



65, rue des Grands Moulins, 75013 PARIS

**DUNUAN – LARA
GAN – MEI
NINEB – SHEHERAZADE**

MANUEL TECHNIQUE APPLICATION BACK-END

**TECHNIQUES DU WEB
M. ELVIS MBONING**

Institut National des Langues et Civilisations Orientales (I.NA.L.C.O)
Master 2 TAL IM

Paris, Février 2021

Sommaire

1.	INTRODUCTION	3
2.	ARCHITECTURE GLOBALE DU SYTEME	3
2.1	ARCHITECTURE GENERALE DE L'APPLICATION	3
2.2	COMPOSANTS	4
2.3	BASE DE DONNEES	4
2.3.1	CREATION DE LA BASE DE DONNEES	4
3.	ENVIRONNEMENT DE DEVELOPPEMENT WEB	7
3.1	MISE EN PLACE DE L'ENVIRONNEMENT DE DEVELOPPEMENT	7
4.	ARCHITECTURE DE L'APPLICATION	7
4.1.1	STRUCTURATION DU CODE	7
4.1.2	DIAGRAMMES DE CLASSES UML	9
4.1.2.1	LA METHODE GET	9
4.1.2.2	LA METHODE PUT	10
4.1.2.3	LA METHODE PATCH	10
4.1.2.4	LA METHODE DELETE.....	11
5.	LANCEMENT DE L'APPLICATION ET TEST DU FONCTIONNEMENT DE L'API	11
6.	UNIT TESTS VIA POSTMAN	11
7.	DEPLOIEMENT SUR HEROKU.....	14
8.	PISTES D'AMELIORATIONS.....	15
ANNEXE 1 : CORRESPONDANCES ENTRE LES ETIQUETTES DES VARIABLES DU FICHIER RF.TXT ET LES VARIABLES DU CODE		19

1. Introduction

Ce document spécifie les implémentations techniques nécessaires au développement et à l'installation d'une application Back-End.

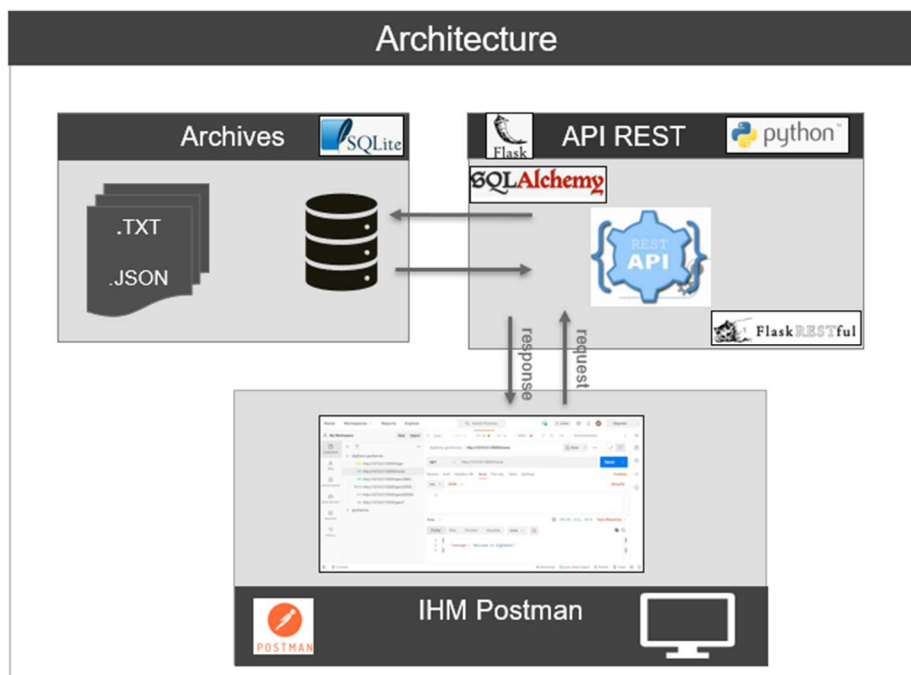
Le besoin exprimé est le suivant :

- Permettre de créer, d'accéder, de modifier et de supprimer des données géographiques appelées « GeoNames ». Ces données fournies par le client sont stockées dans un fichier de données tabulaire au format TXT.
- Mettre en place un système de gestion d'identité (connexion nécessaire pour pouvoir effectuer les différentes tâches ci-dessus). Les données concernant les utilisateurs ayant accès à l'application sont fournies dans un fichier au format JSON nommé users.json.
- Héberger l'application sur un serveur local et la rendre accessible via une URL (digidata.api.localhost).
- Sécuriser l'application si possible.

Seule la partie Back-End est traitée dans cette étude. Postman va donc permettre d'appeler et tester l'API réalisée.

2. Architecture globale du système

2.1 Architecture générale de l'application



2.2 Composants

L'API est développée en Python. Les principaux composants utilisés sont : Flask, FlaskRestful, SQLAlchemy et Marshmallow.

Flask est un micro-service web qui permet entre autres d'implémenter des API REST. Son extension **FlaskRestful** est dédiée au design des API REST et prend en charge la création rapide d'API REST. Il est conçu pour faciliter la création d'une API « from scratch ». Il encourage les meilleures pratiques avec une configuration minimale.

Marshmallow permet la sérialisation et la désérialisation des inputs et outputs dans un format json. Ce package accélère le process de validation des inputs.

SQLAlchemy est un setup basique. L'objet DB permet alors de se connecter à une base de données.

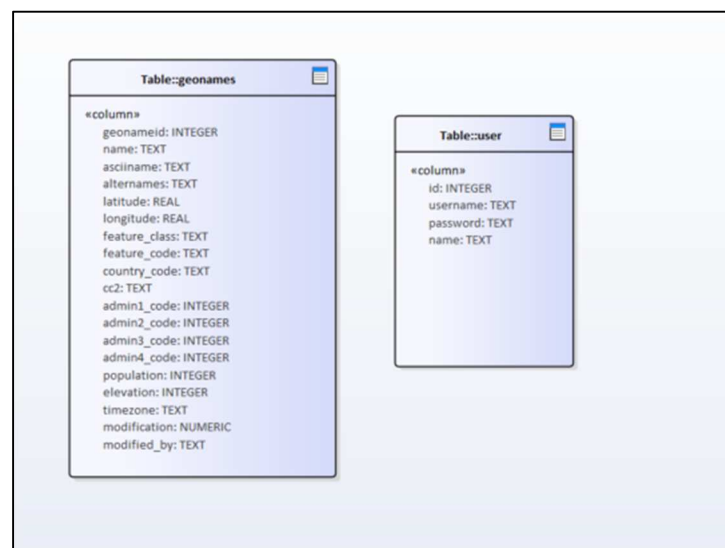
Cette API permet d'ajouter, de créer, de modifier et de supprimer des données dans une base de données.

L'API est connectée à une base de données **SQLite**.

2.3 Base de données

2.3.1 Création de la base de données

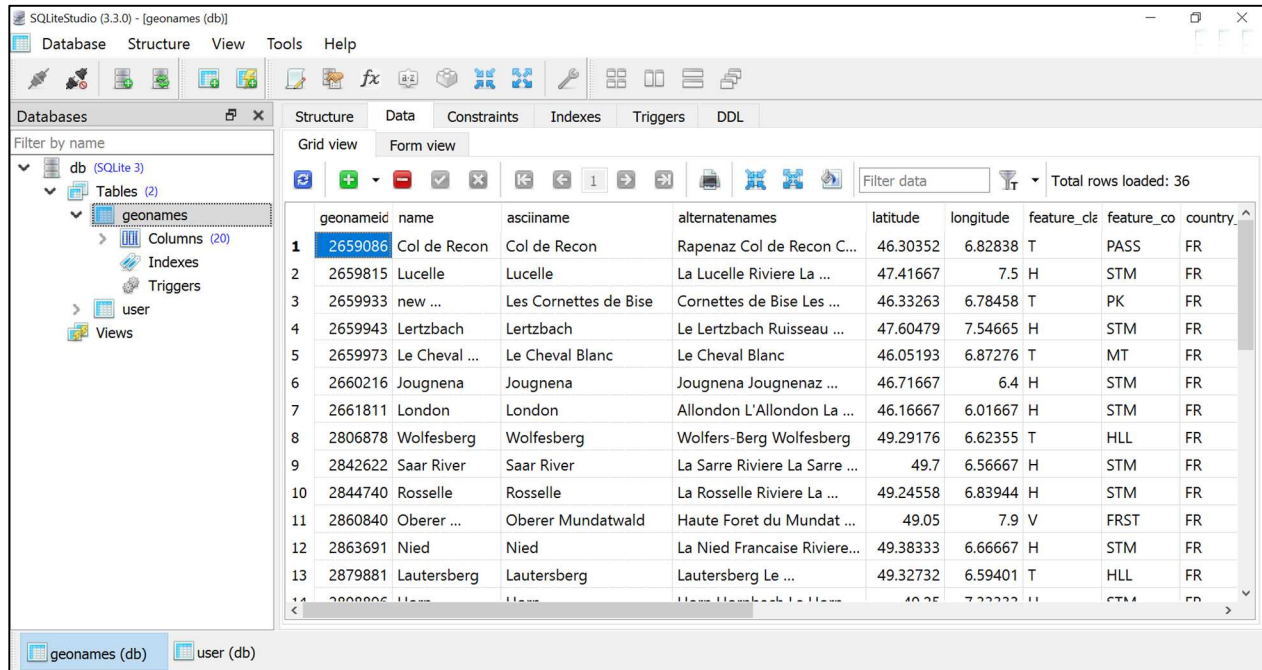
Les fichiers de données du client *users.json* et *FR.txt* ont été stockés dans une base de données avec laquelle l'API communique. Deux tables ont été créées : User et Geonames.



Geonames :

Dans la table *Geonames* sont stockés les données géographiques GeoNames (données accessibles gratuitement par Internet sous une licence Creative Commons).

Un lieu géographique est composé de 19 données/colonnes. Une variable supplémentaire a été ajoutée « *modified_by* » afin de stocker l'identifiant de la personne qui modifie cette information.



	geonameid	name	asciiname	alternatenames	latitude	longitude	feature_cls	feature_co	country
1	2659086	Col de Recon	Col de Recon	Rapenaz Col de Recon C...	46.30352	6.82838	T	PASS	FR
2	2659815	Lucelle	Lucelle	La Lucelle Riviere La ...	47.41667	7.5	H	STM	FR
3	2659933	new ...	Les Cornettes de Bise	Cornettes de Bise Les ...	46.33263	6.78458	T	PK	FR
4	2659943	Lertzbach	Lertzbach	Le Lertzbach Ruisseau ...	47.60479	7.54665	H	STM	FR
5	2659973	Le Cheval ...	Le Cheval Blanc	Le Cheval Blanc	46.05193	6.87276	T	MT	FR
6	2660216	Jougnena	Jougnena	Jougnena Jougnenaz ...	46.71667	6.4	H	STM	FR
7	2661811	London	London	Allondon L'Allondon La ...	46.16667	6.01667	H	STM	FR
8	2806878	Wolfsberg	Wolfsberg	Wolfs-Berg Wolfsberg	49.29176	6.62355	T	HLL	FR
9	2842622	Saar River	Saar River	La Sarre Riviere La Sarre ...	49.7	6.56667	H	STM	FR
10	2844740	Rosselle	Rosselle	La Rosselle Riviere La ...	49.24558	6.83944	H	STM	FR
11	2860840	Oberer ...	Oberer Mundatwald	Haute Foret du Mundat ...	49.05	7.9	V	FRST	FR
12	2863691	Nied	Nied	La Nied Francaise Riviere...	49.38333	6.66667	H	STM	FR
13	2879881	Lautersberg	Lautersberg	Lautersberg Le ...	49.32732	6.59401	T	HLL	FR

Le choix suivant a été fait pour la construction de la BD :

- Construction de la BD avant lancement de l'application mais qui n'embarque que la table *geonames*.
- Construction ou recharge de la table *user* lors du lancement de l'application. Cela permet de prendre en compte de nouveaux utilisateurs ajoutés directement dans le fichier *users.json* sans recharger les *geonames*.

N.B. :

Dans les améliorations :

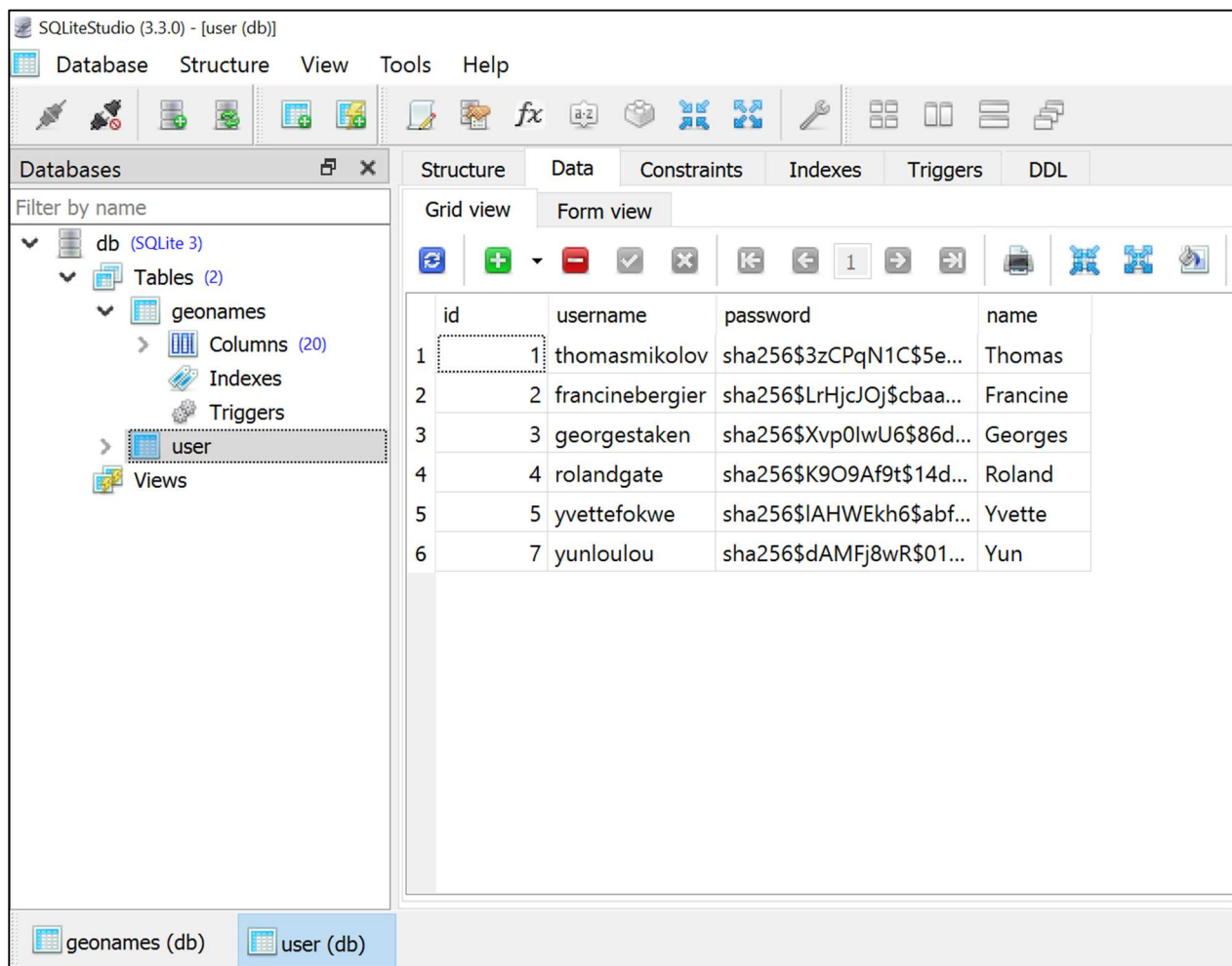
- Nous pourrions proposer au client la possibilité d'ajouter des utilisateurs directement via l'application. Droit fournit aux « super-utilisateurs » uniquement.
- Et créer la BD entièrement « one-shot » avant le lancement de l'application.

Construction de la base de données et de la table *geonames* :

Le script *add_geonames.py* du dossier *utils* crée la base de données *db.sqlite* et construit la table *geonames*. Ce script doit être lancé avant le lancement du serveur pour préparer la base de données.

Users :

Dans le tableau *User*, on stocke les informations de connexion (username, password et name). Il s'agit des utilisateurs qui peuvent accéder au système grâce à l'identifiant et au mot de passe.



Un mot de passe est crypté en sha256 pour bien protéger la sécurité de données des utilisateurs.

L'identifiant, par défaut, est la combinaison du prénom et du nom en minuscule. Le mot de passe est l'identifiant combiné avec l'id correspondant du fichier *users.json*.

Par exemple, le compte pour M. Smikolov est :

- username : thomasmikolov
- password : thomasmikolov001

Construction de la table user :

Le script pour la construction de cette table est *add_users.py* du dossier *utils*.

Une fois l'application lancée, cette table est soit créée (lors de l'installation et du premier lancement), soit est reconstruite en lisant le fichier *users.json*.

Pour un besoin d'ajout/de suppression/de modification d'informations concernant les utilisateurs : il suffit d'effectuer ces tâches directement dans le fichier JSON.

3. Environnement de développement web

3.1 Mise en place de l'environnement de développement

- Création d'un environnement virtuel pour le développement de l'application Flask :

```
$ pip install virtualenv
$ virtualenv env
```

- Activation de l'environnement virtuel :

```
$ source env/bin/activate
```

- Installation des librairies, packages, etc... nécessaires avec pip et le fichier *requirements.txt* (fichier dans lequel on trouve les configurations nécessaires qui répondent aux besoins et dépendances) :

```
(env) $ pip install -r requirements.txt
(env) $ pip install gunicorn
```

Le fichier *requirements.txt* se trouve à la racine du projet.

4. Architecture de l'application

4.1.1 Structuration du code

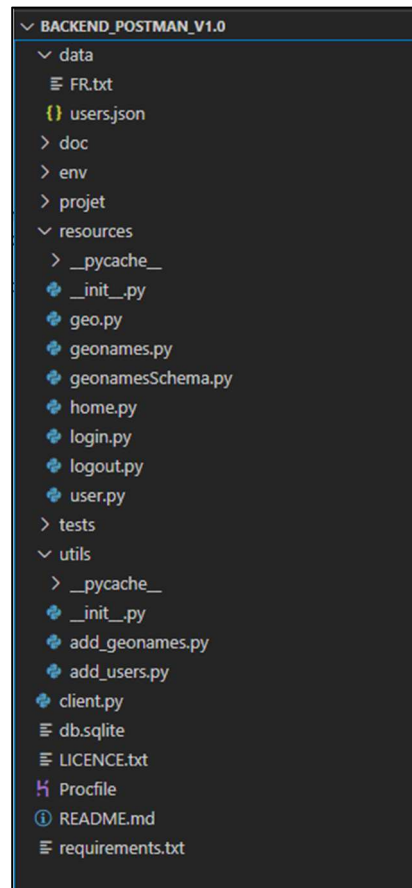
Le programme principal est *client.py* : il se trouve à la racine du projet.

Les classes développées pour le besoin sont dans le dossier *ressources*.

Le dossier *utils* contient les scripts Python qui traitent les tables et la base de données. Les données sont embarquées dans le fichier *db.sqlite* qui se trouve dans la racine du projet.

Les fichiers de données *users.json* et *FR.txt* sont dans le dossier *data*

Ci-dessous une image de la structuration du projet.



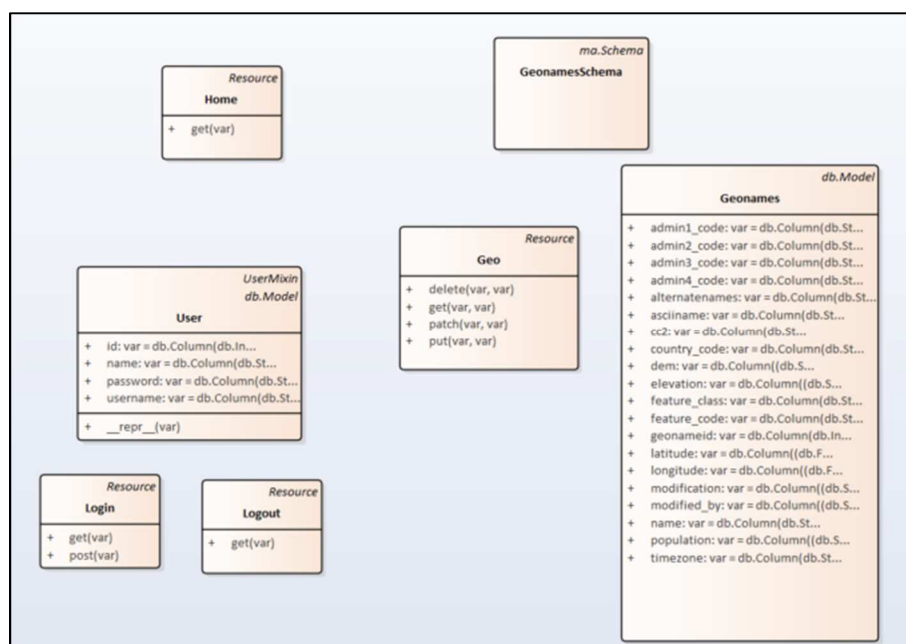
Ci-dessous un tableau qui indique les fichiers et le nom de dossier auxquels ils appartiennent ainsi que leur description.

Nom de fichier	Nom de dossier	Description
geo.py	resources	Permet de manipuler le tableau geonames
geonames.py	resources	class des Geonames
geonamesSchema.py	resources	Schema pour validation des structures geo
home.py	resources	Accès à la page d'accueil

login.py	resources	Classe qui se charge de la connexion d'un utilisateur
logout.py	resources	Classe qui se charge de la déconnexion d'un utilisateur
user.py	resources	classe User
add_user.py	utils	Ajout de la table user dans la bd
add_geonames.py	utils	Ajout de la table geonames dans la bd
users.json	data	Les données de la table user
FR.txt	data	Les données de la table geonames

4.1.2 Diagrammes de classes UML

Ci-dessous un diagramme de l'ensemble des classes ainsi que de leurs attributs.



4.1.2.1 La méthode get

La classe *Ressource* de Restful permet de regrouper les méthodes http par type de données comme indiqué dans le programme principal *client.py* :

```
# routes for each methods from there page
api.add_resource(resources.Home, "/", "/home")
api.add_resource(resources.Login, "/login")
api.add_resource(resources.Logout, "/logout")
api.add_resource(resources.Geo, "/geo/<int:geo_id>")
```

Ainsi plusieurs méthodes GET/PUT/PATCH/DELETE peuvent être implémentées pour l'API. Le décorateur `@api.add_resource(...)` est utilisé pour spécifier les URL qui seront associées à une ressource donnée.

Pour la classe `geo`, un type de paramètre est spécifié à l'aide du nom d'un convertisseur (`int` ici) et de deux points. Tout traitement sur les geonames nécessite de donner l'identifiant du geoname `geo_id`.

Un focus est fait sur la classe `Geo` qui est le cœur du sujet.

La méthode `get` qui prend en argument l'identifiant du géo-point retourne les informations concernant ce `geoPoint`, renvoie un message d'erreur s'il n'existe pas.

```
@login_required
def get(self, geo_id):
    """ Getting specific geo_id
    :param geo_id: geonameid (mandatory field)
    :return: information about this geoPoint if is in the DB else an error message
    """
```

4.1.2.2 La méthode put

La méthode `put` qui prend en argument l'identifiant du géo-point ajoute des informations concernant un nouveau `geoPoint`, renvoie un message d'erreur s'il existe.

```
@login_required
def put(self, geo_id):
    """ Putting (adding) a specific geo_id
    :param geo_id: geonameid (mandatory field)
    :return: add informations about this new geoPoint if is in the DB an error message is given, else it's OK.
    """
```

4.1.2.3 La méthode patch

La méthode `patch` qui prend en argument l'identifiant du géo-point modifie des informations concernant un `geoPoint`, renvoie un message d'erreur s'il n'existe pas.

```
@login_required
def patch(self, geo_id):
    """ Updating a specific geo_id
    :param geo_id: geonameid (mandatory field)
    :return: update informations about this new geoPoint if is not in the DB an error message is given, else it's OK.
    """
```

4.1.2.4 La méthode delete

La méthode *delete* qui prend en argument l'identifiant du géo-point supprime le geoPoint et ses informations, renvoie un message d'erreur s'il n'existe pas.

```
@login_required
def delete(self, geo_id):
    """ Deleting a specific geo_id
    :param geo_id: geonameid (mandatory field)
    :return: delete the geoPoint with all his informations. If is not in the DB an error message is given, else a confirmation is given.
    """
```

5. Lancement de l'application et test du fonctionnement de l'API

Prérequis :

Installation de Postman : à télécharger via le lien

<https://www.postman.com/downloads/>

Command :

- **Étape 0 (optionnel) :**

Si le fichier db.sqlite n'existe pas :

- ✓ Lancez la commande : **python3 ./utils/add_geonames.py** pour construire la bd à partir des données du fichier FR.txt.

- **Étape 1 :**

- ✓ Lancez : **python3 client.py**

Toutes les requêtes s'exécutent à l'adresse ***digidata.api.localhost*** sur Postman.

6. Unit tests via Postman

Certains tests unitaires réalisés sont enregistrés dans le dossier ***tests***.

Voici quelques tests.

Status code is 200:

The screenshot shows the Postman interface for a GET request to `digidata.api.localhost/login`. The **Tests** tab is active, displaying a JavaScript test script:

```
1 pm.test("Status code is 200", function () {
2   pm.response.to.have.status(200);
3 });
```

The test results show a **PASS** status with the message "Status code is 200". A tooltip for the **200 OK** status code is visible, explaining that it is the standard response for successful HTTP requests.

Successful POST request

The screenshot shows the Postman interface for a POST request to `digidata.api.localhost/login`. The **Tests** tab is active, displaying a JavaScript test script:

```
1 pm.test("Successful POST request", function () {
2   pm.expect(pm.response.code).to.be.oneOf([201, 202]);
3 });
```

The test results show a **PASS** status with the message "Successful POST request". The response status is **201 CREATED**.

- ⇒ Response body : JSON value check
- ⇒ PASS : Dans cet exemple, la valeur attendue est «Vaysset» qui est égale à la valeur de test. Le test est réussi.

tests / test_value

GET http://digidata.api.localhost/geo/2970282

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

```
1 pm.test("Test value of key : name", function () {
2   ...var jsonData = pm.response.json();
3   ...pm.expect(jsonData.name).toEqual('Vaysset');
4 });
```

Test scripts are written in JavaScript, and are run after the response is received. [Learn more about tests scripts](#)

SNIPPETS

Response body: Contains string

Response body: JSON value check

Body Cookies (1) Headers (6) Test Results (1/1) 200 OK 153 ms 694 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   ..."geonameid": 2970282,
3   ..."name": "Vaysset",
4   ..."asciiname": "Vaysset",
5   ..."alternatenames": "Vaysset Veyssset",
6   ..."latitude": 45.3522,
7   ..."longitude": 2.74801,
8   ..."feature_class": "P",
9   ..."feature_code": "PPL",
```

tests / test_value

GET http://digidata.api.localhost/geo/2970282

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

```
1 pm.test("Test value of key : name", function () {
2   ...var jsonData = pm.response.json();
3   ...pm.expect(jsonData.name).toEqual('Vaysset');
4 });
```

Test scripts are written in JavaScript, and are run after the response is received. [Learn more about tests scripts](#)

SNIPPETS

Get an environment variable

Get a global variable

Body Cookies (1) Headers (6) Test Results (1/1) 200 OK 108 ms 694 B Save Response

All Passed Skipped Failed

PASS Test geo name

FAIL : Nous nous sommes également assurés que nos tests échouent afin de vérifier les résultats attendus. Dans cet exemple, la valeur attendue est «Vaysset» mais nous l'avons testée avec la valeur «Paris» qui devrait échouer.

tests / test_value

GET http://digidata.api.localhost/geo/2970282 Send

Params Authorization Headers (8) Body Pre-request Script **Tests** Settings Cookies

```

1 pm.test("Test value of key : name", function () {
2   ...var jsonData = pm.response.json();
3   ...pm.expect(jsonData.name).toEqual('Paris');
4 });

```

Test scripts are written in JavaScript, and are run after the response is received. [Learn more about tests scripts](#)

SNIPPETS

Response body: Contains string

Response body: JSON value check

Body Cookies (1) Headers (6) **Test Results (0/1)** 200 OK 131 ms 694 B Save Response

All Passed Skipped Failed

FAIL Test value of key : name | AssertionError: expected 'Vaysset' to deeply equal 'Paris'

Response time is less than 200ms

tests / test_response_time

GET http://digidata.api.localhost/geo/2970282 Send

Params Authorization Headers (8) Body Pre-request Script **Tests** Settings Cookies

```

1 pm.test("Response time is less than 200ms", function () {
2   ...pm.expect(pm.response.responseTime).toBeBelow(200);
3 });

```

Test scripts are written in JavaScript, and are run after the response is received. [Learn more about tests scripts](#)

SNIPPETS

Get an environment variable

Get a global variable

Body Cookies (1) Headers (6) **Test Results (1/1)** 200 OK 29 ms 694 B Save Response

All Passed Skipped Failed

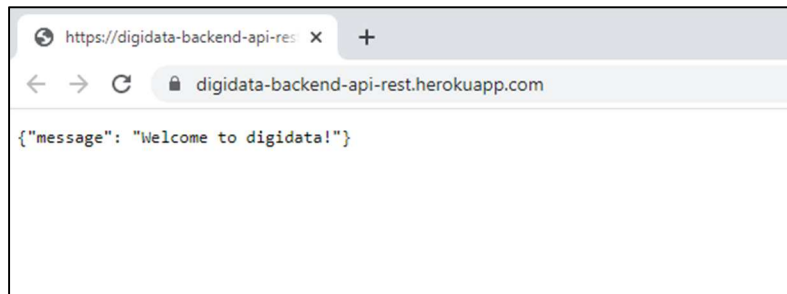
PASS Response time is less than 200ms

7. Déploiement sur Heroku

- Pour déployer l'application sur Heroku:
 - Création de Procfile :
(env) \$ echo "web: gunicorn client:app" > Procfile

- Déploiement sur Heroku :
(env) \$ heroku login
(env) \$ git init
(env) \$ git add .
(env) \$ git commit -m "Init app"
(env) \$ heroku create digidata-backend-api-rest
(env) \$ git remote -v
(env) \$ git push heroku master

Ci-dessous le résultat de l'installation sur Heroku:



8. Pistes d'améliorations

- La recherche d'informations s'effectue à travers l'ID *geonameid* uniquement. Dans une version future, il est envisageable d'étendre cette fonctionnalité par exemple en prenant le nom du geopoint et le code du pays. Voir étendre à plus de champs sans se restreindre à un ID qui est une clé unique.
 - Le code peut encore être optimisé et l'outil pytest lancé pour vérifier la qualité du code.
 - LA génération automatique de la documentation du code de l'API : nous avons entamé ce travail en nous intéressant à Swagger qui offre une interface très intéressante modulo un fichier de config à compléter et quelques insertions dans le code. Sphinx (api-doc) a aussi été étudié/utilisé mais son fonctionnement n'a pas abouti au résultat attendu. Si-dessous une capture d'écran du résultat actuel.
- ⇒ Il faut là encore lire un peu et tester avant d'arriver à la documentation souhaitée.
- L'automatisation des tests de l'API en utilisant des outils disponibles comme la librairie Python unittest ou du JS pour Postman mais il faut apprendre JS....

N.B :

Nous avons aussi commencé le développement front-End avec nos connaissances et lectures. Ci-joint le lien <https://demofrontendflask.herokuapp.com/>

Vous pouvez vous logger grâce au compte de Monsieur Mikolov:

- username : thomasmikolov
- password : thomasmikolov001

The screenshot shows a web application for 'digiData backend' documentation. On the left is a dark sidebar with a blue header containing the site name and a search bar. The sidebar lists 'CONTENTS:' with 'backend_postman_v6' selected. The main content area has a light blue header with a home icon, a welcome message, and a 'View page source' link. The main content includes a 'Welcome to digiData backend's documentation!' heading, a 'Contents:' section with a list of items (backend_postman_v6, client module, resources package, utils package), and an 'Indices and tables' section with links to Index, Module Index, and Search Page. A 'Next' button with a right arrow is at the bottom right. The footer contains copyright information and mentions Sphinx and Read the Docs.

digiData backend

Search docs

CONTENTS:

backend_postman_v6

» Welcome to digiData backend's documentation! [View page source](#)

Welcome to digiData backend's documentation!

Contents:

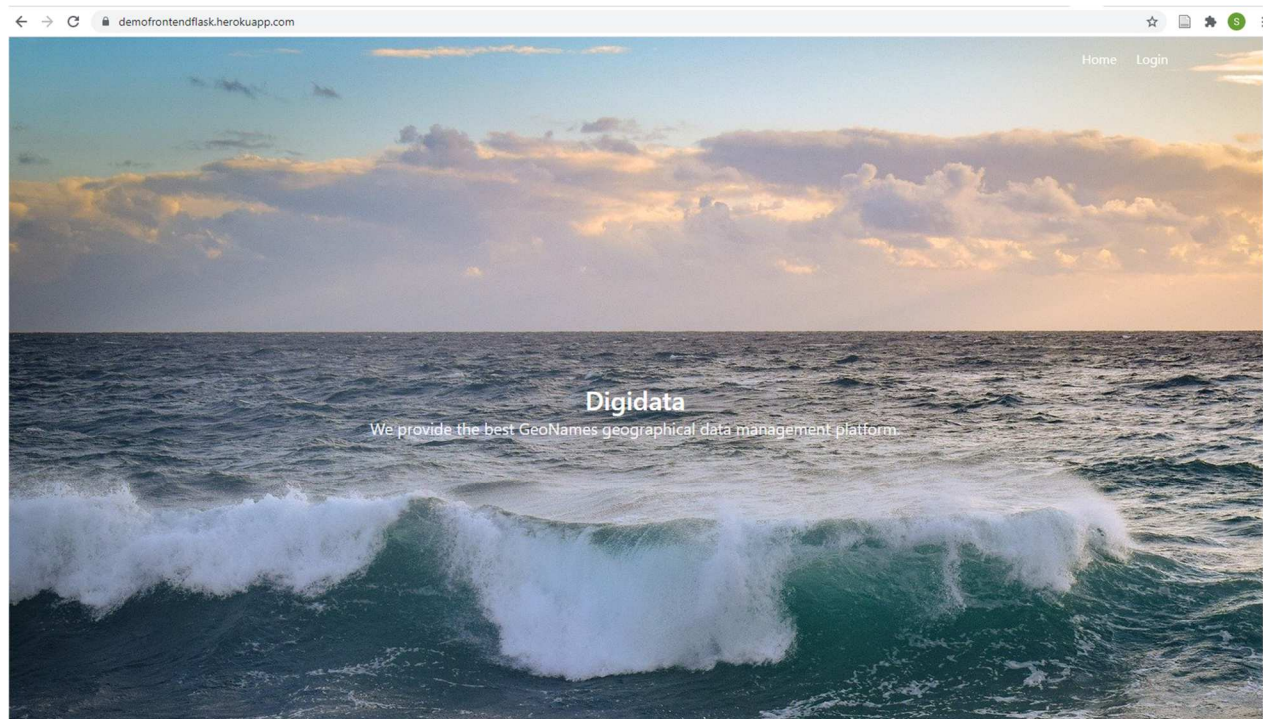
- backend_postman_v6
 - client module
 - resources package
 - utils package

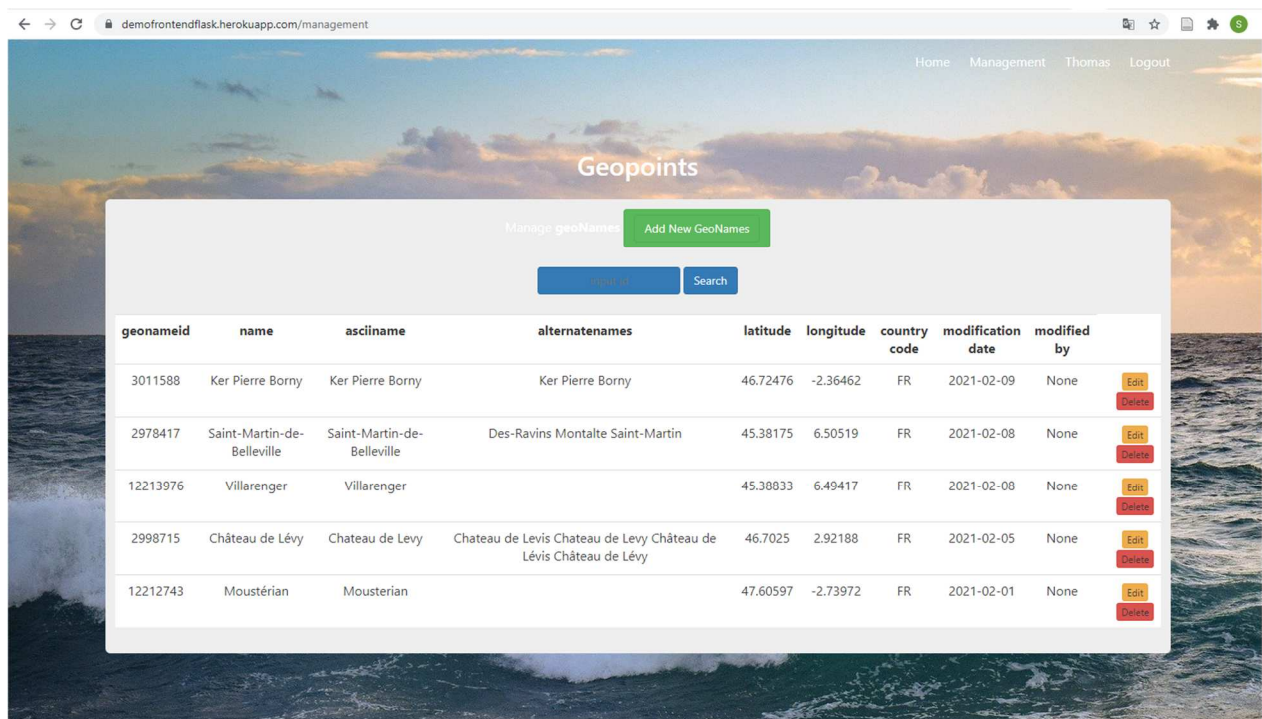
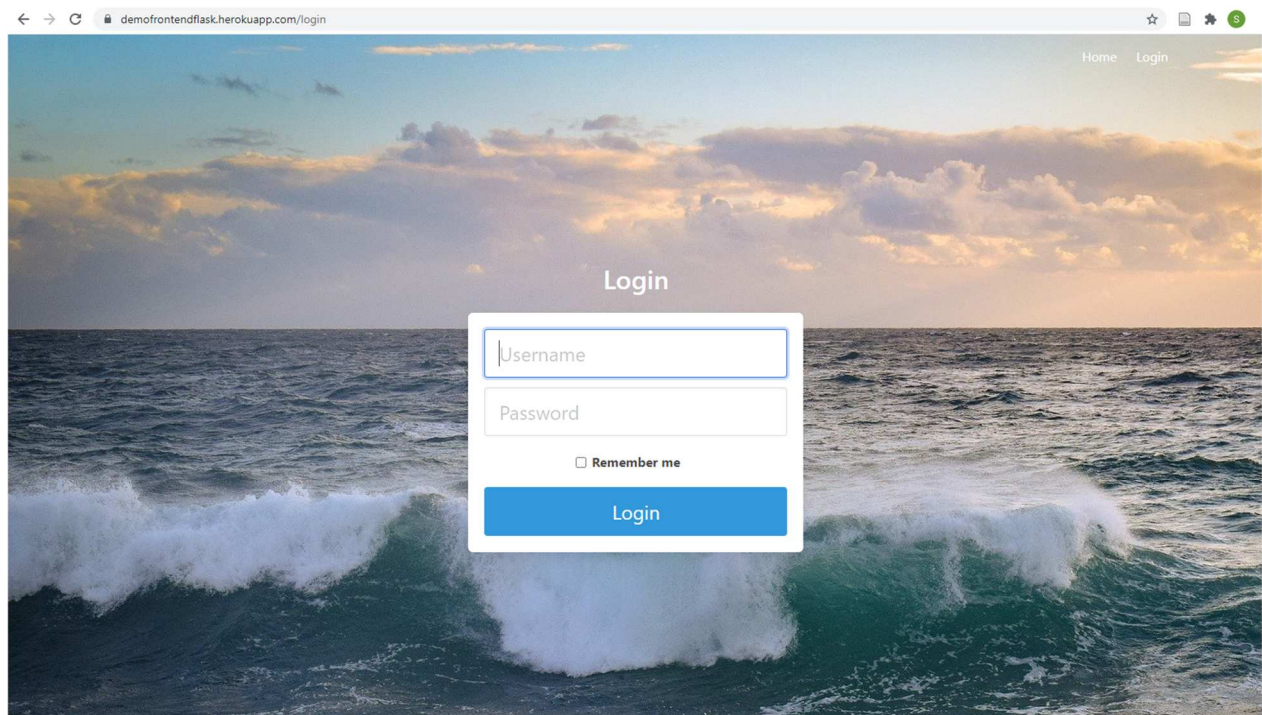
Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

[Next](#) ➔

© Copyright 2021, Mei, Lara and Sheherazade.
Built with Sphinx using a theme provided by Read the Docs.





Annexe 1 : Correspondances entre les étiquettes des variables du fichier RF.txt et les variables du code

KEY de l'application	Txt label
geonameid	geonameid
name	name
asciiname	asciiname
alternatenames	alternatenames
latitude	latitude
longitude	longitude
feature_class	feature class
feature_code	feature code
country_code	country code
cc2	cc2
admin1_code	admin1 code
admin2_code	admin2 code
admin3_code	admin3 code
admin4_code	admin4 code
population	population
elevation	elevation
dem	dem
timezone	timezone
modification	modification date