Rapport de Projet - Architecture Microservices

Gestion de Points d'Intérêt pour la Planification de Voyages

Master 2 MIAGE-IF Apprentissage 2024-2025

Projet Architecture Microservices – M. Menceur

Binôme :

- DOUBABI Mustapha
- HADDAD Chirine

Date : Juin 2025

---

1. Instructions de Compilation et d'Exécution

1.1 Prérequis

- **Java 17** ou supérieur
- **Maven 3.8+**
- **Docker** et **Docker Compose**
- **Git**

1.2 Démarrage Rapide

```
# 1. Cloner le repository
git clone [URL_REPOSITORY]
cd MSA

# 2. Compilation globale
mvn clean install

# 3. Démarrage avec Docker Compose
```

```
docker-compose up --build

# 4. Vérification des services
curl http://localhost:8080/api/health
```

1.3 Démarrage Manuel (Développement)

```
# Terminal 1 - City Service
cd city-service
mvn spring-boot:run

# Terminal 2 - Tourism Service
cd tourism-service
mvn spring-boot:run

# Terminal 3 - Travel Service
cd travel-service
mvn spring-boot:run

# Terminal 4 - Gateway Service
cd gateway-service
mvn spring-boot:run
```

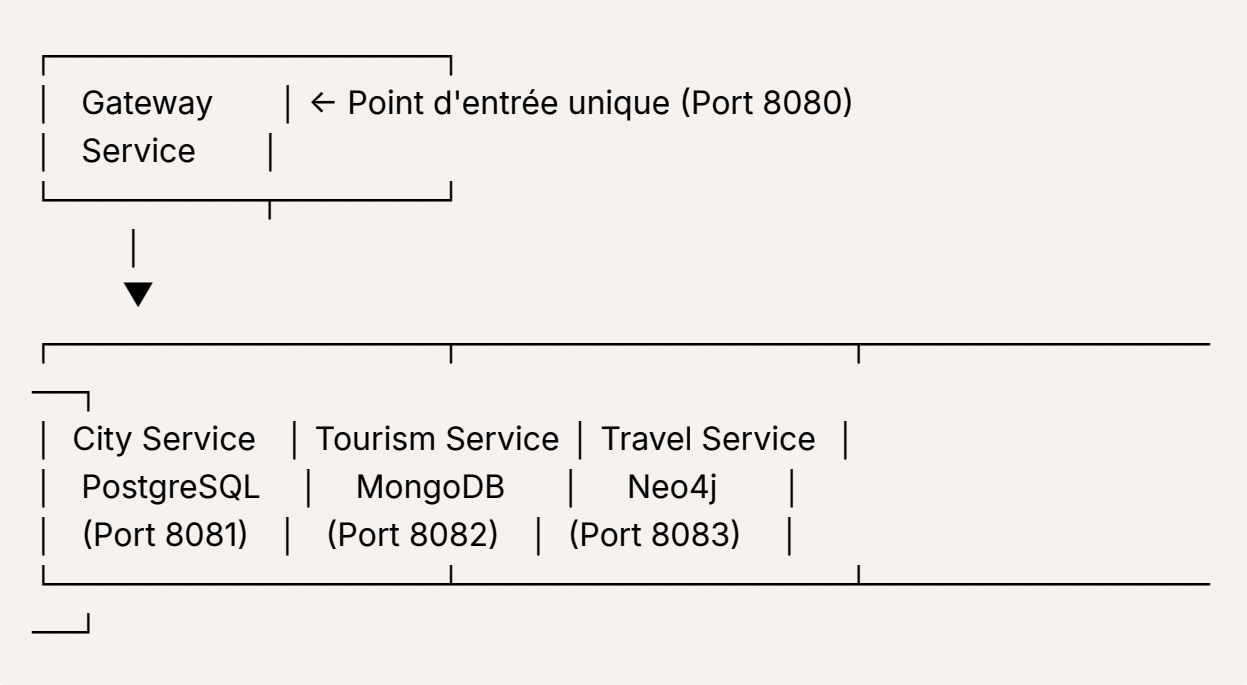
1.4 URLs d'Accès

- **Gateway :** http://localhost:8080
- **City Service :** http://localhost:8081
- **Tourism Service :** http://localhost:8082
- **Travel Service :** http://localhost:8083

---

2. Documentation Technique

2.1 Architecture Globale



2.2 Choix Techniques par Service

City Service (PostgreSQL + JSON)

- **Responsabilité :** Gestion des villes et informations géographiques
- **Choix technique :** PostgreSQL avec colonnes JSONB pour les coordonnées GPS
- **Justification :** Calculs géographiques complexes (distance Haversine) + structure hybride relationnelle/NoSQL

Tourism Service (MongoDB)

- **Responsabilité :** Points d'intérêt, activités, hébergements
- **Choix technique :** MongoDB avec documents flexibles
- **Justification :** Schémas variables selon le type d'activité + recherches par critères multiples

Travel Service (Neo4j)

- **Responsabilité :** Planification de voyages et relations entre entités
- **Choix technique :** Base de données graphe Neo4j

- **Justification :** Modélisation des relations complexes voyage-ville-activité + requêtes de cheminement

Gateway Service

- **Responsabilité :** Point d'entrée unique et routage
- **Choix technique :** Spring Boot + RestTemplate
- **Justification :** Simplicité, gestion CORS centralisée, monitoring unifié

2.3 Modèle de Données

City Service - Modèle Relationnel + JSON

```
@Entity
public class City {
    @Id @GeneratedValue
    private Long id;
    private String cityName;

    @JdbcTypeCode(SqlTypes.JSON)
    private GeographicInfo geographicInfo; // {latitude, longitude, country}
```

Tourism Service - Documents MongoDB

```
@Document(collection = "points_of_interest")
public class PointOfInterest {
    private String id;
    private String name;
    private String cityName;
    private String type;
    private GeographicInfo geographicInfo;
}

@Document(collection = "activities")
public class Activity {
```

```
    private String id;
    private String name;
    private List<String> pointOfInterestIds;
    private List<Integer> availableMonths; // 1-12 NUMÉRO DE MOIS
    private Double price;
    private Integer durationMinutes;
}
```

Travel Service - Graphe Neo4j

```
@Node("Travel")
public class Travel {
    @Id @GeneratedValue
    private Long id;
    private String travelName;
    private LocalDate startDate, endDate;

    @Relationship(type = "HAS_DAY", direction = OUTGOING)
    private List<TravelDay> travelDays;
}

@Node("TravelDay")
public class TravelDay {
    @Id @GeneratedValue
    private Long id;
    private LocalDate date;
    private String accommodationId; // Référence vers Tourism Service
    private List<String> plannedActivityIds;
}
```

2.4 Communication Inter-Services

- **Synchrone :** RestTemplate pour validation des données (ex: vérifier qu'une ville existe)
- **Découplage :** Chaque service peut fonctionner indépendamment

- **Références :** IDs cross-service (cityId, accommodationId, activityIds)

---

3. Bilan du Projet

3.1 Ce que nous avons aimé

Le développement backend pur : Travailler exclusivement sur la logique métier et les APIs REST sans se soucier de l'interface utilisateur a été très enrichissant. Cela nous a permis de nous concentrer sur l'architecture et la qualité du code.

L'écosystème Spring Boot : Le framework se charge automatiquement de nombreuses configurations (auto-configuration, gestion des dépendances, serveur embarqué). La productivité est impressionnante.

L'approche Repository : Les interfaces Spring Data simplifient énormément l'accès aux données. Écrire findByCityNameIgnoreCase(String cityName) et avoir automatiquement la requête générée est magique.

Découverte d'outils professionnels :

- **Postman :** Maîtrise des tests d'APIs, collections, variables d'environnement
- **Docker :** Containerisation et orchestration avec docker-compose
- **Gateway Pattern :** Développement d'un point d'entrée unique pour les microservices

3.2 Ce que nous avons appris

- **Architecture microservices :** Découpage fonctionnel, communication inter-services, gestion de la cohérence
- **Bases NoSQL hétérogènes :** PostgreSQL JSON, MongoDB documents, Neo4j graphes
- **Patterns avancés :** Repository, DTO, Gateway, suppression en cascade
- **DevOps de base :** Containerisation, health checks, configuration multi-environnement
- **Requêtes complexes :** Cypher Neo4j, agrégations MongoDB, fonctions géographiques PostgreSQL

3.3 Ce que nous avons moins aimé

Problèmes techniques “spéciaux” :

- **Sérialisation Jackson :** Configuration des dates LocalDate qui nécessitait des annotations spécifiques et configuration YAML pour éviter les timestamps
- **ID Neo4j :** Le passage de t.id vers ID(t) dans les requêtes Cypher qui a nécessité de refactoriser tous les repositories
- **Gestion des erreurs :** Debugging des communications inter-services avec RestTemplate

Ces problèmes, bien que formateurs, ont pris beaucoup de temps sur des détails techniques plutôt que sur la logique métier.

3.4 Réussites

Projet fonctionnel complet :

- 4 microservices opérationnels
- Base de données polyglotte fonctionnelle
- Toutes les requêtes NoSQL implémentées
- Gateway avec routage automatique
- Containerisation Docker réussie
- Tests unitaires et d'intégration

Architecture cohérente : Respect des principes microservices avec séparation claire des responsabilités et communication bien définie.

3.5 Difficultés rencontrées

Absence de frontend : Le projet ne dispose pas d'interface utilisateur. Tous les tests se font via Postman ou curl. Cela limite la démonstration des fonctionnalités et l'expérience utilisateur finale.

Courbe d'apprentissage : Maîtriser simultanément 3 types de bases NoSQL différentes a demandé un investissement important en documentation.

Débogage distribué : Identifier les erreurs dans un système distribué (4 services) est complèxe.

4. Conclusion

Ce projet nous a permis de mettre en pratique une architecture microservices complète avec une base de données polyglotte. L'expérience a été très formatrice sur les aspects techniques (Spring Boot, NoSQL, Docker) et architecturaux (découpage fonctionnel, communication inter-services).

Le résultat final est un système robuste et extensible qui répond à tous les besoins fonctionnels exprimés. L'ajout futur d'un frontend React ou Angular permettrait de compléter l'écosystème pour une solution end-to-end.

Technologies maîtrisées : Spring Boot, PostgreSQL, MongoDB, Neo4j, Docker, Maven, Postman

Patterns implémentés : Microservices, Repository, Gateway, DTO

Lignes de code : ~3000 (Java + configuration)