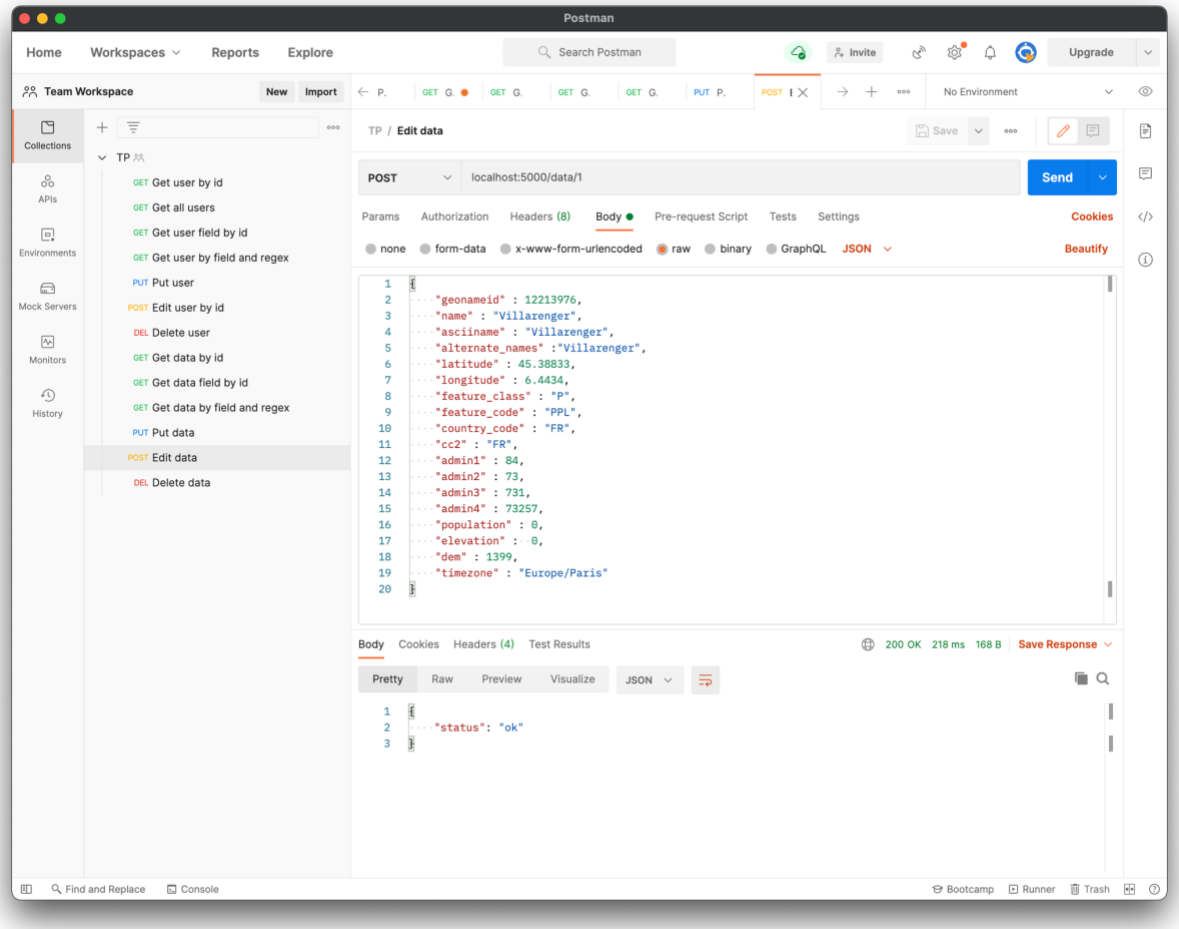
Entrez les champs nécessaires avec les infos que vous voulez ajouter dans la partie « body » et appuyez sur « Send ».



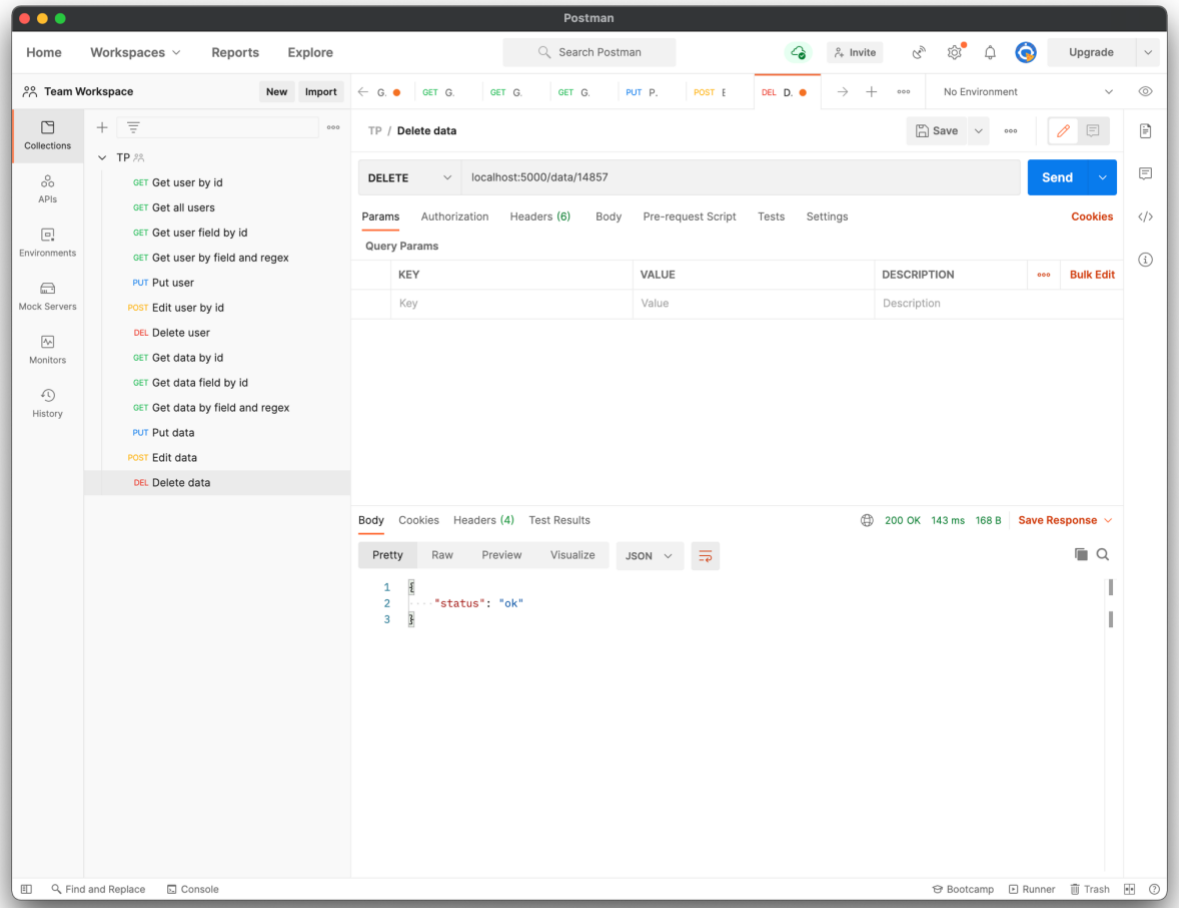
Exemple avec la table data :



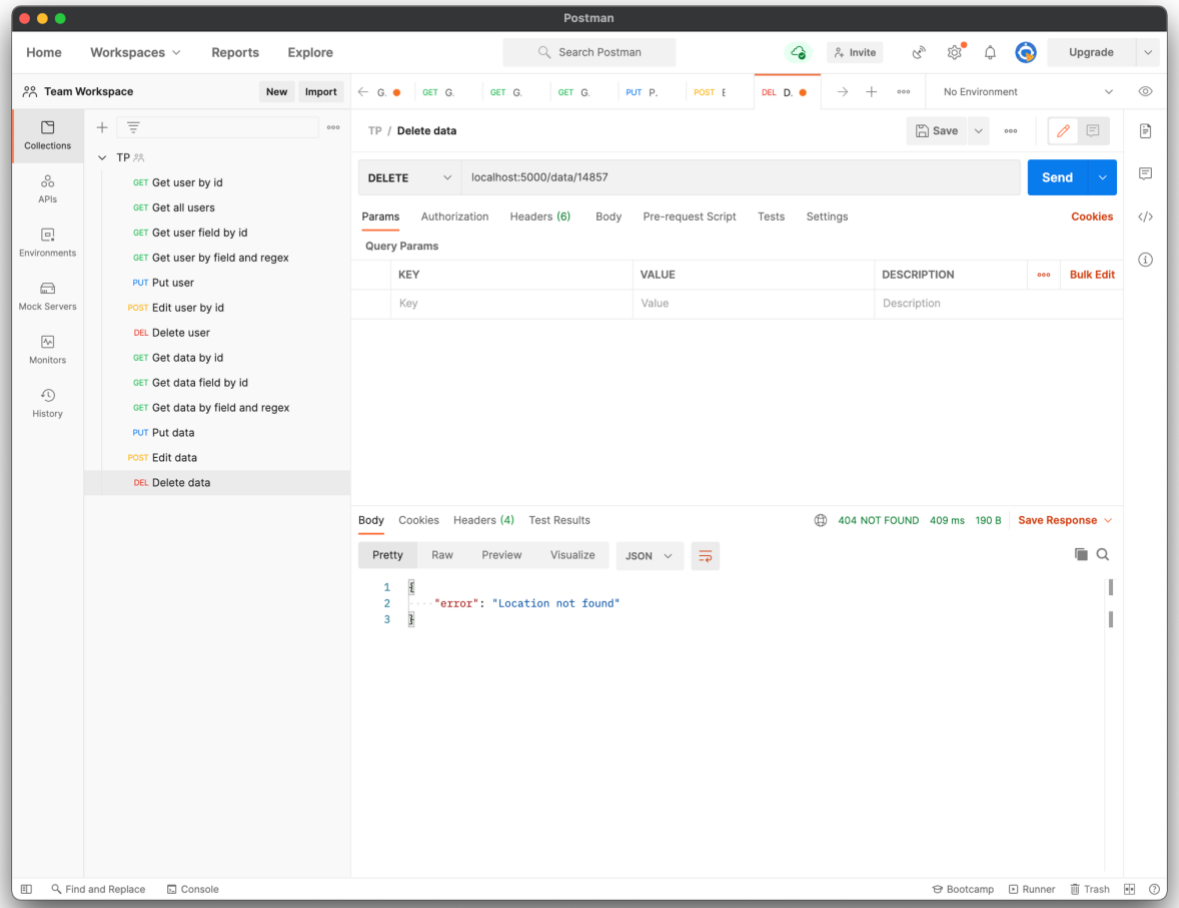
Avec « Post » vous pouvez éditer des données, il faut également cliquer sur l'onglet body (dès que l'on souhaite modifier ou ajouter des données) et y mettre les champs et informations que l'on veut changer :



Pour effacer un lieu ou un user, il faut choisir une requête « del » :

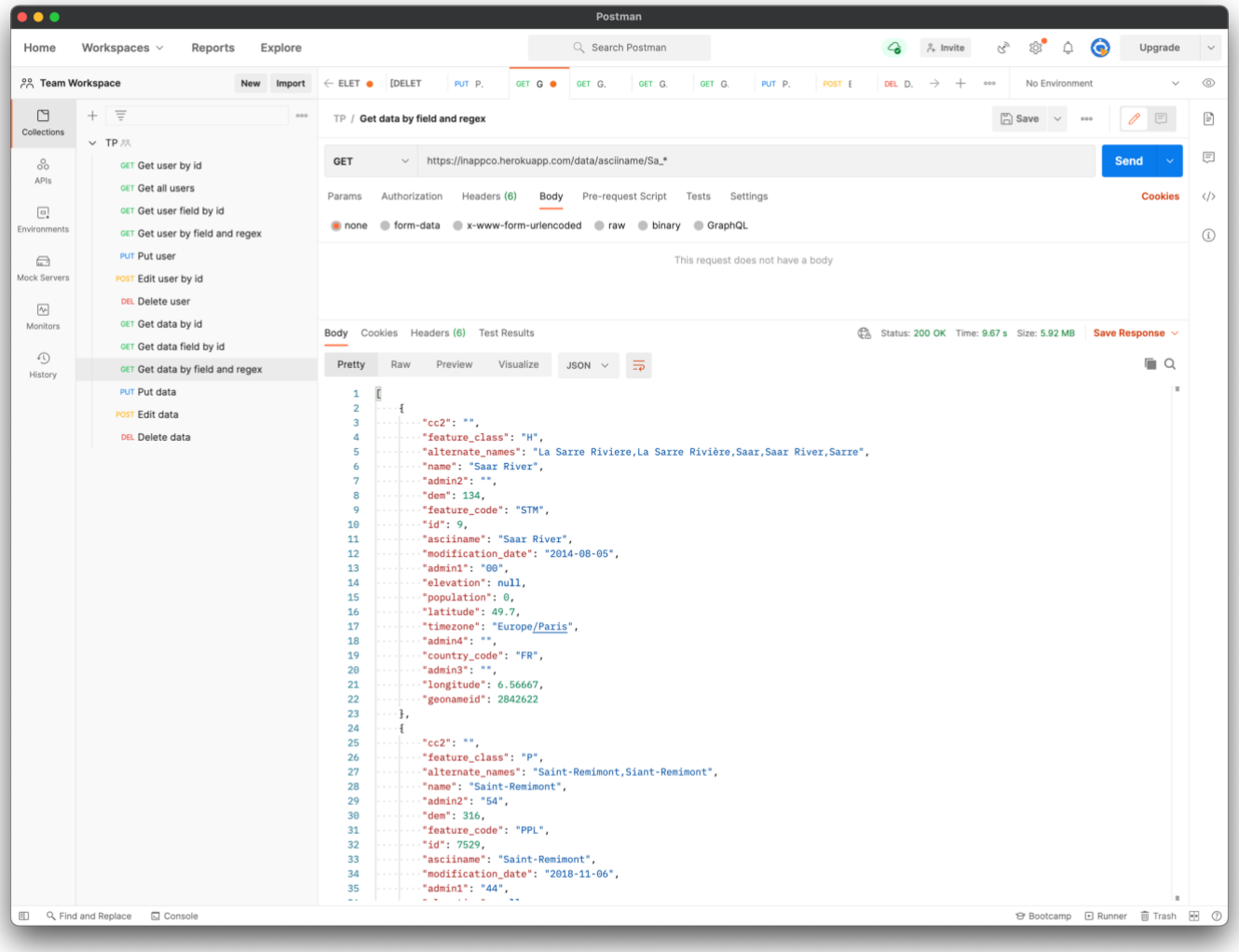


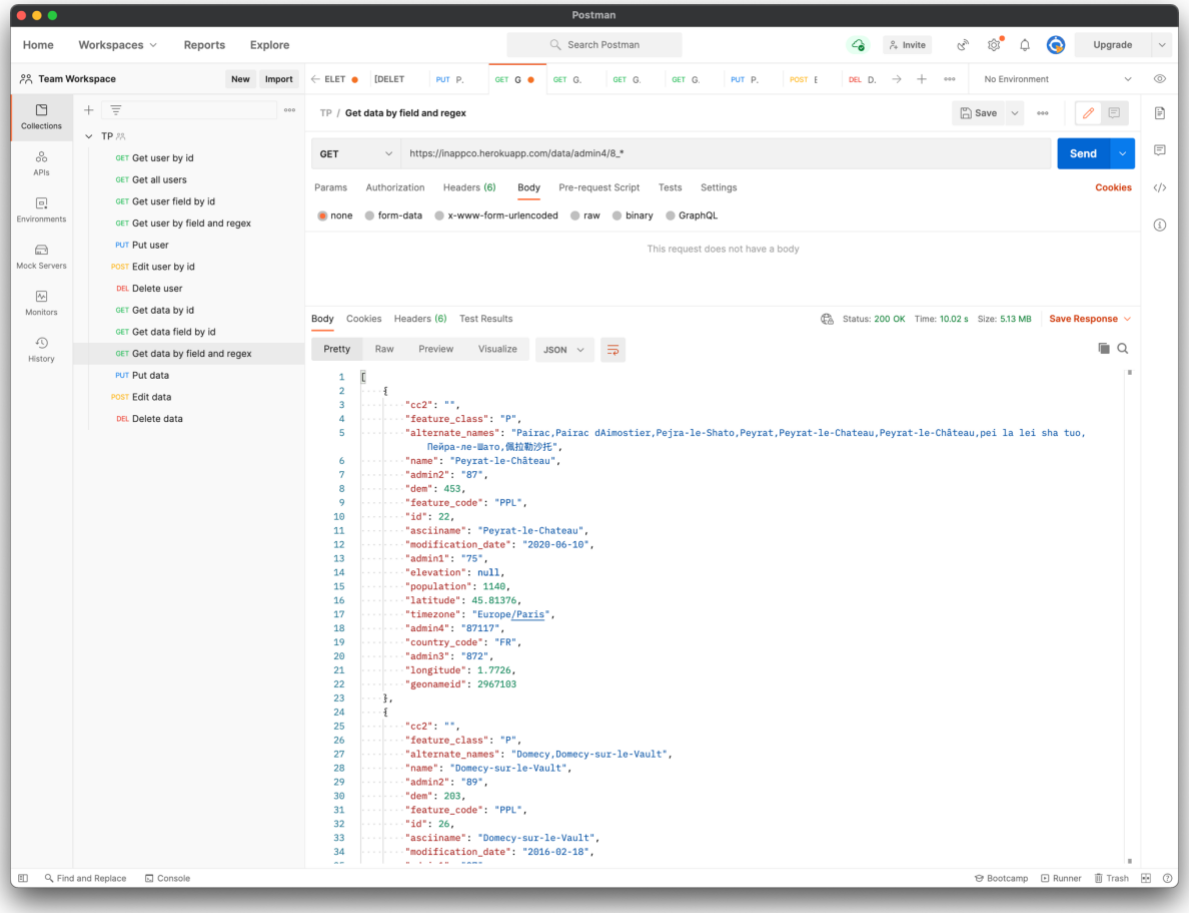
Postman vous indique les erreurs (ici on tente à nouveau d'effacer le lieu/il n'existe plus) :



On peut aussi filtrer les résultats d'une requête avec une regex, il suffit de mettre dans l'URL le champ qu'on veut filtrer et terminer la requête par une regex (champ/regex). Attention, dans la syntaxe postgresql, un caractère est désigné par « _ » (underscore) et non par un point comme souvent.

Deux exemples (toutes les requêtes allant chercher des informations dans la base sont des requêtes « get ») :





4. Conclusion & perspectives

Voilà un back-end qui n'attend plus qu'un frontend avec authentification des utilisateurs autorisés à se connecter à la table « data » via des formulaires html servis par nginx (qui n'est donc pas encore utile au niveau du back-end).