Dauphine | PSL
UNIVERSITÉ PARIS

Rapport Projet Architecture Microservices

Jean-Baptiste COLLIN & Julien MARCILLAUD

Juin 2024

Sommaire

1. Indications pour compiler / exécuter le projet

Pour compiler et exécuter le projet, suivez les étapes suivantes :

2. **Prérequis:**

    - JDK 8 ou supérieur
    - Maven 3.6 ou supérieur
    - Docker et Docker Compose
    - MySQL

2. Cloner le dépôt:

```
git clone
https://github.com/jecollin/Projet_Architecture_Microservices_JO.git
cd Projet_Architecture_Microservices_JO
```

3. Construire les microservices:

```
cd ../Event_Scheduling_Service
mvn clean install
cd ../Sites_Management_Service
mvn clean install
cd ../Sports_Management_Service
mvn clean install
cd ../User_Planning_Service
mvn clean install
```

4. **Lancer les services avec Docker Compose:**

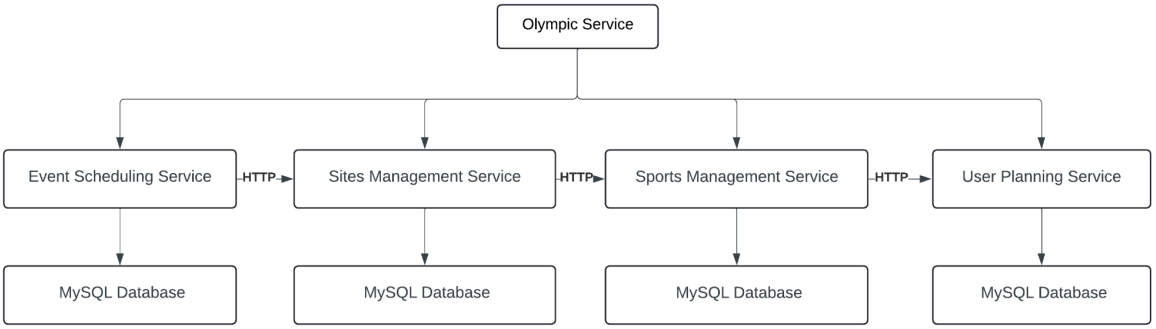
```
docker-compose up --build
```

5. **Accéder aux services:**

- Les services seront disponibles sur `localhost` avec les ports définis dans le fichier `docker-compose.yml`.
- Par exemple, le service de gestion des événements sera disponible sur http://localhost:8081.

2. Documentation Technique

2.1 Schéma d’architecture



2.2 Choix techniques

Frameworks

Spring Boot pour le développement des microservices :

- **Description :** Spring Boot est un framework open-source basé sur Spring Framework, conçu pour simplifier le processus de création d'applications Java autonomes et prêtes pour la production. Il permet de démarrer rapidement des projets avec des configurations par défaut.
- **Pourquoi ce choix :**
    - **Facilité d'utilisation :** Spring Boot simplifie la configuration initiale et réduit le temps nécessaire pour mettre en place un projet.
    - **Convention sur configuration :** Avec des configurations par défaut sensées, Spring Boot permet aux développeurs de se concentrer sur l'écriture de logique métier plutôt que sur la configuration de l'infrastructure.
    - **Écosystème riche :** Il bénéficie de l'écosystème mature et étendu de Spring, offrant une intégration facile avec de nombreux autres outils et bibliothèques.
    - **Microservices :** Idéal pour la création de microservices grâce à son support natif pour les architectures de microservices et ses capacités de conteneurisation et de déploiement faciles.

Spring Data JPA pour l'accès aux données :

- **Description :** Spring Data JPA est un module de Spring qui facilite l'implémentation des couches d'accès aux données en utilisant JPA (Java Persistence API). Il offre des abstractions pour simplifier les interactions avec les bases de données relationnelles.
- **Pourquoi ce choix :**
    - **Abstraction de l'accès aux données :** Spring Data JPA réduit la complexité de la gestion des opérations CRUD (Create, Read, Update, Delete) en offrant des abstractions de haut niveau.
    - **Génération automatique de requêtes :** Grâce à des conventions de nommage, il génère automatiquement des requêtes SQL, ce qui permet de gagner du temps et de réduire les erreurs.
    - **Support pour différentes bases de données :** Spring Data JPA est compatible avec de nombreuses bases de données relationnelles, offrant ainsi une flexibilité dans le choix de la base de données.

Spring Cloud OpenFeign pour la communication entre microservices :

- **Description :** Spring Cloud OpenFeign est une bibliothèque qui permet de simplifier les appels aux API REST entre microservices en utilisant des interfaces Java annotées.
- **Pourquoi ce choix :**
    - **Simplification des appels REST :** OpenFeign offre une manière déclarative de faire des appels HTTP, simplifiant ainsi le code nécessaire pour la communication interservices.
    - **Intégration avec Spring Boot :** Il s'intègre parfaitement avec Spring Boot, offrant des fonctionnalités comme la gestion des exceptions, la tolérance aux pannes et la gestion des configurations.
    - **Maintenance et évolutivité :** La configuration déclarative facilite la maintenance et l'ajout de nouveaux appels API, améliorant ainsi l'évolutivité du système.

Base de données

MySQL pour le stockage des données :

- **Description :** MySQL est un système de gestion de base de données relationnelle open-source très utilisé dans l'industrie.
- **Pourquoi ce choix :**
    - **Performance et fiabilité :** MySQL est réputé pour sa performance, sa stabilité et sa fiabilité dans la gestion des bases de données de production.
    - **Communauté et support :** Il dispose d'une vaste communauté et d'un support étendu, ce qui facilite la résolution des problèmes et l'accès aux ressources.
    - **Compatibilité :** MySQL est compatible avec une multitude de frameworks et de langages de programmation, y compris Spring Boot et Spring Data JPA.

H2 pour les tests :

- **Description :** H2 est une base de données relationnelle Java en mémoire. Elle est principalement utilisée pour les tests et le développement.
- **Pourquoi ce choix :**
    - **Légèreté et rapidité :** H2 est une base de données légère et rapide, idéale pour les tests unitaires et d'intégration.
    - **Mode en mémoire :** Le mode en mémoire permet d'initialiser rapidement la base de données pour chaque test et de la détruire après, garantissant un environnement propre pour chaque exécution de test.
    - **Compatibilité avec JPA :** H2 est entièrement compatible avec JPA, permettant ainsi de tester les mêmes entités et configurations que celles utilisées avec MySQL en production.

Conteneurisation

Docker pour conteneuriser les microservices :

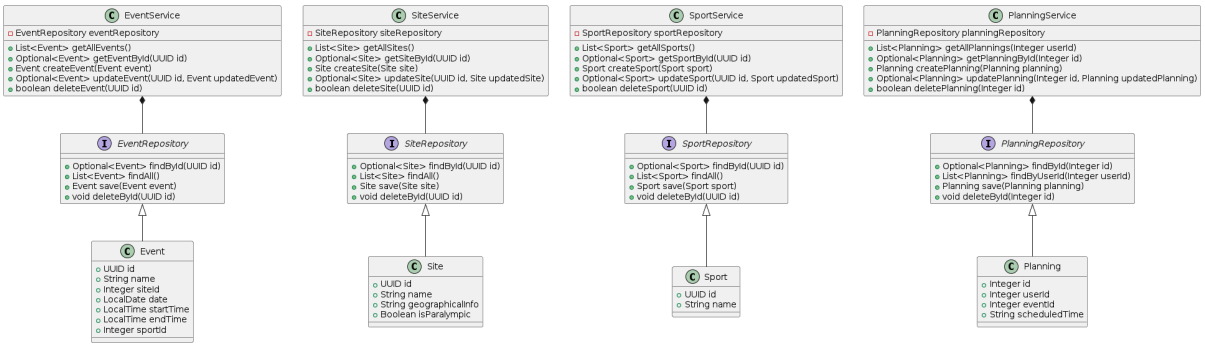
- **Description :** Docker est une plateforme de conteneurisation qui permet de créer, déployer et gérer des conteneurs pour des applications.
- **Pourquoi ce choix :**
    - **Isolation des environnements :** Docker permet de créer des environnements isolés pour chaque microservice, assurant une exécution indépendante et minimisant les conflits.
    - **Portabilité :** Les conteneurs Docker peuvent être exécutés sur n'importe quel environnement supportant Docker, offrant ainsi une grande portabilité.
    - **Consistance :** Assure que l'application fonctionne de la même manière sur tous les environnements (développement, test, production).

Docker Compose pour orchestrer les services :

- **Description :** Docker Compose est un outil de Docker qui permet de définir et de gérer des applications multi-conteneurs. Il utilise un fichier YAML pour configurer les services.
- **Pourquoi ce choix :**
    - **Simplification de l'orchestration :** Docker Compose facilite le déploiement et la gestion de plusieurs conteneurs en définissant simplement les services, les réseaux et les volumes nécessaires.
    - **Configuration centralisée :** Toutes les configurations sont centralisées dans un fichier docker-compose.yml, ce qui simplifie la gestion et le déploiement.
    - **Automatisation des déploiements :** Docker Compose permet d'automatiser le démarrage, l'arrêt et la gestion des dépendances entre les services, améliorant ainsi l'efficacité et la fiabilité du déploiement.

Ces choix techniques ont été faits pour garantir une architecture robuste, évolutive et facilement maintenable pour le projet de gestion des microservices olympiques.

2.3 Diagrammes de classes



3. Bilan du projet

3.1 Ce que nous avons aimé

Apprendre à structurer une application en microservices :

- **Expérience enrichissante :** La structuration d'une application en microservices nous a permis de comprendre les avantages de cette architecture, notamment en termes de modularité, de scalabilité et de maintenance.
- **Clarté et organisation :** Nous avons apprécié comment la séparation des préoccupations en différents services distincts a rendu le code plus clair et plus facile à gérer.
- **Approche décentralisée :** Cette architecture nous a permis de concevoir des services indépendants pouvant être développés, déployés et mis à l'échelle de manière autonome.

Utilisation de Spring Boot et des différents modules Spring Cloud :

- **Facilité d'utilisation :** Spring Boot nous a fourni une base solide et facile à utiliser pour démarrer rapidement avec les microservices.
- **Modules Spring Cloud :** L'intégration des modules Spring Cloud (comme Spring Cloud OpenFeign, Spring Cloud Config, etc.) a simplifié la gestion de la communication interservices, de la configuration et des autres aspects complexes des microservices.
- **Documentation et communauté :** La riche documentation et la vaste communauté de Spring ont été d'une grande aide pour résoudre les problèmes et apprendre de nouvelles fonctionnalités.

Expérience avec Docker et Docker Compose pour déployer les services :

- **Isolation et portabilité :** Docker a facilité l'isolation des services et leur portabilité entre différents environnements, assurant ainsi que les services fonctionnent de manière cohérente partout.
- **Orchestration simplifiée :** Docker Compose nous a permis de définir et de gérer facilement l'ensemble de notre architecture de microservices à l'aide d'un simple fichier de configuration YAML.
- **Déploiement rapide :** Le processus de déploiement a été grandement accéléré grâce à la conteneurisation, rendant les mises à jour et les tests plus efficaces.

3.2 Ce que nous avons appris

La communication entre microservices via FeignClient :

- **Appels REST simplifiés :** FeignClient a rendu la communication entre microservices très simple et intuitive, en utilisant des interfaces Java et des annotations pour définir les clients REST.
- **Tolérance aux pannes et repli :** Nous avons appris à configurer des mécanismes de tolérance aux pannes et des stratégies de repli pour assurer la résilience de nos services.

Gestion de la configuration et des propriétés d'application :

- **Spring Cloud Config :** Nous avons utilisé Spring Cloud Config pour externaliser les configurations des microservices, facilitant ainsi la gestion centralisée et la mise à jour des configurations sans redéployer les services.
- **Propriétés dynamiques :** La possibilité de modifier dynamiquement les propriétés de l'application a été très utile pour adapter les services à différents environnements (développement, test, production).

Création et exécution de tests unitaires et d'intégration :

- **JUnit et Mockito :** Nous avons utilisé JUnit et Mockito pour écrire des tests unitaires, assurant ainsi la qualité et la fiabilité du code.
- **Tests d'intégration :** L'exécution de tests d'intégration avec une base de données H2 en mémoire a permis de vérifier l'intégration et le bon fonctionnement des services de bout en bout.

3.3 Ce que nous avons moins aimé

Les défis liés à la configuration de Docker et Docker Compose :

- **Courbe d'apprentissage :** La courbe d'apprentissage pour maîtriser Docker et Docker Compose a été plus abrupte que prévu, notamment en ce qui concerne la gestion des réseaux, des volumes et des dépendances interconteneurs.
- **Complexité des configurations :** La création et la gestion de fichiers Docker et Docker Compose complexes ont parfois été source de frustration, notamment lors de la résolution de conflits et de la gestion des versions des images.

La complexité de débogage des microservices en développement :

- **Débogage distribué :** Le débogage d'une architecture de microservices s'est avéré plus complexe que celui d'une application monolithique, en raison de la nature distribuée des services et de la nécessité de suivre les appels entre plusieurs services.
- **Logs et traçabilité :** La gestion des logs et la traçabilité des requêtes à travers les services ont nécessité la mise en place de solutions supplémentaires, comme des outils de centralisation des logs et des traceurs distribués.

3.4 Réussites / Difficultés

Réussites :

Déploiement réussi des microservices avec Docker Compose :

- **Conteneurisation et déploiement :** Nous avons réussi à conteneuriser et à déployer l'ensemble de notre architecture de microservices à l'aide de Docker Compose, ce qui a considérablement simplifié le déploiement et la gestion des services.
- **Orchestration efficace :** Le fichier docker-compose.yml a permis d'orchestrer efficacement les services, les réseaux et les volumes nécessaires.

Intégration réussie de Spring Cloud OpenFeign pour la communication interservices :

- **Clients REST déclaratifs :** Nous avons intégré avec succès Spring Cloud OpenFeign pour simplifier la communication entre les microservices, en définissant des interfaces de clients REST déclaratifs.
- **Maintenabilité améliorée :** Cette intégration a permis de réduire la complexité du code et d'améliorer la maintenabilité des services.

Difficultés :

Gestion des dépendances et des versions compatibles entre les différentes bibliothèques :

- **Compatibilité des versions :** Nous avons rencontré des difficultés pour assurer la compatibilité des versions entre les différentes bibliothèques Spring, notamment lors de l'intégration de modules Spring Cloud.
- **Résolution de conflits :** La résolution des conflits de dépendances et la gestion des versions ont nécessité une attention particulière et des ajustements constants.

Configuration et optimisation des tests unitaires et d'intégration :

- **Complexité des tests :** La configuration des tests unitaires et d'intégration pour les microservices a été complexe, notamment en raison de la nécessité de simuler les interactions entre services et de configurer des bases de données en mémoire pour les tests.
- **Optimisation des tests :** L'optimisation des tests pour garantir des temps d'exécution raisonnables et des résultats fiables a été un défi constant, nécessitant des ajustements et des améliorations continues.