Projet Architecture Microservices

Abdelbarie EL METNI - Kenzi ABDELATIF

Juin 2025

1 Indication sommaire pour compiler / exécuter le projet

Le projet est entièrement conteneurisé à l'aide de **Docker**, ce qui simplifie considérablement sa compilation et son exécution. L'ensemble des services (microservices, base de données PostgreSQL, etc.) est orchestré à l'aide de **Docker Compose**.

Les instructions détaillées pour lancer le projet sont disponibles dans le fichier `README.md` situé à la racine du dépôt.

En résumé, après avoir installé Docker, il suffit de se placer dans le répertoire du projet puis d'exécuter la commande suivante pour lancer tous les services :

```
docker compose up --build -d
```

Lors des exécutions suivantes (après le premier build), la commande simplifiée suivante suffit :

```
docker compose up -d
```

Une étape préalable consiste à créer manuellement la base de données `information`, utilisée par les microservices. Cela peut être réalisé via une commande `docker exec` fournie dans le fichier `README.md`.

2 Schéma de la base de données

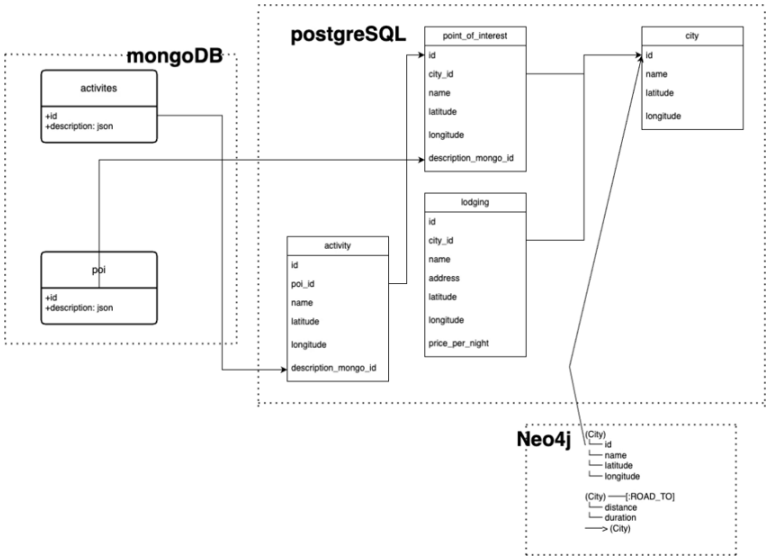

Figure 1: Schéma de la base de données utilisée dans le projet (MongoDB, PostgreSQL, Neo4j)

Le projet repose sur une architecture polyglote combinant trois types de bases de données : **MongoDB**, **PostgreSQL** et **Neo4j**, chacune jouant un rôle spécifique selon la nature des données manipulées.

- **MongoDB** est utilisée pour stocker les données semi-structurées, sous forme de documents JSON. Trois collections principales sont définies :
  - `activities` : contient les descriptions des activités touristiques que définissent les configurateurs.
  - `poi` (points d'intérêt) : contient les descriptions des points d'intérets touristiques que définissent les configurateurs.
  - `trip` : structure complexe représentant un voyage. Chaque document contient :
    - un identifiant (`id`) et une `reference` unique,
    - une `startDate` et `endDate`,
    - une liste de `steps`, où chaque étape inclut :
      - les dates de l'étape,
      - un nom de l'étape (`tripName`),
      - une liste d'`activities` (nom, prix, lieu, date),
      - un objet `lodging` (nom, adresse, prix),
      - un objet `nextStep` précisant la prochaine ville, la distance et le temps de trajet.
- **PostgreSQL** constitue la base de référence relationnelle, utilisée pour les entités principales de géolocalisation :
  - `point_of_interest` : identifie chaque lieu avec ses coordonnées et référence un document MongoDB via `description_mongo_id`.
  - `activity` : décrit une activité localisée, liée à un POI, et associée à un document JSON.
  - `lodging` : décrit les hébergements par ville, avec coordonnées et prix.
  - `city` : répertoire des villes, utilisé comme clé de référence pour les autres tables.
- **Neo4j** est exploitée pour modéliser le graphe des distances entre villes :
  - chaque `City` est un nœud avec ses attributs (nom, latitude, longitude),
  - les relations `:ROAD_TO` connectent deux villes avec les attributs `distance` et `duration`, facilitant les calculs d'itinéraires.

Ce modèle polyglotte permet de tirer parti des points forts de chaque technologie : la flexibilité documentaire de MongoDB pour les voyages et les contenus enrichis, la robustesse relationnelle de PostgreSQL pour les entités géographiques structurées, et la puissance de Neo4j pour les recherches sur les graphes de villes et les calculs de distance.

3 Schéma d'architecture globale

Le projet est basé sur une architecture orientée microservices, utilisant le modèle API Gateway + Service Discovery. Chaque service métier est autonome et accède à sa propre base de données selon ses besoins.

Choix techniques

- **Spring Cloud Eureka** est utilisé pour implémenter un *Discovery Server* (Service Registry). Chaque microservice s'enregistre dynamiquement pour faciliter la détection et la communication inter-services.
- **Spring Cloud Gateway** route les requêtes des clients vers les microservices internes (service gateway).
- **Information Service** accède à une base relationnelle **PostgreSQL**, utilisée pour stocker des entités classiques comme les villes, les points d'intérêt, les activitées et les hébergements.
- **Activities Service** utilise **MongoDB** pour stocker les descriptions enrichies des activités et points d'intérets sous forme de documents JSON, flexibles et extensibles.
- **Roads Service** modélise les villes et leurs connexions routières dans **Neo4j**, chaque relation 'ROAD_TO' contenant des informations de distance et de durée de trajet.
- **Trip Service** est chargé de la gestion des voyages. Chaque document dans MongoDB représente un voyage complet avec ses étapes, activités, hébergements et étapes suivantes.
- **Docker** est utilisé pour conteneuriser tous les services, avec **Docker Compose** pour orchestrer les déploiements locaux.

Bilan du projet

Ce projet nous a permis de mettre en pratique de manière concrète plusieurs notions clés abordées en cours, notamment le développement de microservices et l'utilisation d'outils de conteneurisation.

Nous avons particulièrement apprécié le fait de manipuler docker et docker-compose avec spring-boot, le fait de pouvoir combiner plusieurs technologies (Spring Boot, MongoDB, PostgreSQL, Neo4j) avec plusieurs libraires comme Eureka, Spring-data-jpa spring-web, ceci a été enrichissant mais le vrai plus reste l'expérience en conteneurisation que ne n'avions pas eu l'opportunité de pratiquer auparavant.

En somme, ce projet nous a permis de :

- Comprendre en profondeur l'architecture microservices (modularité, découplage, communication via REST).
- Déployer et orchestrer des services avec Docker et Docker Compose.
- Travailler avec différentes bases de données selon les besoins métiers (NoSQL pour les documents, SQL pour les entités classiques, graphe pour les relations de distance).
- Mettre en place une architecture flexible et évolutive.

Ce que nous avons moins aimé

Le temps initial d'installation et de configuration des outils (Docker, Spring Cloud) a été relativement long. De plus, certaines incompatibilités ou erreurs de dépendance ont pu ralentir le développement.

La documentation de certaines librairies ou connecteurs (notamment pour Neo4j avec Spring) aurait pu être plus claire pour un déploiement fluide.

Réussites et difficultés

Réussites :

- L'application fonctionne correctement avec un routage centralisé via la Gateway.
- Chaque microservice est indépendant, connecté à sa propre base, et interrogeable.
- La collection `trip` modélise de manière expressive un voyage complet en MongoDB.

Difficultés rencontrées :

- La compréhension et la mise en œuvre du service discovery avec Eureka au début du projet.
- La modélisation des étapes du voyage (`steps`) dans un seul document JSON imbriqué.
- La gestion de la cohérence entre plusieurs sources de données (PostgreSQL / MongoDB / Neo4j).
- La conteneurisation et la configuration de docker-compose.