

PROJET NOSQL

06 Aout 2023

Enseignant : Jean-Marie PREIRA

*Joshua Juste Emmanuel Yun Pei
NIKIEMA*

Table des matières

Introduction	2
I. Choix et description du jeu de données.....	3
1. Choix du jeu données	3
2. Description du jeu de données.....	3
II. Choix du Framework.....	4
III. Description de la procédure déploiement de la base de données sur Mongo Atlas.....	6
Etape 1 : Création d'un compte utilisateur sur MongoDB Atlas.....	7
Etape 2 : Création d'un cluster partagé.....	8
Etape 3 : Création des utilisateurs du cluster.....	9
Etape 4 : Configuration des adresses ip qui peuvent accéder au cluster	9
Etape 5 : Création de la base de données et de la collection	10
Etape 6 : Choix de la méthode de connexion.....	11
Etape 7 : Connexion à MongoDB Compass	13
Etape 8 : Importation des données	13
IV. Présentation de l'environnement de travail	14
V. Liste des requêtes d'interrogation simples et analytiques proposées dans l'API.....	15
1. Description des méthodes	15
2. Liste des requêtes de notre API.....	15
VI. Présentation et explication des codes source écrits.....	16
1. La partie connexion à la base de données stockées sur MongoDB Atlas	16
2. Définition du model de nos données	17
3. Partie pour la création de certaines fonctions pour effectuer les requêtes MongoDB	17
4. La partie de la création des Endpoints de notre API.....	18
a. La partie des 'importations'	19
b. Crédit de l'objet pour les Endpoints	19
c. Crédit de Endpoints	19
5. La partie exécution de l'application	32
VII. La documentation de l'API.....	32
VIII. Quelques captures d'écran présentant les résultats de tests obtenus	32
Conclusion	42

Introduction

Après avoir terminé le module sur les bases des bases de données NoSQL, nous avons eu le choix entre trois projets. Le premier consistait à développer une API RESTful connectée à MongoDB Atlas, le deuxième portait sur la création d'un tableau de bord en utilisant MongoDB Charts et le troisième portait sur la création d'un tableau de bord en utilisant NeoDash. Après une discussion, nous avons opté pour le premier projet, car bien que le concept d'API soit plus ou moins nouveau pour nous et un peu abstrait, il nous a semblé particulièrement intéressant à explorer.

Dans les sections suivantes de ce rapport, nous allons commencer par fournir des définitions pour certains concepts fondamentaux. Ensuite, nous détaillerons étape par étape le processus de création de notre cluster sur MongoDB Atlas et expliquerons comment nous avons chargé notre jeu de données. Par la suite, nous aborderons l'environnement de travail que nous avons mis en place et les différentes bibliothèques que nous avons utilisées pour développer l'API. Enfin, nous présenterons une liste des requêtes que nous avons développées pour l'API, couvrant à la fois les interrogations simples et les analyses plus poussées ainsi que certains tests.

I. Choix et description du jeu de données

1. Choix du jeu données

Pour notre projet nous avons choisi un ensemble de données sur la population mondiale EDA & MAP VISULZATION obtenu sur kaggle ; disponible à partir du lien suivant :

Format csv :

<https://www.kaggle.com/datasets/rajkumarpandey02/2023-world-population-by-country?select=countries-table.csv>

Format json :

<https://www.kaggle.com/datasets/rajkumarpandey02/2023-world-population-by-country?select=countries-table.json>

Comprendre les tendances de la population mondiale est crucial pour divers domaines tels que l'urbanisme, l'économie et la santé publique. En analysant cet ensemble de données, nous visons à mieux comprendre la dynamique de la population et à découvrir des modèles et des tendances intéressants.

Ce jeu de données utilisées fournit des informations complètes sur la population mondiale au cours d'une période donnée. Il couvre un large éventail de pays et comprend des variables telles que la taille de la population, le taux de croissance, le taux de fécondité, l'espérance de vie, etc.

2. Description du jeu de données

Notre jeu de données est composé de plusieurs documents dont les attributs (19) sont :

- '**country**' : Représente le nom du pays.
- '**rang**' : Indique le rang du pays en fonction d'un certain critère (par exemple, population, superficie, etc.).
- '**area**' : Indique la superficie totale du pays en kilomètres carrés.
- '**landAreaKm**' : Représente la superficie du pays en kilomètres carrés.
- '**cca2**' : Représente le code de pays à deux lettres attribuées par l'Organisation internationale de normalisation (ISO 3166-1 alpha-2).
- '**cca3**' : Représente le code de pays à trois lettres attribuées par l'Organisation internationale de normalisation (ISO 3166-1 alpha-3).
- '**netChange**' : Indique la variation nette de la population pour une période donnée (par exemple, variation nette annuelle).
- '**growthRate**' : Représente le taux de croissance de la population, généralement exprimé en pourcentage.
- '**worldPercentage**' : Indique le pourcentage de la population mondiale que le pays représente.
- '**density**' : Représente la densité de la population du pays, généralement mesurée en nombre de personnes par kilomètre carré.

- '**densityMi**' : Représente la densité de la population du pays, mesurée en nombre de personnes par mille carrés.
- '**place**' : indique la situation géographique ou la position du pays (par exemple, continent, région, etc.).
- '**pop1980**' : Représente l'estimation de la population pour l'année 1980
- '**pop2000**' : Représente l'estimation de la population pour l'année 2000.
- '**pop2010**' : Représente l'estimation de la population pour l'année 2010
- '**pop2022**' : Représente l'estimation de la population pour l'année 2022
- '**pop2023**' : Représente l'estimation de la population pour l'année 2023.
- '**pop2030**' : Représente l'estimation de la population projetée pour l'année 2030
- '**pop2050**' : Représente l'estimation de la population projetée pour l'année 2050.

II. Choix du Framework

Dans l'univers du développement logiciel, foisonnent une multitude de cadres conceptuels s'appuyant sur une pléthore de langages de programmation. Parmi ces joyaux technologiques, nous trouvons d'éminents noms tels que le vénérable Spring de Java, l'élégant Express.js propulsé par Node.js, l'incontournable Ruby on Rails pour les passionnés de Ruby, sans oublier les remarquables Flask, FastAPI et Django du royaume Python.

C'est au sein de ce panorama éblouissant que notre chronique prendra place, plongeant avec audace dans l'univers de Python et embrassant la prestigieuse librairie FastAPI comme précieux allié dans cette quête de l'excellence.

Pourquoi FastAPI ?

FastAPI : L'étoile montante des Frameworks Python pour le développement d'APIs

Lorsqu'il s'agit de choisir le cadre conceptuel idéal pour la création d'applications web et d'interfaces de programmation (APIs RestFul), FastAPI émerge comme une étoile montante incontestable dans l'univers foisonnant des Frameworks Python. Cette pépite technologique a su conquérir les cœurs des développeurs grâce à une panoplie de caractéristiques remarquables qui en font un choix judicieux et éclairé.

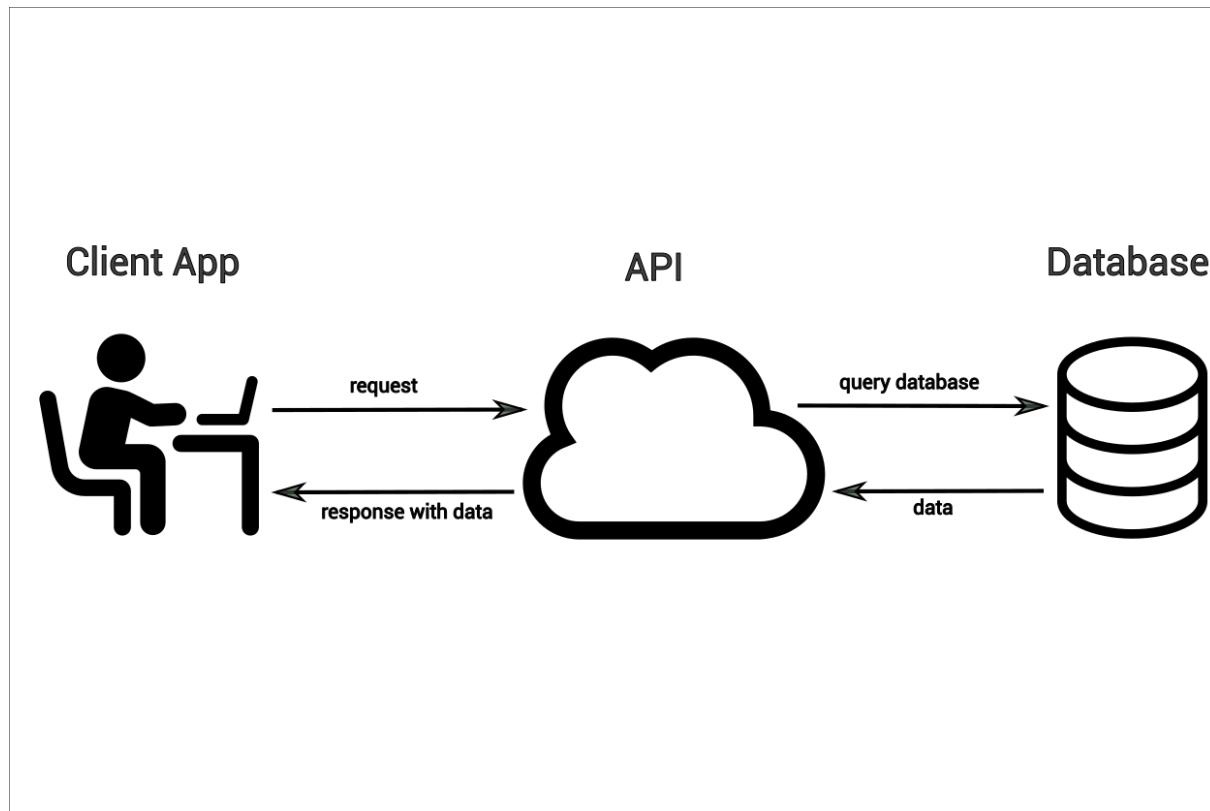
Comme avantages :

- Très facile à utiliser
- Haute performance et rapidité.
- Documentation automatique et interactive.
- Intégration fluide avec l'écosystème Python.
- Sécurité renforcée grâce à la validation de types.
- Communauté active et en constante évolution.

Qu'est-ce qu'une API RESTFUL ?

Une API, qui signifie "Application Programming Interface", représente un ensemble de fonctions informatiques qui permettent à deux logiciels de communiquer directement, sans intervention humaine. Une API peut être soit publique, accessible à tous, soit privée, réservée à certains utilisateurs. Son rôle est de fournir des services et de mettre à disposition des fonctionnalités ou des données.

En parallèle, les développeurs créent des programmes qui utilisent ces APIs pour interagir avec les services qu'elles exposent. Pour ce faire, ils se réfèrent à la documentation des APIs, qui leur fournit toutes les informations nécessaires pour utiliser efficacement ces interfaces de programmation.

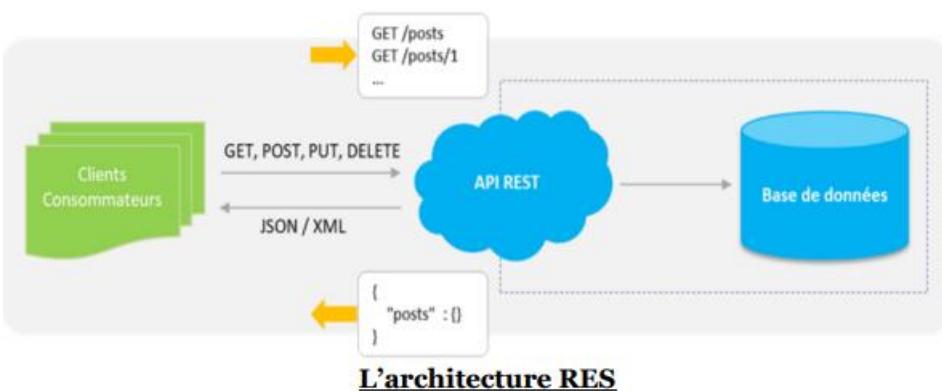


Une API REST, abréviation de "Representational State Transfer Application Program Interface", incarne un style architectural qui facilite la communication entre logiciels, que ce soit sur un réseau ou au sein d'un même appareil. Principalement destinées à la création de services web, ces APIs sont souvent qualifiées de "RESTful" et font usage des méthodes HTTP pour échanger des données entre un client et un serveur, même si ces derniers utilisent des systèmes d'exploitation et des architectures différentes.

Le client peut solliciter des ressources en utilisant un langage compréhensible par le serveur, qui renvoie ensuite la ressource dans un format accepté par le client, couramment JSON ou XML. Pour identifier ces ressources, REST se repose sur les URI (Uniform Resource Identifier).

Ces ressources représentent des informations variées telles que des images, des documents, des personnes, et bien d'autres encore. L'approche REST offre ainsi un mécanisme flexible et

uniforme pour interagir avec les services web, facilitant l'échange d'informations entre les parties prenantes.



III. Description de la procédure déploiement de la base de données sur Mongo Atlas

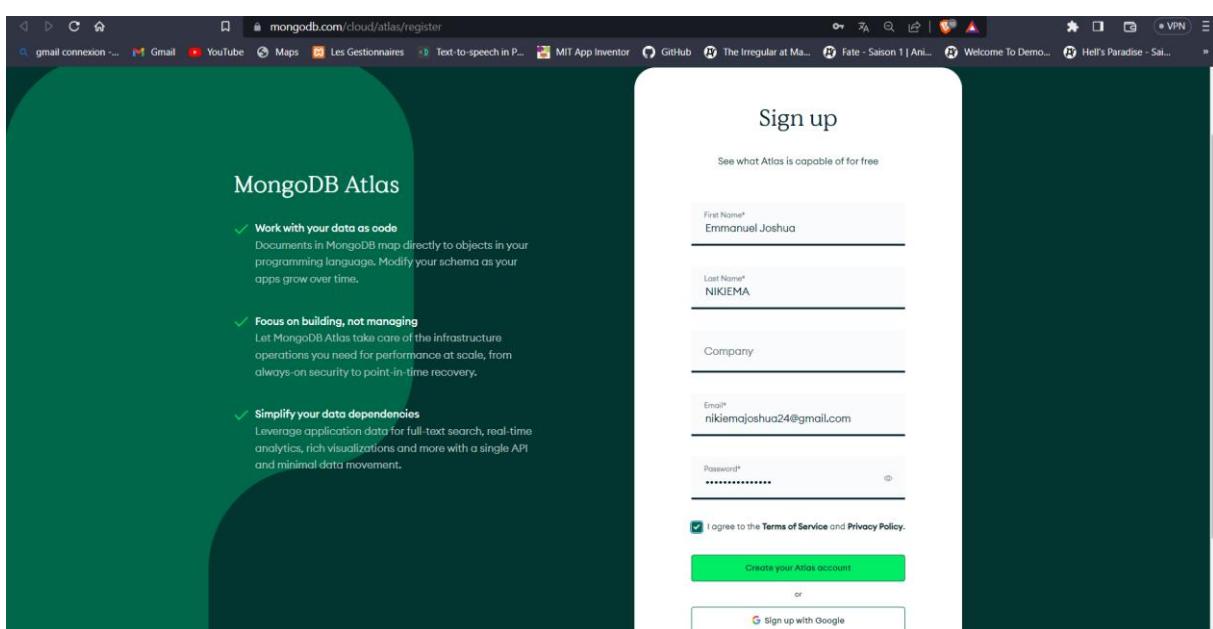
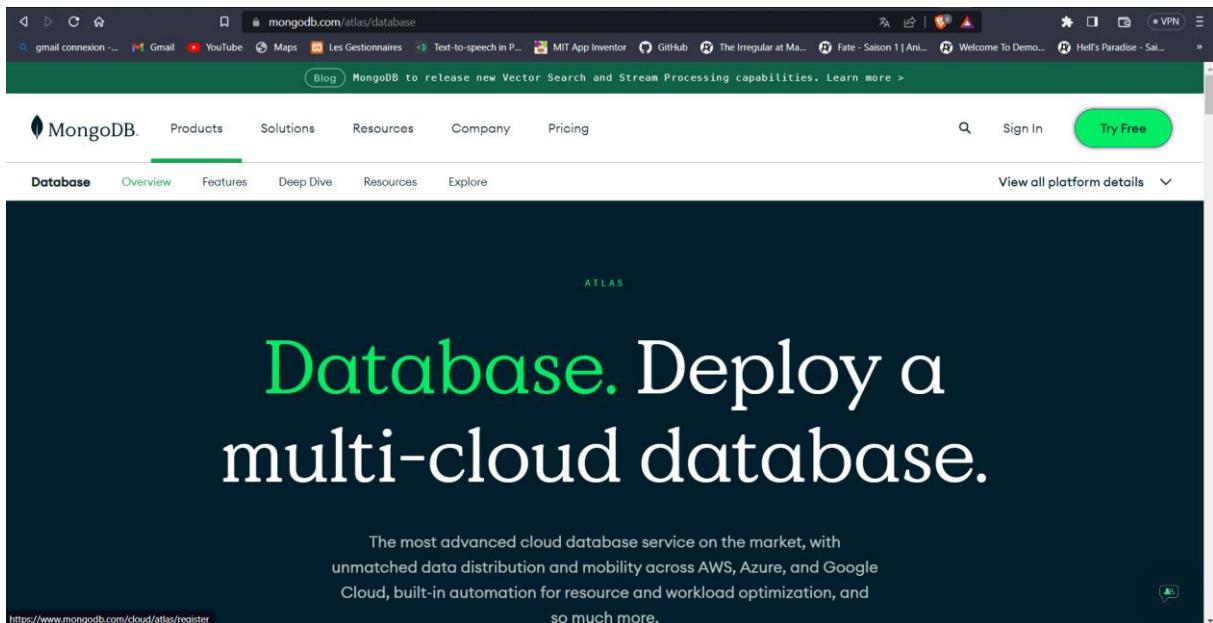
Dans le vaste cosmos de la gestion de données, une étoile scintillante brille avec éclat : MongoDB Atlas. Tel un gardien éclairé de l'univers des bases de données, MongoDB Atlas offre une solution infiniment puissante et flexible pour stocker, gérer et interagir avec les données de manière transparente. Tel un voyage interstellaire vers l'excellence, MongoDB Atlas propulse les développeurs et les entreprises vers de nouveaux horizons, en offrant une plateforme cloud entièrement gérée qui élimine les tracas liés à la mise en place et à la maintenance des infrastructures de bases de données. Cette technologie révolutionnaire permet aux esprits créatifs de se concentrer sur l'essentiel : développer des applications innovantes et révolutionnaires. Le véritable astrolabe de MongoDB Atlas réside dans son architecture distribuée, qui assure une évolutivité sans limite. Que ce soit pour une petite application stellaire ou pour une constellation d'applications à l'échelle planétaire, MongoDB Atlas s'adapte avec agilité, assurant une disponibilité sans faille et une performance sans précédent. Naviguant dans les océans de la sécurité informatique, MongoDB Atlas protège les données avec une vigilance sans égale. Grâce à des fonctionnalités de sécurité avancées et à une intégration transparente avec les principaux fournisseurs d'identité, chaque donnée est verrouillée en sécurité, comme des joyaux cachés au cœur d'une nébuleuse protégée. Grâce à toutes ces caractéristiques, MongoDB Atlas peut s'enorgueillir d'être l'élite des bases de données NoSQL.

Le déploiement de la base de données sur Mongo Atlas s'effectue suivant plusieurs étapes qui seront énoncées dans les lignes suivantes :

7

Etape 1 : Création d'un compte utilisateur sur MongoDB Atlas.

[Cliquez ici pour accéder au site.](#) Après cliquez sur Try Free ou sur Sign In si vous avez déjà un compte.



Suivez les instructions.

Etape 2 : Création d'un cluster partagé

- Choisir l'option FREE

The screenshot shows the MongoDB deployment options page. It features three main plan cards:

- M10 (\$0.08/hour)**: For production applications with sophisticated workload requirements. Storage: 10 GB, RAM: 2 GB, vCPU: 2 vCPUs.
- SERVERLESS (\$0.10/1M reads)**: For application development and testing, or workloads with variable traffic. Storage: Up to 1TB, RAM: Auto-scale, vCPU: Auto-scale.
- MO (FREE)**: For learning and exploring MongoDB in a cloud environment. Storage: 512 MB, RAM: Shared, vCPU: Shared.

- Choisir le fournisseur cloud, la région de stockage et le nom de votre cluster ; et enfin cliquez sur **Create**.

The screenshot shows the MongoDB cluster creation form for the AWS provider. The fields filled in are:

- Provider**: AWS (selected)
- Region**: N. Virginia (us-east-1) (selected)
- Name**: JoshuaCluster
- Tag (optional)**: Create your first tag to categorize and label your resources; more tags can be added later. (Leave empty)

At the bottom, the plan selected is **FREE**, and the **Create** button is highlighted.

Etape 3 : Création des utilisateurs du cluster

The screenshot shows the MongoDB Atlas interface under the 'Data Services' tab. On the left, there's a sidebar with 'Project 0' and various service icons. Under 'SECURITY', 'Quickstart' is selected. The main area is titled 'Security Quickstart' and displays instructions for creating a user. It shows two options: 'Username and Password' (selected) and 'Certificate'. A note says 'We autogenerated a username and password for your first database user in this project using your MongoDB Cloud registration information.' Below this, there's a form to enter a 'Username' (nikiemajoshua24) and a 'Password'. There are buttons for 'Autogenerate Secure Password' and 'Copy'. At the bottom is a 'Create User' button.

Ces utilisateurs représentent les applications qui peuvent avoir accès au Cluster. Dans notre cas nous avons créé un utilisateur « nikiemajoshua24 » qui a le droit de lecture et d'écriture sur toutes les base de données du Cluster, après s'être authentifié par son mot de passe.

Database Access

The screenshot shows the 'Database Users' section of the MongoDB Atlas 'Database Access' page. It lists a single user: 'User Name': nikiemajoshua24, 'Authentication Method': SCRAM, and 'MongoDB Roles': readWriteAnyDatabase@admin. To the right, there are 'Resources' (All Resources), 'Actions' (EDIT, DELETE), and a '+ ADD NEW DATABASE USER' button.

Etape 4 : Configuration des adresses ip qui peuvent accéder au cluster

The screenshot shows the 'IP Access List' configuration page. It has two sections: 'My Local Environment' (with a note about adding network IP addresses to the IP Access List) and 'Cloud Environment' (with a note about configuring network access between Atlas and a cloud or on-premise environment). A message at the top says 'We added your current IP address. You can connect to your cluster locally from this device.' Below this, there's a section to 'Add entries to your IP Access List'. It says 'Only an IP address you add to your Access List will be able to connect to your project's clusters. You can manage existing IP entries via the Network Access Page.' There are fields for 'IP Address' (Enter IP Address) and 'Description' (Enter description), and a 'Add My Current IP Address' button. A note below says 'This IP address has already been added.' At the bottom, it shows the IP Access List entry: 'IP Access List': 41.83.78.55/32, 'Description': My IP Address, with 'EDIT' and 'REMOVE' buttons. At the very bottom is a 'Finish and Close' button.

10

Etape 5 : Création de la base de données et de la collection

On se trouve au niveau de l'interface de notre cluster. Cliquez sur **Browse Collections**

The screenshot shows the MongoDB Atlas interface for a cluster named 'JoshuaCluster'. On the left, there's a sidebar with options like Deployment, Services, Security, and New On Atlas. The main area is titled 'Database Deployments' and shows a message to 'Load sample datasets to JoshuaCluster'. Below this, there are tabs for 'Connect', 'View Monitoring', and 'Browse Collections', with 'Browse Collections' being highlighted with a red box. The interface displays various metrics such as connections, bandwidth usage, and data size. At the bottom, it shows the cluster details: VERSION 6.0.8, REGION AWS / N. Virginia (us-east-1), CLUSTER TIER M0 Sandbox (General), TYPE Replica Set - 3 nodes, BACKUPS Inactive, LINKED APP SERVICES None Linked, ATLAS SQL Connect, and ATLAS SEARCH Create Index.

Si le cluster ne possède pas de base de données vous aurez ceci, donc cliquez sur **Add My Own Data** :

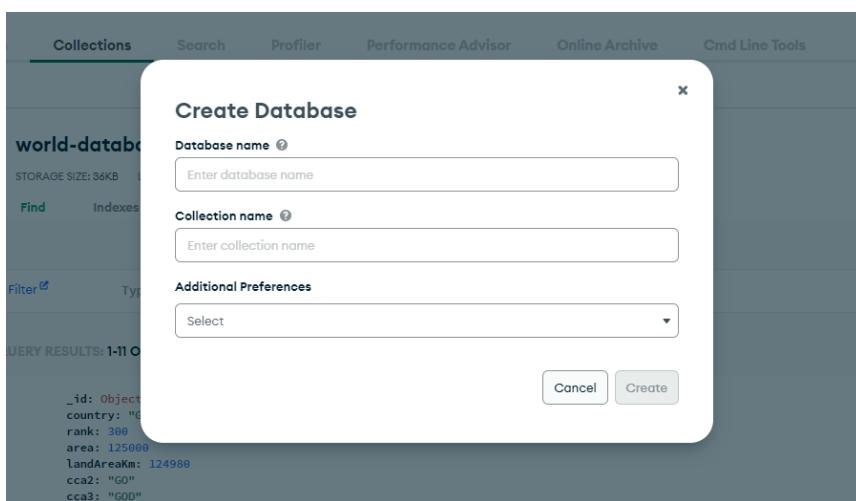
The screenshot shows the MongoDB Atlas interface for a cluster named 'JoshuaCluster'. The top navigation bar includes Overview, Real Time, Metrics, Collections (which is selected and highlighted with a red box), Search, Profiler, Performance Advisor, Online Archive, and Cmd Line Tools. The cluster details are shown as VERSION 6.0.8 and REGION AWS N. Virginia (us-east-1). The main area features a 'Explore Your Data' section with a magnifying glass icon and a list of actions: Find, Indexes, Aggregation, and Search. Below this are buttons for 'Load a Sample Dataset' and 'Add My Own Data', with 'Add My Own Data' being highlighted with a red box. A link to 'Learn more in Docs and Tutorials' is also present.

Si votre cluster possède déjà une base de données.

Cliquer sur **+ Create Database**

The screenshot shows the MongoDB Atlas interface for managing databases. It displays 'DATABASES: 1' and 'COLLECTIONS: 1'. Below this is a button labeled '+ Create Database' which is highlighted with a red box. There is also a search bar labeled 'Search Namespaces'.

Et remplir les champs



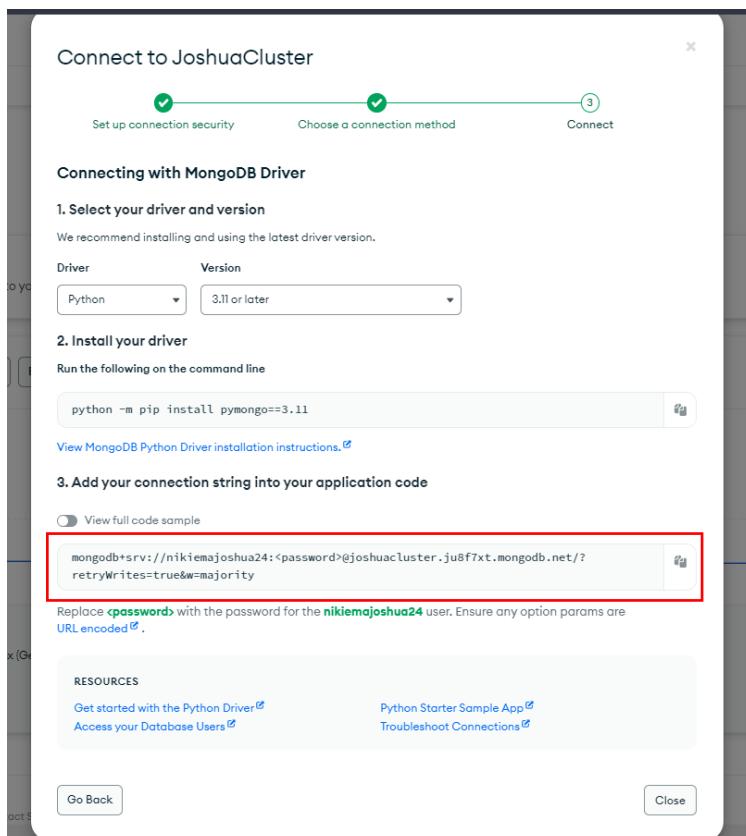
Etape 6 : Choix de la méthode de connexion

- Cliquez sur Connect.

- Cliquez sur Drivers

12

- Choisir le langage Python et sa version et enfin copier le code encerclé. Ce code sera utilisé au niveau de notre programme python pour se connecter au cluster.



On peut voir un exemple de code illustrant l'utilisation du code en activant **View full code sample**.

[View MongoDB Python Driver installation instructions.](#)

3. Add your connection string into your application code

View full code sample

```
from pymongo.mongo_client import MongoClient

uri = "mongodb+srv://nikiemajoshua24:<password>joshuacluster.ju8f7xt.mongodb.net/?retry"

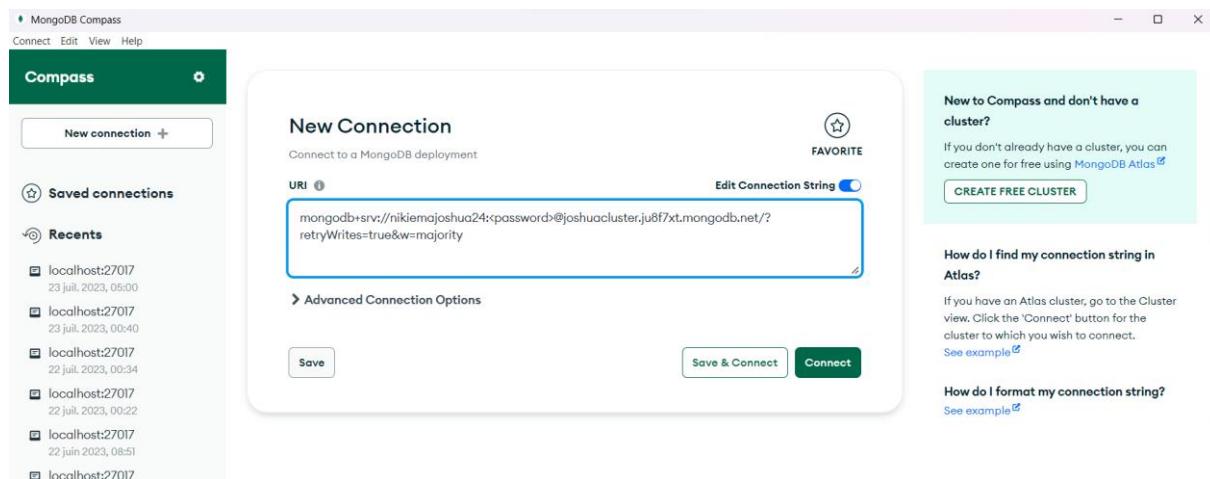
# Create a new client and connect to the server
client = MongoClient(uri)

# Send a ping to confirm a successful connection
try:
    client.admin.command('ping')
    print("Pinged your deployment. You successfully connected to MongoDB!")
except Exception as e:
    print(e)
```

Replace <password> with the password for the **nikiemajoshua24** user. Ensure any option params are URL encoded.

Etape 7 : Connexion à MongoDB Compass

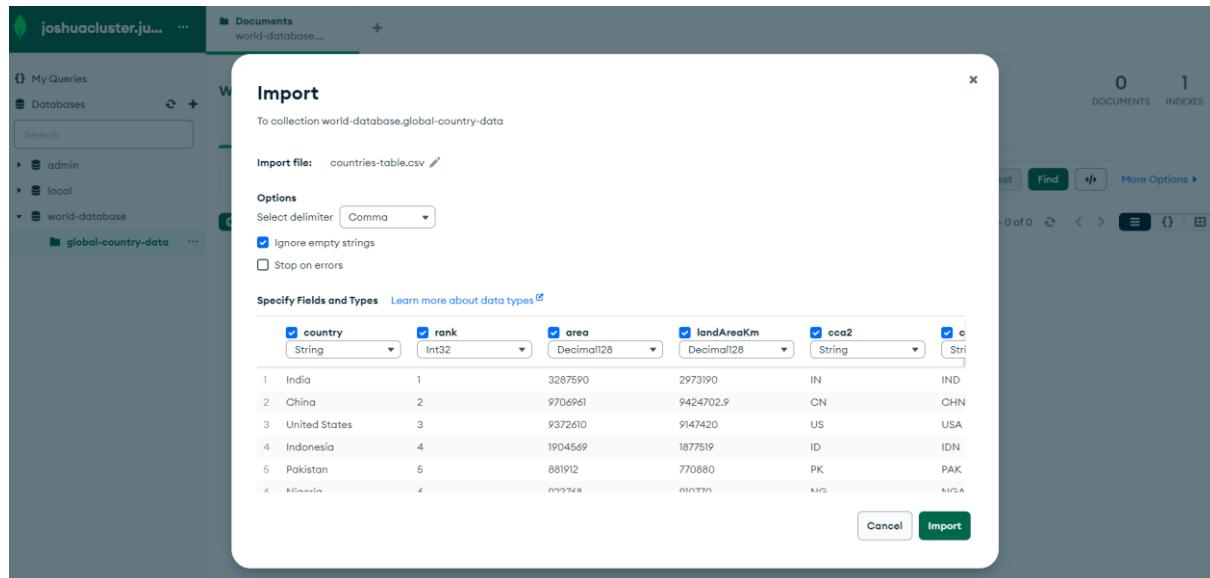
Nous avons la possibilité de connecter notre base de données à mongoDB Compass en entrant l'URI de notre cluster ;



Et donc à partir de MongoDB Compass nous pouvons créer la base de données et la collection et ensuite importer nos données.

Etape 8 : Importation des données

A travers **MongoDB Compass** nous allons importer nos données dans notre base de données créée sur **MongoDB Atlas**.

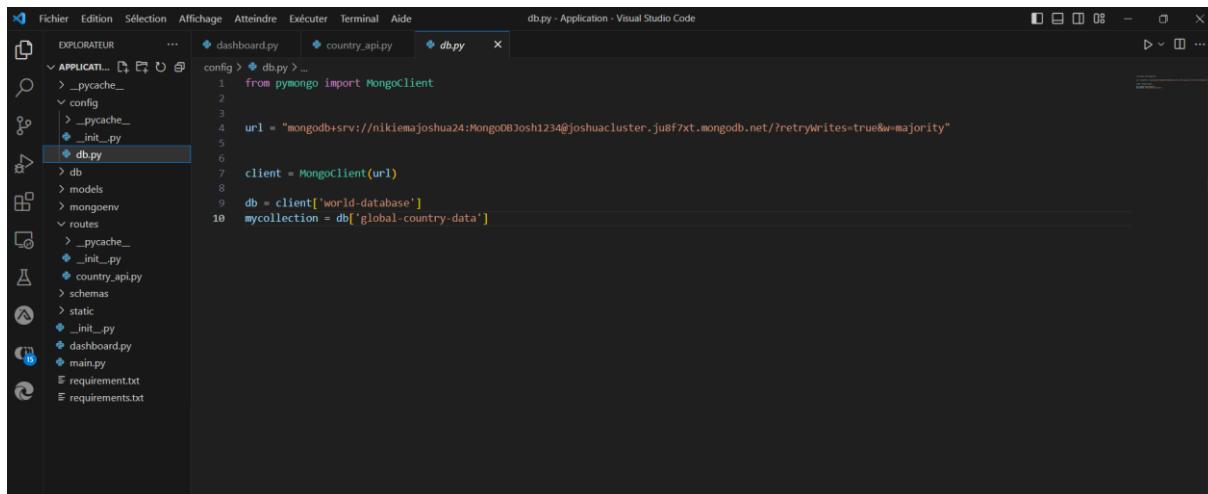


En allant sur MongoDB Atlas nous pouvons constater l'importation à belle et bien eu lieu.

IV. Présentation de l'environnement de travail

Visual Studio Code (VS Code) est un éditeur de code moderne, léger et polyvalent, développé par Microsoft. C'est un outil puissant et largement apprécié par les développeurs du monde entier. Grâce à ses fonctionnalités avancées, sa facilité d'utilisation et son écosystème florissant d'extensions, VS Code est devenu l'un des éditeurs de code les plus populaires et les plus appréciés de l'industrie du développement logiciel.

Cliquez ici pour télécharger [VS Code](#).



Présentation de l'environnement de travail VS Code

Bibliothèques utilisées

Pour la création de notre **API Rest** nous avons installé certaines bibliothèques en utilisant la commande « **pip install nom_bibliothèque** » :

Fastapi, uvicorn, streamlit, pandas, pymongo, pipreqs, pymongo[tls,srv], lxml, nltk, pdfplumber, rake_nltk, PyPDF2, pydantic, matplotlib, seaborn, plotly, missingno, streamlit_option_menu, numerize, gunicorn.

La classe Python **MongoClient** permet aux développeurs d'établir des connexions à MongoDB en développement à l'aide d'instances clientes, elle facilite le codage et la connexion à MongoDB. **MongoClient** fait partie de la bibliothèque **Pymongo** et peut être importé dans le code Python en utilisant le code « **from pymongo import MongoClient** ».

V. Liste des requêtes d'interrogation simples et analytiques proposées dans l'API

- La méthodologie la plus répandue de conception des API s'appelle REST.
- L'aspect le plus important de REST est qu'elle est basée sur quatre méthodes définies par le protocole HTTP : **GET, POST, PUT et DELETE**.
- Celles-ci correspondent aux quatre opérations standard effectuées sur une base de données : **READ, CREATE, UPDATE et DELETE**.

1. Description des méthodes

Les méthodes de ressources correspondent aux différents types de traitements que nous pouvons effectuer lors d'une requête API REST. Il existe principalement 4 méthodes courantes, qui correspondent aux méthodes CRUD, à savoir :

POST : la méthode qui sert à publier, envoyer des informations au serveur.

GET : celle pour récupérer des informations venant du serveur. Elle peut retourner plusieurs formats de réponses, comme nous l'avons souvent répété au cours de cette chronique.

PUT : il s'agit de la méthode à utiliser lorsque l'on effectue une mise à jour auprès d'un serveur.

DELETE : qui, comme son nom l'indique permet de supprimer une entrée sur un serveur.

Cependant, il existe d'autres méthodes que l'on peut utiliser lors d'un appel d'une API REST comme PATCH, HEAD (l'équivalent du "Test d'existence"), OPTIONS (l'équivalent de "Lister les commandes disponibles"), TRACE et CONNECT.

2. Liste des requêtes de notre API

Voici donc les requêtes http de notre API Rest.

<https://josh-mongodb-api.onrender.com/>

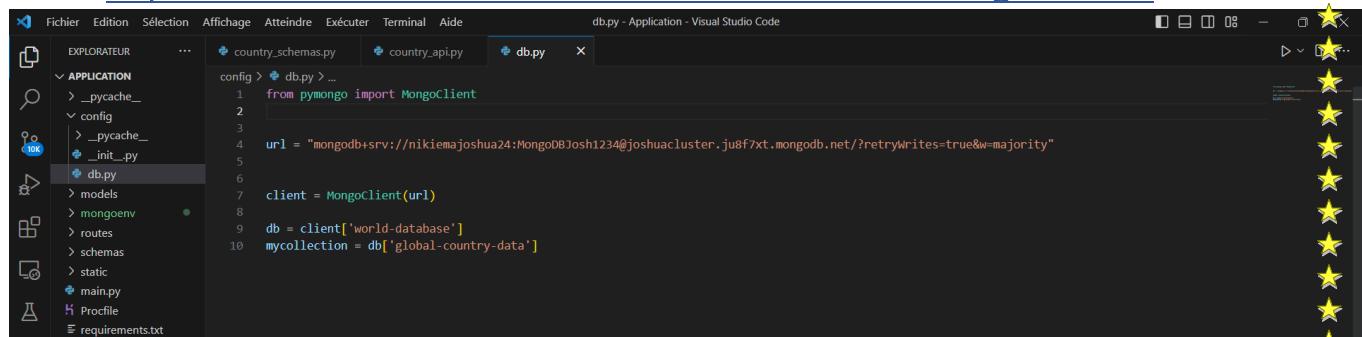
https://josh-mongodb-api.onrender.com/insert_country/
https://josh-mongodb-api.onrender.com/update_country/{id}
https://josh-mongodb-api.onrender.com/delete_country/{id}
https://josh-mongodb-api.onrender.com/countries_info/
<https://josh-mongodb-api.onrender.com/country/{name}>
https://josh-mongodb-api.onrender.com/countries_density/{D1}/{D2}
https://josh-mongodb-api.onrender.com/country_most_populated/{year}
https://josh-mongodb-api.onrender.com/country_least_populated/{year}
https://josh-mongodb-api.onrender.com/average_pop/
https://josh-mongodb-api.onrender.com/countries_pop/
https://josh-mongodb-api.onrender.com/countries_areas_sup1_sup2/{sup1}/{sup2}
https://josh-mongodb-api.onrender.com/custom_aggregation/{query}
https://josh-mongodb-api.onrender.com/custom_find/{query}
https://josh-mongodb-api.onrender.com/custom_distinct/{query}
https://josh-mongodb-api.onrender.com/nb_countries_supavg/{year}
https://josh-mongodb-api.onrender.com/nb_countries_infavg/{year}

NB : Le lien <https://josh-mongodb-api.onrender.com> parce que nous avons déployé notre API en ligne sur le cloud Render pour apprendre plus cliquez sur ce [lien](#).

VI. Présentation et explication des codes source écrits

Dans cette partie, nous vous présenterons le code source de notre API écrit en python avec l'aide du Framework FastAPI.

1. La partie connexion à la base de données stockées sur MongoDB Atlas



```

Fichier Edition Sélection Affichage Atteindre Exécuter Terminal Aide
db.py - Application - Visual Studio Code
EXPLORATEUR ...
APPLICATION
> __pycache__
> config
> __pycache__
> __init__.py
> db.py
> models
> mongoenv
> routes
> schemas
> static
> main.py
> Profile
requirements.txt

config > db.py > ...
1  from pymongo import MongoClient
2
3
4  url = "mongodb+srv://nikiemajoshua24:MongoDBJosh1234@joshuocluster.ju8f7xt.mongodb.net/?retryWrites=true&w=majority"
5
6
7  client = MongoClient(url)
8
9  db = client['world-database']
10 mycollection = db['global-country-data']

```

Dans cette partie, nous avons d'abord importé la méthode `MongoClient` de la bibliothèque `pymongo` et ensuite récupéré l'url de l'atlas MongoDB dans la variable `url`; cette variable nous la mettons dans `MongoClient()`.

Sur la ligne 9 il s'agit du lien vers notre base de données '`'world-database'`' et sur la ligne 10 nous avons la connexion à notre collection.

Cette partie nous l'avons fait dans le fichier « `./config/db.py` » pour des raisons de bonnes pratiques dans la programmation.

2. Définition du modèle de nos données

```

EXPLORATEUR APPLICATION
country_schemas.py country_api.py country.py
models > country.py > Country
    from pydantic import BaseModel
    ...
    Test this class
    class Country(BaseModel): # définition de notre modèle de donnée
        country: str
        rank: int
        area: float
        landArea: float
        cca2: str
        cca3: str
        netChange: float
        growthRate: float
        worldPercentage: float
        density: float
        densityM: float
        place: int
        pop1980: int
        pop2000: int
        pop2010: int
        pop2022: int
        pop2023: int
        pop2030: int
        pop2050: int
    ...

```

Grâce à sa nous pourrons créer et manipuler facilement des objets de type « `country` » qui représentent les documents de notre collection.

3. Partie pour la création de certaines fonctions pour effectuer les requêtes MongoDB

```

Fichier Edition Sélection Affichage Atteindre Exécuter Terminal Aide country_schemas.py - Application - Visual Studio Code
EXPLORATEUR APPLICATION
country_schemas.py country_api.py
schemas > country_schemas.py > getCountryDensityD1D2
    df = pd.DataFrame([{"Population in 1980": [pop1980],
                        "Population in 2010": [pop2010],
                        "Population in 2023": [pop2023],
                        "Population in 2050": [pop2050]})

    return df
else:
    # Le pays n'a pas été trouvé dans la collection
    print("Country not found in the database.")
    return None
# Fonction pour récupérer les pays dont la densité est comprise entre d1 et d2 et les mettre dans un DataFrame
def getCountryDensityD1D2(d1, d2, mycollection):
    d1 = float(d1)
    d2 = float(d2)
    # On récupère les pays
    countries = mycollection.find({"density": {"$gte": d1, "$lte": d2}})
    # On récupère les données de densité de population
    data = []
    for country in countries:
        country_data = {
            "Country": country["country"],
            "Density": country.get("density")
        }
        data.append(country_data)
    # On met les données dans un DataFrame
    df = pd.DataFrame(data)
    return df

```

Pour une bonne pratique de programmation dans le fichier '/schemas/country_schemas.py' nous avons implémenter certaines fonctions grâce auxquelles des requêtes MongoDB pourront être exécutées.

Par exemple: `def getCountriesDensityD1D2(d1, d2, mycollection)`, est la fonction qui permettra à l'utilisateur de rechercher les pays qui ont une densité comprise entre d1 et d2 ; elle utilise la requête `find()` de MongoDB.

Il y a bien d'autres fonctions :

```
# Fonction de requêtes pour extraire un pays et la population de 1980, 2010, 2023 et 2050 et les mettre dans un dataframe
Test this function
def getCountryPop(countryName, mycollection): ...

# Fonction pour recuperer les pays dont la densité est compris entre d1 et d2 et les mettre dans un dataframe
Test this function
def getCountryDensityD1D2(d1, d2, mycollection): ...

# Obtenir la population moyenne du monde en 1980, 2000, 2010, 2023, 2030 et 2050 en utilisant une requête d'agrégation
Test this function
def getWorldAveragePop(mycollection): ...

# Ré recuperer toutes les données de la collection
Test this function
def getAllCountries(mycollection): ...

# Ré recuperer une ligne spécifique de la collection en fonction du nom du pays
Test this function
def getCountry(countryName, mycollection): ...

# Trouver le pays le moins peuplé en fonction de l'année
Test this function
def getLeastPopulatedCountry(year, mycollection): ...

# Trouver le pays le plus peuplé en fonction de l'année
Test this function
def getMostPopulatedCountry(year, mycollection): ...

# Ré recuperer le paays dont la superficie est comprise entre sup1 et sup2
Test this function
def getCountryAreasSup1Sup2(sup1, sup2, mycollection): ...

# Fonction de requêtes pour extraire la population de 1980, 2000, 2010, 2023 et 2050 des pays grace une requête d'agrégation
Test this function
def getCountryPop(mycollection): ...

# Fonction pour recevoir n'importe quelle requête d'aggrégation et retourner le résultat dans un dataframe
Test this function
> def getAggregationRequest(aggregation, mycollection): ...

# Fonction pour recevoir n'importe quelle requête "find" et retourner le résultat dans un dataframe
Test this function
> def getFindRequest(query, mycollection): ...

# Fonction pour recevoir n'importe quelle requête 'distinct' et retourner le résultat dans un dataframe
Test this function
> def getDistinctRequest(distinct, mycollection): ...
```

4. La partie de la création des Endpoints de notre API

D'abord, un "Endpoint" dans une API (Application Programming Interface) est une URL spécifique qui représente une ressource ou une fonctionnalité exposée par l'API. Il agit comme un point d'accès permettant aux clients (applications ou services) d'interagir avec le serveur pour effectuer des opérations spécifiques.

Nos Endpoints, nous les avons codé dans le fichier '/routes/country_api.py'

a. La partie des 'importations'

```
country_api.py
routes > country_api.py > ...
1  from fastapi import APIRouter, HTTPException
2  import json
3  from fastapi.responses import JSONResponse
4  from models.country import Country
5  from config.db import mycollection
6  from bson.objectid import ObjectId
7  from starlette import responses as _responses
8  from schemas.country_schemas import *
9
```

APIRouter ➔ Utilisé pour instancier l'objet qui permettra de créer les Endpoints de notre api.

HTTPException ➔ Pour lever des exceptions HTTP

json ➔ Pour manipuler des objets JSON

JSONResponse ➔ Utilisé pour permettre aux Endpoints d'envoyer les données dans le format JSON.

Country ➔ La classe représentant le modèle des documents de notre collection.

mycollection ➔ Représente notre collection

ObjectId ➔ Pour récupérer l'Id de nos documents.

_responses ➔ méthode pour permettre de faire des redirections vers une page spécifique.

from schemas.country_schemas import * ➔ Pour importer toutes les fonctions contenues dans 'country_schemas.py'

b. Création de l'objet pour les Endpoints

```
L0
L1  country_api = APIRouter()
L2
```

c. Création des Endpoints

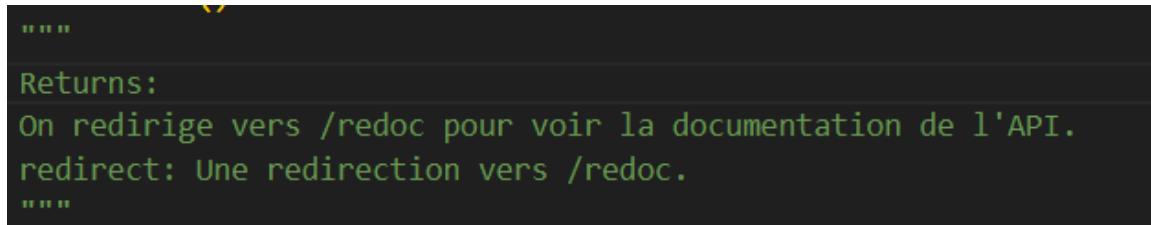
- La racine de notre API

```
Test this function
@country_api.get("/", tags=["Countries"])
async def home():
    """
    Returns:
        On redirige vers /redoc pour voir la documentation de l'API.
        redirect: Une redirection vers /redoc.
    """
    return _responses.RedirectResponse(url='/redoc')
```

Lorsque l'API est lancée, sans nous redirige vers /redoc pour voir la documentation de notre API.

20

Vu que nous avons ajouté la docstring, les clients pourront avoir une idée précise de ce que fait chaque Endpoint.



Docstring

The screenshot shows the Redoc API documentation for a 'Countries' endpoint. On the left, there's a sidebar with various endpoints like 'Home', 'Insert Country', 'Update Country', etc. The main area shows the 'Home' endpoint details:

Home
Returns: On redirige vers /redoc pour voir la documentation de l'API. redirect: Une redirection vers /redoc.

Responses
> 200 Successful Response

Insert Country
Insère un nouveau document de pays dans la collection MongoDB.
Parameters:

- country (Country): Un objet Country contenant les informations du pays à insérer.

Returns: dict: Un dictionnaire contenant un message de confirmation et les données insérées.

Request samples
Payload
Content type application/json

La page redoc

• La partie concernant la méthode POST

Cette méthode sert à publier, envoyer ou créer de nouveaux documents dans la collection de notre base de données.

```
27 @country_api.post("/insert_country/", response_model=dict, tags=["Countries"])
28 async def insert_country(country: Country):
29     """
30         Insère un nouveau document de pays dans la collection MongoDB.
31
32     Parameters:
33         - country (Country): Un objet Country contenant les informations du pays à insérer.
34
35     Returns:
36         dict: Un dictionnaire contenant un message de confirmation et les données insérées.
37     """
38     country_dict = dict(country)
39     mycollection.insert_one(country_dict)
40
41     return {"message": "Country inserted successfully", "data": country_dict}
42
```

```
@country_api.post("/insert_country/", response_model=dict, tags=["Countries"])
```

Permet grâce à `@country_api.post()` de dire qu'il s'agit d'une méthode POST.

`"/insert_country/"` défini l'url qui sera chargé d'acheminer les données. `response_model=dict` est optionnel, mais il permet de définir le format d'envoi de la donnée ; ici on veut que ça soit un **dictionnaire**. `tags=["Countries"]` permet de grouper les docstrings des Endpoints.

`async def insert_country(country: Country):` avec cette fonction qu'on exécutera les opérations d'inscriptions dans la collection. `'async'` pour signifier que la fonction est asynchrone. L'utilisation de l'asynchronisme permet d'exécuter certaines parties du code de manière non bloquante, ce qui peut être particulièrement utile lors d'opérations longues ou d'appels à des API externes. `country: Country` pour que le client saisisse les champs du document (`country`) qui sera envoyé grâce à la méthode POST et sera converti en dictionnaire avant d'être insérer dans notre collection grâce à la requête `'insert_one()'` de MongoDB.

- [La partie concernant la méthode PUT](#)

```
test this function
48 @country_api.put("/update_country/{id}", response_model=dict, tags=["Countries"])
49 async def update_country(id: str, country: Country):
50     """
51         Met à jour les informations d'un pays dans la collection MongoDB.
52
53         Parameters:
54             - id (str): L'ID du document de pays à mettre à jour.
55             - country (Country): Un objet Country contenant les nouvelles informations du pays.
56
57         Returns:
58             dict: Un dictionnaire contenant un message de confirmation et les données mises à jour.
59     """
60     country_dict = dict(country)
61     updated_country = updateCountry(id, country_dict, mycollection)
62     if updated_country is not None:
63         return {"message": "Country updated successfully", "data": country_dict}
64     else:
65         raise HTTPException(status_code=404, detail="Country not found")
66
```

`@country_api.put("/update_country/", response_model=dict, tags=["Countries"])`

Permet grâce à `@country_api.put()` de dire qu'il s'agit d'une méthode PUT. Cette méthode permet au client de modifier des informations d'un document (ici `country`) de notre collection. Pour cela, il faut spécifier d'abord l'Id du document et les nouvelles données. Et dans la fonction `updateCountry(id, country_dict, mycollection)` se trouve la requête pour faire la mise à jour.

Après la modification, un message est envoyé soit la mise à jour s'est bien passée et il envoie le document mise à jour (`country_dict`) ou il n'a pas trouvé le document à modifier.

```
test this function
41 def updateCountry(id, newCountryData, mycollection):
42
43     for key, value in newCountryData.items():
44         if (key == "country" and value != "") or (key == "rank" and value != 0) \
45             or (key == "area" and value != 0) or (key == "landAreaKm" and value != 0) \
46             or (key == "netChange" and value != 0) or (key == "growthRate" and value != 0) \
47             or (key == "worldPercentage" and value != 0) or (key == "density" and value != 0) \
48             or (key == "densityMl" and value != 0) or (key == "place" and value != 0) \
49             or (key == "pop1990" and value != 0) or (key == "pop2000" and value != 0) or (key == "pop2010" and value != 0) \
50             or (key == "pop2022" and value != 0) or (key == "pop2023" and value != 0) or (key == "pop2030" and value != 0) \
51             or (key == "pop2050" and value != 0) or (key == "cca2" and value != "") or (key == "cca3" and value != ""):
52
53         upt_co = mycollection.find_one_and_update({"_id": ObjectId(id), {"$set": {str(key): value}}})
54
55     return upt_co
```

La requête MongoDB utilisée est `find_one_and_update()` et grâce à l'id on trouve le document à modifier et avec l'option "`$set`" on modifie l'information qui a changé.

- **La partie concernant la méthode DELETE**

```
test this function
71 @country_api.delete("/delete_country/{id}", response_model=dict, tags=["Countries"])
72     async def delete_country(id: str):
73         """
74             Supprime un document de pays de la collection MongoDB.
75
76             Parameters:
77             - id (str): L'ID du document de pays à supprimer.
78
79             Returns:
80             dict: Un dictionnaire contenant un message de confirmation et les données supprimées.
81             """
82             deleted_country = mycollection.delete_one({"_id": ObjectId(id)})
83             if deleted_country.deleted_count == 1:
84                 return {"message": "Country deleted successfully", "data": serializeDict(deleted_country)}
85             else:
86                 raise HTTPException(status_code=404, detail="Country not found")
```

`@country_api.delete("/delete_country/", response_model=dict, tags=["Countries"])`

Permet grâce à `@country_api.delete()` de dire qu'il s'agit d'une méthode DELETE.

Cette méthode permet au client de supprimer un document (ici country) de notre collection. Pour cela, il faut spécifier d'abord l'Id du document. Et grâce la requête '`delete_one()`' de MongoDB qui à partir de l'Id va retrouver le document spécifié et ensuite le supprimer.

Après la modification, un message est envoyé soit la suppression s'est bien passée et il envoie le document supprimé (`deleted_country`) ou il n'a pas trouvé le document à modifier.

- **La partie concernant les méthodes GET**

Cette partie est la plus nombreuse car l'objectif principal de notre API Rest est de permettre au client récupérer ou visualiser certaines informations qui peuvent être extraite de notre base de données sur MongoDB Atlas. Pour cela les principaux Endpoints de notre API sont des méthodes de type **GET**.

Nous avons implémenté 13 Endpoints de type GET.

 [Endpoint 1](#)

```
# Documentation de l'endpoint get_countries
Test this function
@country_api.get("/countries_info/", response_model=dict, tags=["Countries"])
async def get_countries():
    """
    Récupère toutes les informations des pays depuis la collection MongoDB et les renvoie au format JSON.

    Returns:
    JSONResponse: Un objet JSONResponse contenant les données des pays au format JSON.
    """
    df = getAllCountries(mycollection)
    return JSONResponse(content=df.to_dict(orient="records"))
```

Dans cette partie, l'instruction indique à FastAPI que la fonction `get_countries()` correspond au chemin `'/countries_info/'`. Cette fonction a pour rôle d'aller sur MongoDB Atlas dans notre base de donnée pour récupérer tous les documents et les envoi au format JSON (`JSONResponse()`). La requête MongoDB utilisée se trouve dans la fonction `getAllCountries(mycollection)`.

```
Test this function
def getAllCountries(mycollection):
    # On récupère les pays
    countries = mycollection.find({})

    colonnes_a_convertir = ["area", "landAreaKm", "netChange", "growthRate", "worldPercentage", "density"]
    # On met les données dans un dataframe
    df = pd.DataFrame(serializerList(countries))
    df[colonnes_a_convertir] = df[colonnes_a_convertir].astype(str)
    return df
```

Cette fonction utilise `find({})` pour récupérer tous les documents de la collection. Pour des raisons de compatibilité nous avons converti certaines colonnes en `string`.

 [Endpoint 2](#)

```
Test this function
106 @country_api.get("/country/{name}", response_model=dict, tags=["Countries"])
107 async def get_country(name: str):
108     """
109     Récupère les informations d'un pays depuis la collection MongoDB et les renvoie au format JSON.
110
111     Parameters:
112     - name (str): Le nom du pays à récupérer.
113
114     Returns:
115     JSONResponse: Un objet JSONResponse contenant les données du pays au format JSON.
116     """
117     df = getCountry(name, mycollection)
118     if df is not None:
119         return JSONResponse(content=df.to_dict(orient="records"))
120     else:
121         raise HTTPException(status_code=404, detail="Country not found")
```

Dans cette partie, l'instruction indique à FastAPI que la fonction `get_country(name : str)` correspond au chemin `'/country/{name}'`. Cette fonction a pour rôle d'aller sur MongoDB Atlas dans notre base de donnée pour récupérer le document qui a pour nom de pays le contenu de la variable `name` qui sera fourni par l'utilisateur et l'envoi au format JSON (`JSONResponse()`). La requête MongoDB utilisée se trouve dans la fonction `getCountry(name, mycollection)`.

Au niveau du chemin, `{name}` signifie que l'Endpoint s'attend à recevoir une information de la part du client.

```
121 # Récuperer une ligne spécifique de la collection en fonction du nom du pays
Test this function
122 def getCountry(countryName, mycollection):
123     # On récupère le pays
124     country = mycollection.find_one({"country": countryName}, {"_id": 0})
125     if country:
126         # On met les données dans un dataframe
127         df = pd.DataFrame(serializedDict(country), index=[0])
128         return df
129     else:
130         return None
131
```

Cette fonction utilise `find_one({"country": countryName}, {"_id": 0})` pour récupérer un pays en fonction de son nom de la collection.

Endpoint 3

```
Test this function
125 @country_api.get("/countries_density/{D1}/{D2}", response_model=dict, tags=["Countries"])
126 async def get_countries_densityD1D2(D1: float, D2: float):
127     """
128     Récupère les informations de densité (density) de tous les pays dont la densité est comprise
129     entre D1 et D2 depuis la collection MongoDB et les renvoie au format JSON.
130
131     Parameters:
132     - D1 (float): La valeur minimale de densité.
133     - D2 (float): La valeur maximale de densité.
134
135     Returns:
136     JSONResponse: Un objet JSONResponse contenant les données des pays au format JSON.
137     """
138     df = getCountriesDensityD1D2(D1, D2, mycollection)
139     return JSONResponse(content=df.to_dict(orient="records"))
140
```

Dans cette partie, l'instruction indique à FastAPI que la fonction `get_countries_densityD1D2(name : str)` correspond au chemin `'/countries_density/{D1}/{D2}'`. Cette fonction a pour rôle d'aller sur MongoDB Atlas dans notre base de donnée pour récupérer tous les documents (pays) dont la densité est comprise entre 2 densités (D1 et D2) fourni par le client et l'envoi au format JSON (`JSONResponse()`). La requête MongoDB utilisée se trouve dans la fonction `getCountriesDensityD1D2(D1, D2, mycollection)`.

```
64 def getCountriesDensityD1D2(d1, d2, mycollection):
65     d1 = float(d1)
66     d2 = float(d2)
67     # On récupère les pays
68     countries = mycollection.find({"density": {"$gte": d1, "$lte": d2}})
69     # On récupère les données de densité de population
70     data = []
71     for country in countries:
72         country_data = {
73             "Country": country["country"],
74             "Density": country.get("density")
75         }
76         data.append(country_data)
77     # On met les données dans un dataframe
78     df = pd.DataFrame(data)
79     return df
```

Cette fonction utilise `find({"density": {"$gte": d1, "$lte": d2}})` pour trouver les pays concernés. Après les avoir trouvés on crée un dataframe avec les colonnes "country" et "Density" et on l'envoie.

Endpoint 4 :

```

143 @country_api.get("/country_most_populated/{year}", response_model=dict, tags=["Countries"])
144 async def get_most_populated_country(year: int):
145     """
146         Récupère le pays le plus peuplé en fonction de l'année depuis la collection MongoDB et les renvoie au format JSON.
147
148     Parameters:
149     - year (int): L'année de référence pour laquelle on veut récupérer le pays le plus peuplé.
150
151     Returns:
152     JSONResponse: Un objet JSONResponse contenant les données du pays le plus peuplé au format JSON.
153     """
154     df = getMostPopulatedCountry(year, mycollection)
155     return JSONResponse(content=df.to_dict(orient="records"))

```

Dans cette partie, l'instruction indique à FastAPI que la fonction `get_most_populated_country(year:int)` correspond au chemin `'/country_most_populated/{year}'`. Cette fonction a pour rôle d'aller sur MongoDB Atlas dans notre base de donnée pour récupérer le pays qui a la plus grande population en une année donnée fourni par le client et l'envoyer au format JSON (`JSONResponse()`). La requête MongoDB utilisée se trouve dans la fonction `getMostPopulatedCountry(year, mycollection)`.

```

150 def getMostPopulatedCountry(year, mycollection):
151     year = str(year)
152     # On récupère le pays le plus peuplé en fonction de l'année
153     country = mycollection.find_one(sort=[("pop"+year, -1)])
154     # On récupère les données de population
155     data = []
156     if country:
157         country_data = {
158             "Country": country["country"],
159             "Population in "+year: country.get("pop"+year)
160         }
161         data.append(country_data)
162     # On met les données dans un dataframe
163     df = pd.DataFrame(data)
164     return df

```

Cette fonction utilise `find_one(sort=[("pop"+year, -1)])` pour trouver le pays le plus peuplé pour une année donnée ; grâce à "sort" et '-1' il classe les pays par ordre décroissante et retourne le premier élément (pays). Après les avoir trouvés on crée un dataframe avec les colonnes "country" et "Population" et on l'envoie.

Endpoint 5 :

```

158 @country_api.get("/country_least_populated/{year}", response_model=dict, tags=["Countries"])
159 async def get_least_populated_country(year: int):
160     """
161         Récupère le pays le moins peuplé en fonction de l'année depuis la collection MongoDB et les renvoie au format JSON.
162
163     Parameters:
164     - year (int): L'année de référence pour laquelle on veut récupérer le pays le moins peuplé.
165
166     Returns:
167     JSONResponse: Un objet JSONResponse contenant les données du pays le moins peuplé au format JSON.
168     """
169     df = getLeastPopulatedCountry(year, mycollection)
170     return JSONResponse(content=df.to_dict(orient="records"))

```

```
Test this function
133 def getLeastPopulatedCountry(year, mycollection):
134     year = str(year)
135     # On récupère le pays le moins peuplé en fonction de l'année
136     country = mycollection.find_one(sort=[("pop"+year, 1)])
137     # On récupère les données de population
138     data = []
139     if country:
140         country_data = {
141             "Country": country["country"],
142             "Population in "+year: country.get("pop"+year)
143         }
144         data.append(country_data)
145     # On met les données dans un dataframe
146     df = pd.DataFrame(data)
147     return df
```

C'est le même concepte que le Endpoint 4, mais ici on veut trouver le pays le moins peuplé en une année donnée.

Endpoint 6 :

```
Test this function
174 @country_api.get("/average_pop/", response_model=dict, tags=["Countries"])
175 async def get_world_averagePop():
176     """
177         Récupère la population moyenne mondiale en 1980, 2000, 2010, 2022, 2023, 2030 et 2050
178         depuis la collection MongoDB et les renvoie au format JSON.
179
180     Returns:
181         JSONResponse: Un objet JSONResponse contenant les données de population moyenne au format JSON.
182     """
183     df = getWorldAveragePop(mycollection)
184     return JSONResponse(content=df.to_dict(orient="records"))
```

Dans cette partie, l'instruction indique à FastAPI que la fonction `get_world_averagePop(name : str)` correspond au chemin `'/average_pop/'`. Cette fonction a pour rôle d'aller sur MongoDB Atlas dans notre base de donnée pour calculer la population moyenne mondiale pour les années représentées dans notre base de données et l'envoi au format JSON (`JSONResponse()`). La requête MongoDB utilisée se trouve dans la fonction `getWorldAverage(mycollection)`.

```
Test this function
82 def getWorldAveragePop(mycollection):
83     # On récupère la densité de population moyenne du monde
84     worldAverageDensity = mycollection.aggregate([
85         {"$group": [
86             {"_id": "World Average Density",
87             "PopulationMoyEn1980": {"$avg": "$pop1980"},
88             "PopulationMoyEn2000": {"$avg": "$pop2000"},
89             "PopulationMoyEn2010": {"$avg": "$pop2010"},
90             "PopulationMoyEn2022": {"$avg": "$pop2022"},
91             "PopulationMoyEn2023": {"$avg": "$pop2023"},
92             "PopulationMoyEn2030": {"$avg": "$pop2030"},
93             "PopulationMoyEn2050": {"$avg": "$pop2050"}
94         ]}
95     ]
96     # On récupère les données de population
97     data = []
98     for country in worldAverageDensity:
99         country_data = {
100             "PopulationMoyEn1980": country.get("PopulationMoyEn1980"),
101             "PopulationMoyEn2000": country.get("PopulationMoyEn2000"),
102             "PopulationMoyEn2010": country.get("PopulationMoyEn2010"),
103             "PopulationMoyEn2022": country.get("PopulationMoyEn2022"),
104             "PopulationMoyEn2023": country.get("PopulationMoyEn2023"),
105             "PopulationMoyEn2030": country.get("PopulationMoyEn2030"),
106             "PopulationMoyEn2050": country.get("PopulationMoyEn2050")
107         }
108         data.append(country_data)
109     # On met les données dans un dataframe
110     df = pd.DataFrame(data)
111     return df
```

Ici on utilise une requête d'agrégation. Et on retourne un dataframe contenant la population moyenne mondiale de chaque année présente dans notre base.

Endpoint 7 :

```
188 @country_api.get("/countries_pop/", response_model=dict, tags=["Countries"])
189 async def get_countries_pops():
190     """
191         Récupère les informations de population (pop1980, pop2000, pop2010, pop2022, pop2023, pop2030, pop2050)
192         de tous les pays depuis la collection MongoDB et les renvoie au format JSON.
193     """
194     Returns:
195     JSONResponse: Un objet JSONResponse contenant les données des pays au format JSON.
196     """
197     df = getCountriesPop(mycollection)
198     return JSONResponse(content=df.to_dict(orient="records"))
```

Cette Endpoint nous permet de récupérer uniquement la population de chaque pays et pour toutes les années présentes dans notre base. La requête MongoDB utilisée se trouve dans `getCountriesPop(mycollection)`.

```
188 def getCountriesPop(mycollection):
189     # On récupère les données de population
190     countries = mycollection.aggregate([
191         {"$project": {
192             "country": "$country",
193             "pop1980": "$pop1980",
194             "pop2000": "$pop2000",
195             "pop2010": "$pop2010",
196             "pop2023": "$pop2023",
197             "pop2030": "$pop2030",
198             "pop2050": "$pop2050"
199         }}
200     ])
201     # On met les données dans un dataframe
202     df = pd.DataFrame(serializerList(countries))
203     return df
```

Ici on a utilisé une requête d'agrégation.

Endpoint 8 :

```
202 @country_api.get("/countries_areas_sup1_sup2/{sup1}/{sup2}", response_model=dict, tags=["Countries"])
203 async def get_countries_areas_sup1_sup2(sup1: float, sup2: float):
204     """
205         Récupère les informations de surface (area) de tous les pays dont la surface est comprise entre sup1 et sup2 depuis
206         la collection MongoDB et les renvoie au format JSON.
207     """
208     Parameters:
209     - sup1 (float): La valeur minimale de surface.
210     - sup2 (float): La valeur maximale de surface.
211
212     Returns:
213     JSONResponse: Un objet JSONResponse contenant les données des pays au format JSON.
214     """
215     df = getCountriesAreasSup1Sup2(sup1, sup2, mycollection)
216     return JSONResponse(content=df.to_dict(orient="records"))
```

Cet Endpoint permet de récupérer tous les pays dont la superficie est comprise entre sup1 et sup2 qui représentent des superficies fournies par le client. La requête MongoDB se trouve dans la fonction `getCountriesAreasSup1Sup2(sup1, sup2, mycollection)`.

```

    Test this function
170  def getCountriesAreasSup1Sup2(sup1, sup2, mycollection):
171      sup1 = float(sup1)
172      sup2 = float(sup2)
173      # On récupère les pays
174      countries = mycollection.find({"area": {"$gte": sup1, "$lte": sup2}})
175      # On récupère les données de superficie
176      data = []
177      for country in countries:
178          country_data = {
179              "Country": country["country"],
180              "Area": country.get("area")
181          }
182          data.append(country_data)
183      # On met les données dans un dataframe
184      df = pd.DataFrame(data)
185      return df

```

La requête utilisée est `find({"area": {"$gte": sup1, "$lte": sup2}})`. Après avoir trouvé les pays respectant la contrainte on crée un dataframe avec les colonnes “Country” et “Area” et on l’envoie.

Endpoint 9 :

```

    Test this function
220  @country_api.get("/custom_aggregation/{query}", response_model=dict, tags=["countries"])
221  async def get_aggregated_request(query: str):
222      """
223          Récupère les informations de tous les pays depuis la collection MongoDB et les renvoie au format JSON.
224
225          Parameters:
226          - query (str): La requête personnalisée à effectuer sur la collection MongoDB vous devez saisir sa sous
227          | forme de liste entre [contenu de la requête].
228          exemple : [{"$project": {"country": "$country", "pop1980": "$pop1980", "pop2000": "$pop2000",
229                      "pop2010": "$pop2010", "pop2023": "$pop2023", "pop2030": "$pop2030", "pop2050": "$pop2050"}}]
230
231          Returns:
232          JSONResponse: Un objet JSONResponse contenant les données des pays au format JSON.
233          """
234          query = json.loads(query)
235
236          df = getAggregationRequest(query, mycollection)
237          return JSONResponse(content=df.to_dict(orient="records"))
238

```

Dans cette partie, l’instruction indique à FastAPI que la fonction `get_aggregated_request(query: str)` correspond au chemin `/custom_aggregation/{query}`. Cette fonction a pour rôle de permettre au client d’écrire sa propre requête d’agrégation et de renvoyer au format JSON (`JSONResponse()`) la réponse de sa requête. Dans la fonction `getAggregationRequest(query, mycollection)` la requête sera exécutée.

```

182  # Fonction pour recevoir n'importe quelle requête d'agreg
183  def getAggregationRequest(aggregation, mycollection):
184      Test this function
185      # On récupère les données de population
186      countries = mycollection.aggregate(aggregation)
187      # On met les données dans un dataframe
188      df = pd.DataFrame(serializerList(countries))
189      return df

```

 **Endpoint 10 :**

```

241     test this function
242     @country_api.get("/custom_find/{query}", response_model=dict, tags=["Countries"])
243     async def get_find_request(query: str):
244         """
245             Récupère les informations de tous les pays depuis la collection MongoDB et les renvoie au format JSON.
246
247             Parameters:
248                 - query (str): La requête personnalisée à effectuer sur la collection MongoDB vous devez saisir sa sous
249                 | forme de liste entre [contenu de la requête].
250                 exemple : [{"area": {"$gte": 220000, "$lte": 280000}}, {"_id":0}]
251
252             Returns:
253                 JSONResponse: Un objet JSONResponse contenant les données des pays au format JSON.
254
255                 # convertir string en dictionnaire
256                 query = json.loads(query)
257                 df = getFindRequest(query, mycollection)
258                 return JSONResponse(content=df.to_dict(orient="records"))

```

Pour cette partie, c'est le même concept que l'**Endpoint 9** à la différence que grâce à cet Endpoint le client peut saisir sa propre requête MongoDB de type `find()` pour interroger notre base de données ; et ceci est traité dans la fonction `getFindRequest(query, mycollection)`.

```

191     test this function
192     def getFindRequest(query, mycollection):
193         if len(query) == 1:
194             # On récupère les données de population
195             countries = mycollection.find(query[0])
196         elif len(query) == 2:
197             countries = mycollection.find(query[0], query[1])
198         # On met les données dans un dataframe
199         df = pd.DataFrame(serializerList(countries))
200         return df

```

Lorsque la variable `query` contenant la requête du client, elle est transformée en une liste et ensuite transmise à la fonction `getFindRequest` dans laquelle on va extraire les éléments importants pour la requête. Lorsque l'info recherchée est trouvée elle est renvoyée au client sous forme d'un dataframe.

 **Endpoint 11 :**

```

261     Test this function
262     @country_api.get("/custom_distinct/{query}", response_model=dict, tags=["Countries"])
263     async def get_distinct_request(query: str):
264         """
265             Récupère les informations de tous les pays depuis la collection MongoDB et les renvoie au format JSON.
266
267             Parameters:
268                 - query (str): La requête personnalisée à effectuer sur la collection MongoDB.
269                 exemple : ["area", {"_id":0}]
270
271             Returns:
272                 JSONResponse: Un objet JSONResponse contenant les données des pays au format JSON.
273
274                 query = json.loads(query)
275                 df = getDistinctRequest(query, mycollection)
276                 return JSONResponse(content=df.to_dict(orient="records"))

```

```

Test this function
212 def getDistinctRequest(distinct, mycollection):
213     if len(distinct) == 1:
214         # On récupère les données de population
215         countries = mycollection.distinct(distinct[0])
216     elif len(distinct) == 2:
217         countries = mycollection.distinct(distinct[0], distinct[1])
218
219     # On met les données dans un dataframe
220     df = pd.DataFrame(countries)
221
222     return df

```

Cette partie est pareille que les 2 précédentes ; sauf que cet Endpoint permet au client d'écrire ces propres requêtes MongoDB de type `distinct()`.

Endpoint 12 :

```

Test this function
277 @country_api.get("/nb_countries_supavg/{year}", response_model=dict, tags=["Countries"])
278 async def get_nb_countries_supavg(year: int):
279     """
280         Récupère le nombre de pays dont la population en une année est supérieure à la moyenne mondiale depuis
281         a collection MongoDB et les renvoie au format JSON.
282
283     Returns:
284         Un dict contenant le nombre de pays dont la population en une année est supérieure à la moyenne mondiale.
285         Sous la forme {"Mondiale average population in " + str(year) : moy,
286         |   "nb_countries_supavg": nb_countries_supavg}
287     """
288     nb_countries_supavg, moy = getNbCountriesPopSupAvg(year, mycollection)
289
290     return {"Mondiale average population in " + str(year) : moy,
291             "Number of countries with population grather than average population in " + str(year) : nb_countries_supavg}
292

```

Dans cette partie, l'instruction indique à FastAPI que la fonction `get_nb_countries_supavg(year: int)` correspond au chemin `/nb_countries_supavg/{year}`. Cette fonction a pour rôle d'aller dans notre base de données en fonction de l'année fourni par le client va rechercher les pays dont la population est supérieure à la population moyenne mondiale et enfin fait un décompte et retourne la moyenne de la population et le nombre de pays trouvé (`JSONResponse()`). Cette opération se trouve dans la fonction `getNbCountriesPopSupAvg(year, mycollection)`.

```

Test this function
210 def getNbCountriesPopSupAvg(year, mycollection):
211     year = str(year)
212
213     # On récupère la population moyenne
214     avg = mycollection.aggregate([
215         {"$group": {
216             "_id": "World Average Population",
217             "avg": {"$avg": "$pop"+year}
218         }
219     })
220
221     # On convertir le curseur en une liste de documents Python
222     avg = list(avg)
223     avg = avg[0]["avg"]
224
225     # On récupère le nombre de pays dont la population est supérieure à la population moyenne
226     nbCountries = mycollection.count_documents({"pop"+year: {"$gte": avg}})
227
228     return nbCountries, avg

```

Ici nous avons utilisé 2 requêtes MongoDB, la première (une requête de type aggregate) pour trouver la population moyenne. Ensuite une requête `count_documents()` pour compter le nombre de pays ayant une population supérieure à la moyenne et retourne le nombre et la moyenne.

Endpoint 13 :

```

Test this function
294 @country_api.get("/nb_countries_infavg/{year}", response_model=dict, tags=["Countries"])
295 async def get_nb_countries_infavg(year: int):
296     """
297     Récupère le nombre de pays dont la population en une année est inférieure à la moyenne mondiale depuis
298     la collection MongoDB et les renvoie au format JSON.
299
300     Returns:
301     Un dict contenant le nombre de pays dont la population en une année est inférieure à la moyenne mondiale.
302     Sous la forme {"Mondiale average population in " + str(year) : moy,
303     | "nb_countries_infavg": nb_countries_infavg}
304     """
305
306     nb_countries_infavg, moy = getNbCountriesPopInfAvg(year, mycollection)
307
308     return {"Mondiale average population in " + str(year) : moy,
309             "Number of countries with population lower than average population in " + str(year) : nb_countries_infavg}

Test this function
310 def getNbCountriesPopInfAvg(year, mycollection):
311     year = str(year)
312
313     # On récupère la population moyenne
314     avg = mycollection.aggregate([
315         {"$group": {
316             "_id": "World Average Population",
317             "avg": {"$avg": "$pop"+year}
318         }}
319     ])
320
321     # On convertir le curseur en une liste de documents Python
322     avg = list(avg)
323     avg = avg[0]["avg"]
324
325     # On récupère le nombre de pays dont la population est inférieure à la population moyenne
326     nbCountries = mycollection.count_documents({"pop": {"$lt": avg}})
327     return nbCountries, avg

```

Cette partie est similaire à la précédente sauf qu'ici c'est le nombre de pays ayant une population inférieure à la population moyenne en fonction de l'année fournie par le client.

5. La partie exécution de l'application

```

Fichier Edition Sélection Affichage Atteindre Exécuter Terminal Aide main.py - Application -
EXPLORATEUR ...
APPLICATION
> __pycache__
> config
> models
> mongoenv
routes
> __pycache__
__init__.py
country_api.py
> schemas
> static
main.py
Procfile

main.py > ...
1 from fastapi import FastAPI
2 from routes.country_api import country_api
3
4 app = FastAPI()
5
6
7 app.include_router(country_api)
8
9
10 #if __name__ == "__main__":
11 #    import uvicorn
12 #    uvicorn.run(app, host="127.0.0.1", port=8000)

```

Cette partie représente le point d'entrée de l'API. A la ligne 4, on instancie l'objet `app` qui est un objet `FastAPI()`. A la ligne 7 on connecte le fichier contenant les Endpoints de notre API. Les trois dernières lignes servent à lancer notre API en local, mais ici nous les avons mis en commentaire parce que nous avons déployé notre API en ligne.

VII. La documentation de l'API

La documentation d'une API (Interface de Programmation Applicative) est un ensemble de ressources et d'informations destinées à aider les développeurs à comprendre, utiliser et intégrer une API dans leurs applications.

Ainsi, une documentation accessible et facilement exploitable est une condition préalable au développement des API. Il est important d'avoir une documentation toujours à jour quand le code/les fonctionnalités de l'API évoluent.

A travers le lien suivant accédez à la documentation claire de notre API [cliquez ici](#).

VIII. Quelques captures d'écran présentant les résultats de tests obtenus

Une des supers fonctionnalités dont dispose le Framework FastAPI, est une plateforme web nous permettant de tester nos API et elle offre une description d'utilisation. Donc plus besoin de faire appel à des logiciels extérieurs tel que Postman pour les tests. N'est-ce pas génial ça !?

Les requêtes d'interrogations

The screenshot shows a MongoDB API interface for the 'countries_info' endpoint. The top bar indicates a GET request to '/countries_info/' with the description 'Get Countries'. Below this, a note states: 'Récupère toutes les informations des pays depuis la collection MongoDB et les renvoie au format JSON.' and 'Returns: JSONResponse: Un objet JSONResponse contenant les données des pays au format JSON.'.

The 'Parameters' section shows 'No parameters' with 'Cancel' and 'Execute' buttons.

The 'Responses' section includes a 'Curl' command:

```
curl -X 'GET' \
'https://josh-mongodb-api.onrender.com/countries_info/' \
-H 'accept: application/json'
```

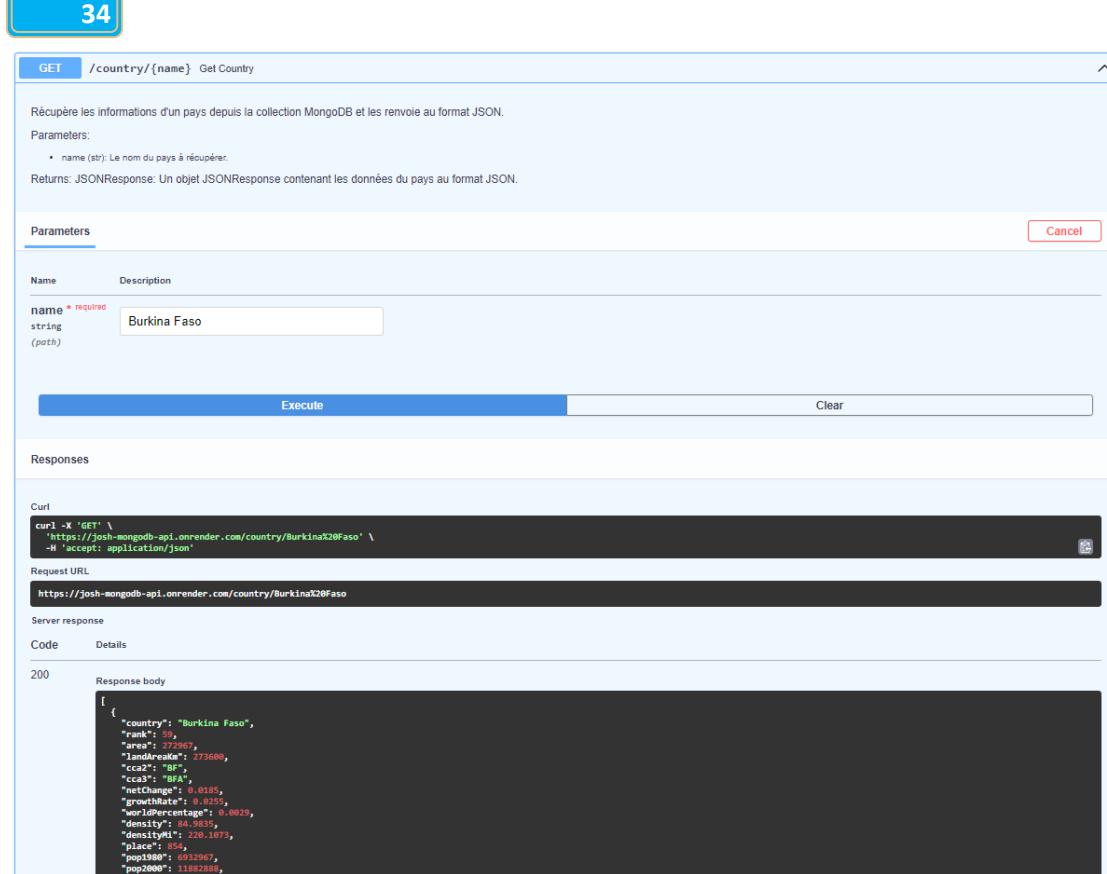
The 'Request URL' is listed as https://josh-mongodb-api.onrender.com/countries_info/.

The 'Server response' section shows a JSON array of two documents:

```
[
  {
    "_id": "6dc9be365396a056d4811863",
    "country": "Pakistan",
    "rank": 5,
    "area": "881912.0",
    "landAreaKm": "770880.0",
    "code": "PK",
    "ccn3": "PAK",
    "netChange": "0.1495",
    "growthRate": "0.0198",
    "worldPercentage": "0.03",
    "density": "311.9625",
    "iso3166alpha2": "PK",
    "place": "586",
    "pop1980": 80624057,
    "pop2000": 154369924,
    "pop2010": 194454498,
    "pop2012": 204445427,
    "pop2013": 244445658,
    "pop2030": 274029836,
    "pop2050": 367808468
  },
  {
    "_id": "6dc9be365396a056d481187b",
    "country": "South Korea",
    "rank": 29,
    "area": "100240.0"
  }
]
```

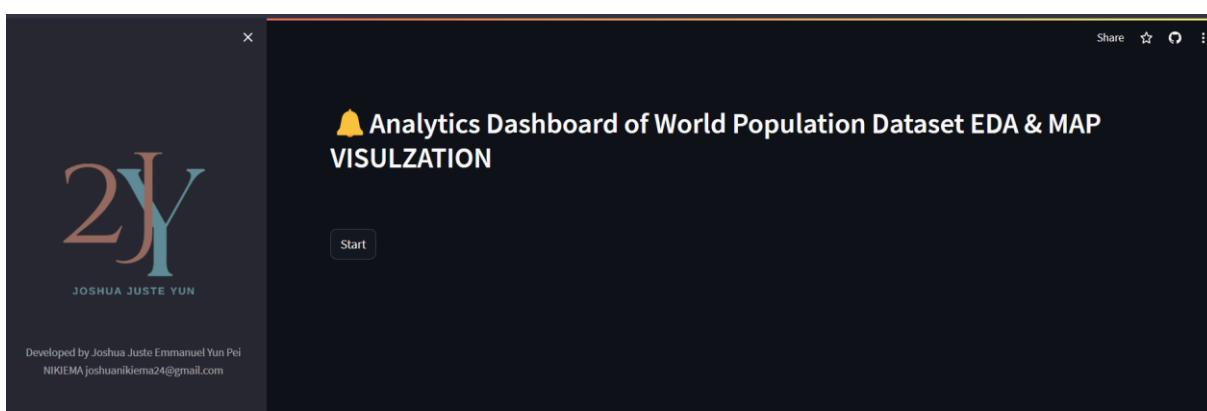
There are 'Code' and 'Details' tabs for the 200 status code, and a 'Download' button is visible.

Il s'agit de l'Endpoint nous permettant d'aller récupérer tous les documents de la collection.



L'Endpoint pour rechercher un pays à partir de son nom.

Pour la suite nous allons utiliser une interface graphique (Dashboard) que nous avons développé avec le Framework Streamlit de python. Ce Dashboard est en quelque sorte une application pour consommer notre API.



Page d'accueil du Dashboard (cliquez sur Start)



35

Analytics Dashboard of World Population Dataset EDA & MAP VISULATION

Specific Requests

Country Name

Find Country

Year

1980
2000
2010
2023
2030
2050

Main Menu

Home
IUDC
Countries and their area
Map of the population in 2000, 2010 and 2023
Specific Requests
Personalized Requests

Share

Cliquez sur "Specific Requests" pour faire quelques requêtes.

Year

1980
2000
2010
2023
2030
2050

Find the most populated country following the year

Search The Most Populated Country

The data has been successfully recovered.

	Country	Population in 2010
0	China	1,348,191,368

Il s'agit du test du **Endpoint 4** pour trouver le pays le plus peuplé en fonction de l'année (ici en 2010 c'est la Chine).

Year

1980
2000
2010
2023
2030
2050

Find the most populated country following the year

Search The Most Populated Country

Find the least populated country following the year

Search The Least Populated Country

The data has been successfully recovered.

	Country	Population in 2010
0	Vatican City	596

Il s'agit du test du **Endpoint 5** pour trouver le pays le moins peuplé en fonction de l'année (ici en 2010 c'est le Vatican).



Find countries whose area is between two values

Minimum value
240000,00

Maximum value
280000,00

The data has been successfully recovered.

	Country	Area
0	Gabon	267,668
1	New Zealand	270,467
2	Ecuador	276,841
3	United Kingdom	242,900
4	Western Sahara	266,000
5	Burkina Faso	272,967
6	Guinea	245,857
7	Uganda	241,550

Il s'agit du test du **Endpoint 8** pour trouver les pays dont la superficie est comprise entre 2 valeurs ; ici entre **240000km²** et **280000km²**.

Display the average world population for each year

The data has been successfully recovered.

	PopulationMoyEn1980	PopulationMoyEn2000	PopulationMoyEn2010	PopulationMoyEn2022	PopulationMoyEn2023	PopulationMoyEn2030	PopulationMo
0	18,984,616.9829	36,514,605.3333	29,845,235.0427	34,074,414.7137	34,374,424.7436	None	41,486.;

Il s'agit du test du **Endpoint 6** pour trouver la population mondiale de chaque année.

Find number of countries with a population less than average

Select years
2023

The data has been successfully recovered.

Mondiale average population in 2023: 34374424.743589744

Number of countries with population lower than average population in 2023: 190

Il s'agit du test du **Endpoint 13** pour trouver le nombre de pays ayant une population inférieure à la population moyenne (190 pays).

The screenshot shows a MongoDB aggregation pipeline in a UI. The pipeline is:

```
[{"$group": { "_id": "World Average Population", "PopulationMoyEn2000": {"$avg": "$pop2000"}, "PopulationMoyEn2010": {"$avg": "$pop2010"}, "PopulationMoyEn2022": {"$avg": "$pop2022"} }}
```

A button labeled "Search_Agg" is visible. A green success message at the bottom states: "The data has been successfully recovered." Below this is a table with the following data:

	_id	PopulationMoyEn2000	PopulationMoyEn2010	PopulationMoyEn2022
0	World Average Population	26,269,468.8162	29,845,235.0427	34,074,414.7137

Il s'agit du test du **Endpoint 9** permettant à un utilisateur d'exécuter sa propre requête d'agrégation. La requête que nous avons exécutée permet de trouver la moyenne de la population en 2000, 2010 et 2022.

The screenshot shows a MongoDB find query in a UI. The query is: [{"cca2": "SN"}, {"_id": 0, "country": 1, "pop2023": 1, "cca2": 1, "cca3": 1}]

A button labeled "Search_Find" is visible. A green success message at the bottom states: "The data has been successfully recovered." Below this is a table with the following data:

	country	cca2	cca3	pop2023
0	Senegal	SN	SEN	17,763,163

Il s'agit du test du **Endpoint 10** permettant à un utilisateur d'exécuter sa propre requête `find()`. La requête que nous avons exécutée recherche le pays dont l'abréviation à 2 caractères est "**SN**" et on fait une projection sur son nom, sa population en 2023 et ses abréviations à 2 et 3 caractères.

The data has been successfully recovered.

0	0	BJ
---	---	----

Il s'agit du test du **Endpoint 11** permettant à un utilisateur d'exécuter sa propre requête **distinct()**. La requête que nous avons exécutée recherche les abréviations à 2 caractères de manière distincte et on passe un filtre qui est le pays nommé **Benin**.

La requête de création

Nous allons tester une insertion d'un pays que nous allons appeler **JoshLand**. Mais avant nous allons faire un test pour s'assurer qu'il n'existe pas.

NB : Pour les requêtes d'insertion, de mise à jour et suppression cliquez sur **IUDC**

Developed by Joshua Juste Emmanuel Yun Pei
NIKIEMA joshuaniema24@gmail.com

Main Menu

- Home
- IUDC**
- Countries and their area
- Map of the population in 2000, 2010 and 2023
- Specific Requests
- Personalized Requests

Country Name
JoshLand

Find Country

Country not found!

Il n'existe pas.

Insert, Update and Delete a Country

Action
 Insert
 Update
 Delete

Country

Rank

Area

Land Area (km²)

CCA2

Population in 2023

Population in 2030

Population in 2050

Population in 2050

Insert

Successfully inserted data.

	0
country	JoshLand
rank	1
area	120000.0
landAreaKm	121000.0
cca2	JL

Au niveau de MongoDB Atlas :

Atlas Emmanuel J... Access Manager Billing

Project 0 Data Services App Services Charts

DEPLOYMENT Database SERVICES world-database.global-country-data

DATABASES: 1 COLLECTIONS: 1

+ Create Database Search Namespaces

Find Indexes Schema Anti-Patterns Aggregati

Filter {country:"JoshLand"}

QUERY RESULTS: 1-1 OF 1

```

_id: ObjectId('64cf711d2df4e8d7b1d36fba')
country: "JoshLand"
rank: 1
area: 120000
landAreaKm: 121000
cca2: "JL"
cca3: "JL"
netChange: 0.01
growthRate: 0.4
worldPercentage: 0.2
density: 430
densityMi: 1280
place: 20
pop1988: 7600000
pop2000: 9000000
pop2010: 10500000
pop2022: 20430000
pop2023: 21000000
pop2050: 33000000

```

On voit que l'insertion à belle et bien réussie.

La requête de mise à jour

Nous allons tester la modification de quelques données du pays "JoshLand".

NB : Si l'on ne souhaite pas changer une donnée on garde sa valeur par défaut dans la zone de saisie.

Ici nous voulons modifier le Net Change, World Percentage, Density et Density (mi²)

Vérifions dans MongoDB Atlas

Avant

QUERY RESULTS: 1-1 OF 1

```
_id: ObjectId('64cf711d2df4e8d7b1d36fba')
country: "JoshLand"
rank: 1
area: 120000
landAreaKm: 121000
cca2: "JL"
cca3: "JLD"
netChange: 0.01
growthRate: 0.4
worldPercentage: 0.2
density: 430
densityMi: 1200
place: 20
pop1980: 7000000
pop2000: 9000000
pop2010: 10500000
pop2022: 20430000
pop2023: 21000000
pop2030: 27000000
pop2050: 33000000
```

Après

QUERY RESULTS: 1-1 OF 1

```
_id: ObjectId('64cf711d2df4e8d7b1d36fba')
country: "JoshLand"
rank: 1
area: 120000
landAreaKm: 121000
cca2: "JL"
cca3: "JLD"
netChange: 0.02
growthRate: 0.4
worldPercentage: 0.33
density: 720
densityMi: 2100
place: 20
pop1980: 7000000
pop2000: 9000000
pop2010: 10500000
pop2022: 20430000
pop2023: 21000000
pop2030: 27000000
pop2050: 33000000
```

La requête de suppression

Nous allons tester maintenant la suppression d'un document (le pays JoshLand)

The screenshot shows a web interface titled "Insert, Update and Delete a Country". On the left, there's a sidebar with a "Main Menu" containing "Home", "IUDC" (which is highlighted in red), "Countries and their area", "Map of the population in 2000, 2010 and 2023", and "Specific Requests". The main content area has a heading "Insert, Update and Delete a Country" with a file icon. Below it, there are three radio buttons for "Action": "Insert" (unselected), "Update" (unselected), and "Delete" (selected). An "ID" input field contains the value "64cf711d2df4e8d7b1d36fba". A "Delete" button is present. At the bottom, a green bar displays the message "Successfully deleted data."

Au niveau de MongoDB Atlas :

The screenshot shows the MongoDB Atlas interface. On the left, there's a sidebar with "+ Create Database" and "Search Namespaces". Under "world-database", the "global-country-data" collection is selected. The main panel shows the "world-database.global-country-data" collection with the following details: STORAGE SIZE: 104KB, LOGICAL DATA SIZE: 73.01KB, TOTAL DOCUMENTS: 236, INDEXES TOTAL SIZE: 40KB. Below this, there are tabs for "Find", "Indexes", "Schema Anti-Patterns", "Aggregation", and "Search Indexes". A "Filter" button is shown with the query "[country: "JoshLand"]". At the bottom, a "QUERY RESULTS: 0" message is displayed.

Il n'existe plus.

Nous avons ici présenté quelques résultats de requêtes. Pour tester vous-même, ou découvrir les autres fonctionnalités de notre API, nous allons mettre à votre disposition les liens vers notre Dashboard utilisant notre API et vers la plateforme FastAPI de test de notre API.

- ➡️ **Lien vers le Dashboard : [cliquez ici](#).**
- ➡️ **Lien vers la plateforme FastAPI de test de l'API : [cliquez ici](#).**

Conclusion

En conclusion, notre décision de choisir le projet visant à créer une API RESTful connectée à MongoDB Atlas s'est avérée être une expérience enrichissante. Malgré le caractère abstrait du concept d'API au départ, notre engagement dans le processus de développement nous a permis de mieux comprendre les rouages de cette technologie et son importance dans la création d'interfaces flexibles et interactives.

Nous avons parcouru les différentes étapes de création, depuis la mise en place du cluster sur MongoDB Atlas jusqu'au chargement des données. Nous avons également exploré l'environnement de travail et utilisé diverses bibliothèques pour construire une API fonctionnelle et conviviale. Les requêtes que nous avons développées, allant des interrogations simples aux analyses plus approfondies, ont démontré la puissance de l'API que nous avons conçue.

Ce projet nous a non seulement permis d'acquérir des compétences pratiques en matière de développement d'API et de manipulation de bases de données NoSQL, mais il nous a également ouvert les portes vers un domaine essentiel dans le monde de la programmation moderne. En fin de compte, cette expérience a renforcé notre compréhension de la façon dont les technologies de base de données et les API peuvent collaborer pour créer des solutions innovantes et efficaces.

Vous pouvez télécharger le code complet du projet sur mon GitHub :

- API : [ici](#)
- Dashboard : [ici](#)