Projet New York Times - Bootcamp Data Engineer Février 2023

Mickael Gaspar, Can Baskurt, Clément Guiraud

Github : https://github.com/dst-nynews/dst-nynews
API New York TImes : https://developer.nytimes.com/apis
Doc de suivi projet: Suivi de projet
Documentation des fonctions: Fonctions du projet

Rappel sur les objectifs du projet et les données utilisées

Le projet que nous souhaitons mettre en place se fixe 3 objectifs métiers différents pouvant être, ou non, réalisés en fonction du temps et de notre avancement.

Nous souhaitons tout d'abord proposer un *dashboard* permettant de trier, filtrer et rechercher les éléments essentiels des articles et de la sémantique proposés par le New York Times. Il s'agira de permettre à l'utilisateur d'obtenir facilement des informations sur ce que propose le journal.

Dans un deuxième temps, il nous semblerait intéressant de mettre en relation les données sur le COVID proposées par le New York Times, et si le temps nous le permet par d'autres sources, avec les articles publiés par le New York Times. Cette comparaison pourrait permettre, notamment, d'avoir une meilleure compréhension de comment l'évolution réelle de la pandémie (évaluée par les données covid brutes) et sa retransmission dans un média grand public ont pu, ou pas, évoluer dans le temps.

Enfin, si le temps nous le permet, nous souhaiterions mettre en place une application web sous forme de mini-jeu de prédiction d'articles à succès. Il s'agira de permettre aux utilisateurs de prédire, parmi un set d'articles publiés par le New York Times, lequel sera présent dans les données "most popular" que propose le journal. La proposition de l'utilisateur pourra être comparée avec celle d'un algorithme de *Machine Learning* pour savoir si l'utilisateur est "meilleur ou moins bon qu'une intelligence artificielle".

Rappel sur les livrables 1, 2 et 3

A la fin de notre premier livrable nous présentions les suites du projet comme suit :

La prochaine grande étape à mettre en place est le choix du type de stockage qu'il nous faudra utiliser. Ce choix est à la fois technique (BDD SQL ? NoSQL ?) et pratique (stockage en local par chaque membre du groupe ? stockage sur un VM? stockage dans le cloud ?).

La solution vers laquelle nous nous orientons est celle d'un stockage en local avec l'utilisation de scripts python assurant la similarité des données exploitées chez chaque membre du groupe.

Comme nous le notions dans le 2eme livrable, le choix des outils s'est finalement porté sur une base de données NoSQL, MongoDB dans sa version cloud Atlas, pour les données directement issues de l'API du New-York Times, et une base de données SQL, PostgreSQL, pour les données du Covid. Le peuplement de ces bases de données se fait à l'aide de scripts pythons pour l'ensemble des API du New-York Times que nous avons sélectionné.

Le livrable 3 présentait les 3 cas d'usages que nous avons sélectionné : l'affichage des articles les plus populaires d'un jour choisi par l'utilisateur, la présentation d'un tableau de données sur le Covid en fonction d'une date de début et de fin déterminées par l'utilisateur et enfin l'affichage des informations d'un concept officiels du New-York Times choisi par l'utilisateur en fonction de la chaîne de caractère de son choix.

Ce quatrième livrable sera centré sur la mise en production de notre travail, c'est-à-dire la mise en place d'un API à l'interface entre nos bases de données et nos *frontend* et la dockérisation de notre projet.

L'API à l'interface entre BDD et frontend

Pour faciliter la mise en production de notre travail, nous avons décidé de proposer une API permettant de connecter facilement nos bases de données et nos *frontends*. Cette solution à le double avantage de faciliter le déploiement de notre projet d'introduire de la flexibilité dans son évolution. En effet, grâce à l'API toutes modifications futures de notre *backend* sera invisible à toute personne souhaitant utiliser les données que nous mettons à disposition.

Pour mettre cela en place nous avons décidé d'utiliser le *framework* FastAPI.

Notre code est structuré de la manière suivante :

1. Un dossier backend contenant un fichier d'initialisation des connexions avec nos BDD (database.py) et d'un fichier d'orchestration de l'API (main.py)
2. De sous-dossiers pour chaque cas d'usage

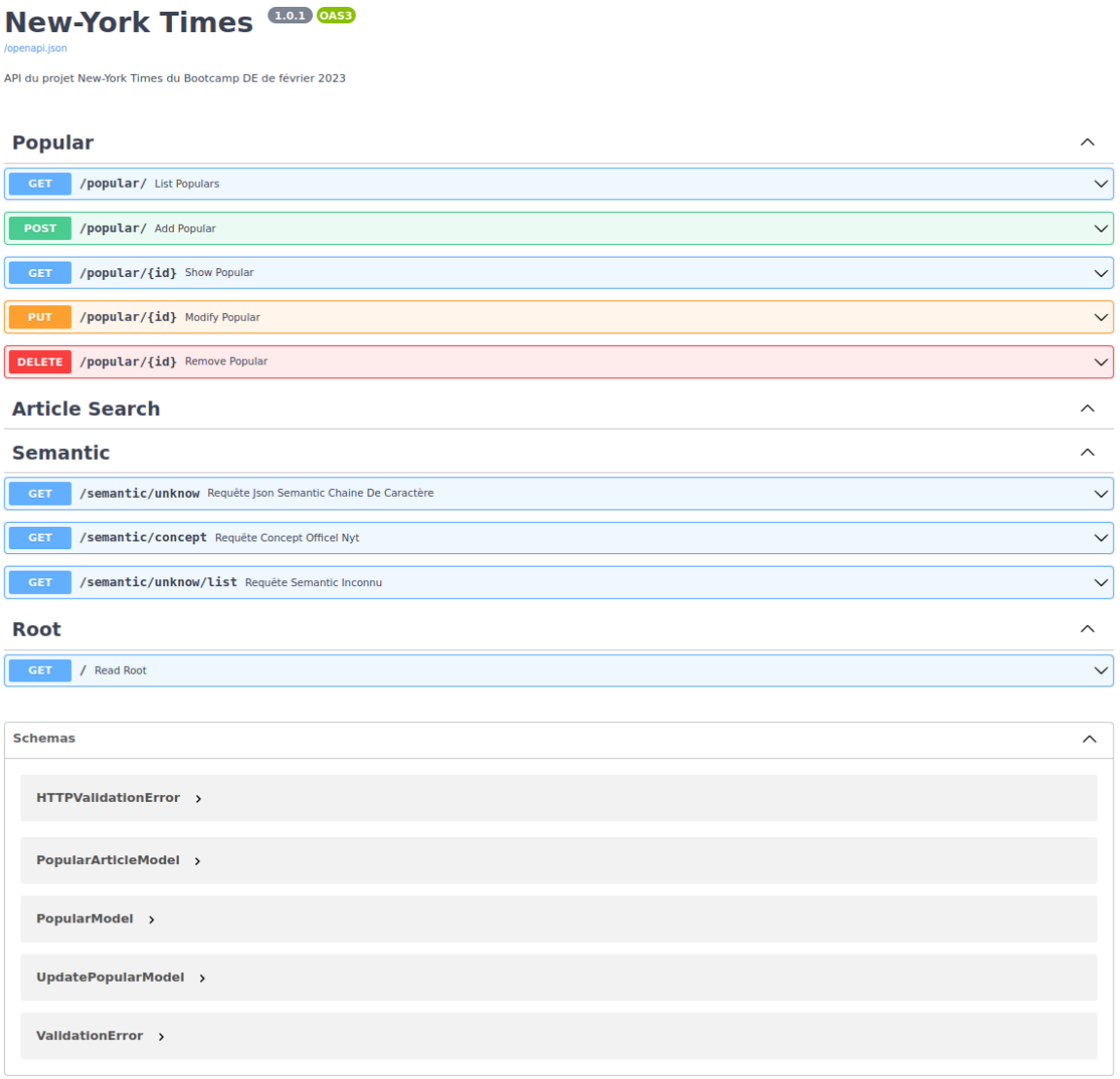
```python
from fastapi import FastAPI

# Local imports

from popular.routes import router as popular_router
from semantic.routes import routeSe as semantic_router


# Instanciate the API controller
app = FastAPI(
    title = "New-York Times  ",
    description = "API du projet New-York Times du Bootcamp DE de février 2023",
    version = "1.0.1",
    openapi_tags=[
    {
        'name': 'Popular'
    },
    {
        'name': 'Article Search',
    },
    {
        'name': 'Semantic'
    }
])

# Inject the endpoint "popular" in the scope of the controller
app.include_router(popular_router, tags=["Popular"], prefix="/popular")

# Inject the endpont "semantic"
app.include_router(semantic_router, tags=["Semantic"], prefix="/semantic")


# Root endpoint
@app.get("/", tags=["Root"])
async def read_root():
    return {"message": "Welcome to the NyNews API!"}


if __name__ == "__main__":
    # DEV mode: start local server for testing
    import uvicorn

    uvicorn.run(
        "main:app",
        host="0.0.0.0",
        port=8000,
        reload=True,
    )
```

Fichier main.py dans une version non définitive

```python
""" Routes associating each http request with a specific path for this endpoint.
"""

from fastapi import APIRouter, Body
from fastapi.encoders import jsonable_encoder

# Local imports
from .crud import (create_popular, delete_popular,
                   read_popular, read_popular_index, update_popular)
from .models import (PopularModel, UpdatePopularModel,
                     ResponseModel, ErrorResponseModel)


# Instanciate a router for this endpoint
router = APIRouter()


@router.get("/", response_description="Index of popular articles retrieved")
async def list_populars():
    populars = await read_popular_index()
    if populars:
        return ResponseModel(
            populars, "Index of group of popular articles retrieved successfully"
        )
    return ResponseModel(populars, "Empty list returned")


@router.get("/{id}", response_description="Popular articles retrieved")
async def show_popular(id):
    popular = await read_popular(id)
    if popular:
        return ResponseModel(
            popular, "Group of popular articles retrieved successfully"
        )
    return ErrorResponseModel(
        "An error occurred.", 404, "Group of popular articles doesn't exist."
    )
```

Une partie du fichier routes.py du sous-dossier popular codant pour le cas d'usage 1

Cette solution nous permet d'obtenir un ensemble de routes permettant la mise en place de nos cas d'usages. Lorsque l'API est en état de fonctionnement ces routes sont disponibles en local à l'adresse localhost:8000/docs.



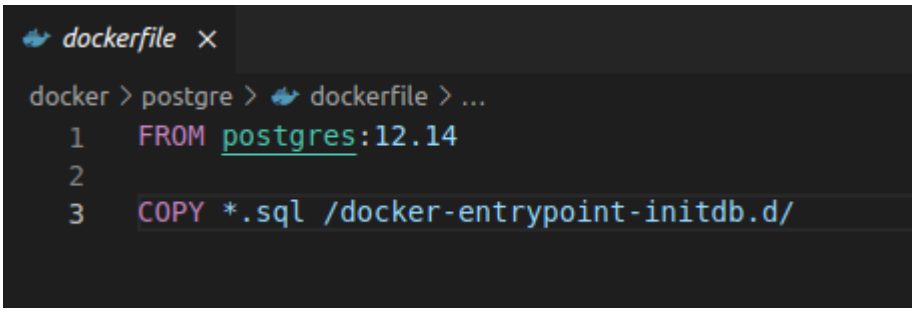
Version non définitive du Swagger de l'API disponible en local sur localhost:8000/docs

Containerisation

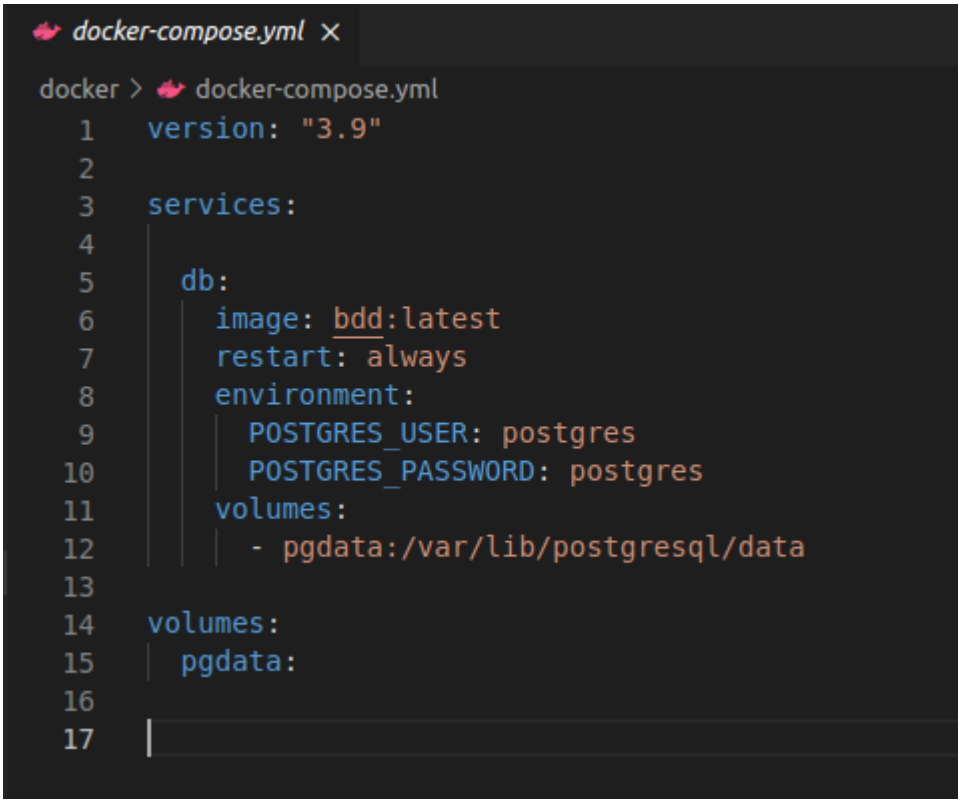
A termes, notre projet sera composé de 3 conteneurs docker. Le premier, déjà fonctionnel, contient la base de données SQL des données sur le Covid. Le deuxième sera le container de l'API tandis que le troisième contiendra nos *frontends*.

Le docker de la base de données SQL est constitué d'une image postgres officielle (version 12.14) à laquelle a été ajouté un fichier .sql provenant d'un dump de la base de données construite en local. Ainsi, lors de son démarrage le container détecte la présence d'un fichier .sql dans son *entrypoint* et reconstitue la base de données. Il suffit donc de lancer les requêtes sur l'adresse ip du container pour accéder aux informations que l'on souhaite obtenir. A termes, les dockers seront lancés à l'aide d'un fichier docker-compose.yml. Ils partageront ainsi un même réseau, les rendant accessible non plus seulement par l'utilisation de leur adresse ip mais *via* le nom du conteneur. Cela permettra de déployer le

projet en faisant totalement abstraction de la machine sur laquelle il sera lancé et sans avoir à rechercher l'adresse ip des containers.



Le dockerfile de la BDD SQL



Le docker-compose permettant de lancer le container de la bdd (et à termes l'ensemble des containers)

Une fois la containerisation des différentes parties de notre projet effectuée, nous automatiserons le déploiement à l'aide d'un simple script shell permettant de lancer les différents containers.

La seule potentielle limite à cette automatisation est la nécessité, pour se connecter à la base de données NoSQL Atlas, d'avoir une adresse ip autorisée. Si nous arrivons à contourner ce besoin d'une autorisation pour chaque adresse ip, alors notre travail sera déployable sur n'importe quelle machine linux équipée de Docker.

Perspectives

La dernière étape de notre projet sera celle de l'automatisation, non plus du déploiement, mais de la mise à jour de nos données. Si le temps nous le permet, nous souhaitons automatiser le requêtage de l'API MostPopular. En effet, cette API est la seule à avoir ses données qui changent tous les jours. Pour le moment, son requêtage se fait à l'aide d'un cron job tournant en local sur la machine d'un des membres du projet. Idéalement, nous voudrions remplacer ce cronjob soit par une dag airflow soit par un job sur github action.