ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR
Membre de HONORIS UNITED UNIVERSITIES

Ingénierie Informatique et Réseaux MIAGE

École Marocaine des Sciences de l'Ingénieur

RAPPORT

Thème

Système de gestion de cinéma basé sur les microservices utilisant Spring Boot, Spring Cloud, Eureka, Openfeign, Zipkin, Postgres et Docker

Réalisé par : BELKAMEL Mouad - LOUKILI IDRISSI Oussama

Encadrant Académique : Mr LACHGAR Mohamed

L'année Universitaire : 2023/2024

*

Résumé

Dans le cadre de ce projet académique, nous avons développé un système de gestion de cinéma basé sur l'architecture des microservices en utilisant des technologies telles que Spring Boot, Spring Cloud, Eureka, OpenFeign, Zipkin, PostgreSQL et Docker. L'objectif était de créer une application modulaire et évolutive, divisée en services indépendants mais interconnectés. Nous avons réalisé des microservices pour la gestion des films, des salles de cinéma et des réservations, en utilisant Spring Boot et Spring Cloud pour la gestion des microservices, Eureka pour la découverte des services, OpenFeign pour la communication entre les microservices, et Zipkin pour le traçage des performances. PostgreSQL a été utilisé comme base de données, et Docker a facilité le déploiement et la gestion de l'application. Ce projet nous a permis d'approfondir nos connaissances sur les microservices et de développer des compétences pratiques dans la mise en œuvre de cette architecture.

Chapitre 1 :
Introduction

1.1 Aperçu du projet :

Le projet consistait à développer un système de gestion de cinéma basé sur l'architecture des microservices. Nous avons utilisé des technologies telles que Spring Boot, Spring Cloud, Eureka, OpenFeign, Zipkin, PostgreSQL et Docker. L'objectif était de créer une application modulaire et évolutive, avec des services indépendants mais interconnectés pour la gestion des films, des salles de cinéma et des réservations. Nous avons utilisé Spring Cloud pour la gestion des microservices, Eureka pour la découverte des services, OpenFeign pour la communication entre les microservices, Zipkin pour le traçage des performances, PostgreSQL comme base de données, et Docker pour faciliter le déploiement. Le projet nous a permis d'approfondir notre compréhension des microservices et de développer des compétences pratiques dans leur mise en œuvre.

1.2 Importance de l'architecture microservices :

Les microservices jouent un rôle essentiel dans le développement d'applications modernes et évolutives. Leur importance réside dans plusieurs aspects clés :

- Modularité et scalabilité : Les microservices permettent de diviser une application en petits services autonomes et indépendants, chacun se concentrant sur une fonctionnalité spécifique. Cette approche favorise la modularité, ce qui facilite le développement, la maintenance et les mises à jour de l'application. De plus, chaque microservice peut être déployé et mis à l'échelle indépendamment selon les besoins, ce qui améliore la flexibilité et la scalabilité de l'application.
- Développement agile : Les microservices favorisent le développement agile en permettant à différentes équipes de travailler simultanément sur des services distincts. Cela permet d'accélérer le processus de développement, d'itérer rapidement et de mettre en production des fonctionnalités plus rapidement. Chaque équipe peut se concentrer sur son microservice spécifique, ce qui facilite la collaboration et l'innovation.

- Facilité de déploiement et de gestion : Grâce à leur nature indépendante, les microservices peuvent être déployés et gérés de manière autonome. Cela simplifie les processus de déploiement, de mise à jour et de maintenance de l'application. De plus, l'utilisation de technologies telles que les conteneurs (comme Docker) facilite la gestion des microservices et permet une meilleure isolation des ressources.
- Résilience et tolérance aux pannes : Les microservices sont conçus pour être résilients et tolérants aux pannes. Si l'un des microservices échoue, les autres peuvent continuer à fonctionner sans interruption. Cela garantit une meilleure disponibilité de l'application globale et réduit les risques d'impact majeur en cas de problème.
- Intégration et évolutivité : Les microservices facilitent l'intégration avec d'autres systèmes et services externes. Chaque microservice peut être développé dans un langage ou un framework différent, selon les besoins spécifiques. De plus, grâce à leur nature modulaire, les microservices permettent une évolutivité plus granulaire, en ne mettant à l'échelle que les services nécessaires, ce qui optimise l'utilisation des ressource

Chapitre 2 :
Architecture Microservices

2.1 Architecture :

L'architecture du système de gestion de cinéma repose sur l'approche des microservices. Cela signifie que l'application est divisée en plusieurs services indépendants qui sont responsables de fonctionnalités spécifiques. Chaque microservice est autonome et peut être développé, déployé et mis à l'échelle de manière indépendante. Les microservices sont interconnectés pour permettre une communication et une collaboration harmonieuses entre eux. L'utilisation de l'architecture des microservices favorise la modularité, la flexibilité et la scalabilité de l'application.

2.2 Description des services :

Dans le système de gestion de cinéma, nous avons deux principaux le service Cinema et le service Client :

Service Cinema :

Le service Cinema est responsable de la gestion des informations liées aux salles de cinéma. Il offre des fonctionnalités telles que la création, la modification et la suppression des salles de cinéma. Ce service maintient une base de données des salles disponibles, y compris des détails tels que le nom de la salle, sa capacité, l'emplacement, les horaires de projection, etc. Les utilisateurs peuvent interagir avec ce service pour rechercher des salles, consulter les horaires de projection et effectuer des réservations.

Service Client :

Le service Client gère les informations relatives aux clients du cinéma. Il permet aux utilisateurs de créer des comptes clients, de gérer leurs informations personnelles, leurs réservations passées, etc. Les utilisateurs peuvent s'inscrire, se connecter et mettre à jour leurs données personnelles via ce service. De plus, le service Client peut offrir des fonctionnalités telles que la récupération des informations de réservation, l'historique des transactions et la gestion des préférences des utilisateurs

2.3 Mécanismes de communication :

Pour permettre la communication entre les microservices, nous avons utilisé plusieurs mécanismes. Tout d'abord, nous avons utilisé Spring Cloud, un ensemble d'outils et de bibliothèques qui facilite la gestion des microservices. Spring Cloud fournit des fonctionnalités telles que la découverte de services, la gestion des configurations, la résilience, etc.

Pour la découverte de services, nous avons utilisé Eureka, qui permet d'enregistrer et de localiser les différents microservices. Cela permet à chaque microservice de trouver et de communiquer avec d'autres services dont il dépend.

La communication entre les microservices a été gérée à l'aide d'OpenFeign, une bibliothèque qui facilite l'appel de services distants de manière déclarative. OpenFeign simplifie l'écriture du code client en masquant les détails de la communication réseau.

Enfin, nous avons utilisé Zipkin, un outil de traçage distribué, pour surveiller et analyser les performances des microservices. Zipkin permet de suivre les appels entre les microservices et de visualiser les délais et les erreurs éventuelles.

Ces mécanismes de communication assurent une interaction fluide et fiable entre les différents microservices, contribuant ainsi à l'efficacité globale du système de gestion de cinéma.

Chapitre 3 :
Conception des microservices

1/ Approche de conception pour le service Cinema :

Pour concevoir le service Cinema, nous avons adopté une approche basée sur les principes de conception orientée objet et de modularité. Voici les principales considérations de conception pour ce service :

Modèle de données : Nous avons créé un modèle de données pour représenter les salles de cinéma, comprenant des entités telles que la salle elle-même, l'emplacement, la capacité, les horaires de projection, etc. Le modèle de données a été conçu de manière à être extensible, permettant l'ajout de fonctionnalités supplémentaires si nécessaire.

API REST : Nous avons exposé une API REST pour permettre l'interaction avec le service Cinema. Cette API offre des endpoints pour la création, la modification et la suppression des salles de cinéma, ainsi que des endpoints pour la recherche et la récupération des détails des salles. L'API est conçue de manière à être conviviale et intuitive pour les clients.

Séparation des responsabilités : Nous avons veillé à séparer les différentes responsabilités du service Cinema. Par exemple, la gestion des salles de cinéma est gérée par un module spécifique, tandis que la recherche et la récupération des détails des salles sont gérées par un autre module. Cette approche de séparation des responsabilités facilite la maintenance et l'évolutivité du service.

Intégration avec d'autres services : Le service Cinema peut avoir des dépendances avec d'autres services, tels que le service de gestion des films. Nous avons conçu des interfaces claires et des mécanismes d'intégration pour permettre une communication fluide et cohérente entre ces services. Cela garantit une expérience utilisateur transparente lors de la recherche de films et de la réservation de sièges.

1/ Entities :

```
package ma.imet.cinema.entities;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
```

```
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data @AllArgsConstructor @NoArgsConstructor @Builder
public class Cinema {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;


}
```

2/ Repository :

```
package ma.imet.cinema.repository;

import ma.imet.cinema.entities.Cinema;
import org.springframework.data.jpa.repository.JpaRepository;

public interface CinemaRepository extends JpaRepository<Cinema,Long> {
}
```

3/ Services :

```
package ma.imet.cinema.service;

import lombok.RequiredArgsConstructor;
import ma.imet.cinema.connection.FullCinemaResponse;
import ma.imet.cinema.connection.client.Clientcl;
import ma.imet.cinema.entities.Cinema;
import ma.imet.cinema.repository.CinemaRepository;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
@RequiredArgsConstructor
public class CinemaServiceImpl implements CinemaService {

    private final CinemaRepository cinemaRepository;
    private Clientcl clientcl;


    @Override
    public void saveCinema(Cinema cinema) {
        cinemaRepository.save(cinema);
    }

    @Override
    public List<Cinema> allCinemas() {
        return cinemaRepository.findAll();
    }

    @Override
    public FullCinemaResponse findCinemasWithClients(Long cinemaId) {
        var cinema = cinemaRepository.findById(cinemaId)
                .orElse(
                        Cinema.builder()
                                .name("NOT FOUND")
```

```
                                .email("NOT FOUND")
                                .build()
                );
        var clients = clientcl.findAllClientsByCinema(cinemaId);
        return FullCinemaResponse.builder()
                .name(cinema.getName())
                .email(cinema.getEmail())
                .clients(clients)
                .build();
    }
}
```

2/ Approche de conception pour le service Client :

Pour le service Client, nous avons adopté une approche centrée sur l'utilisateur et axée sur la gestion des informations personnelles et des réservations. Voici les aspects clés de la conception de ce service :

Modèle de données : Nous avons conçu un modèle de données pour représenter les informations relatives aux clients, telles que leurs informations personnelles, les réservations passées, les préférences, etc. Ce modèle de données est structuré de manière à être adapté à la gestion des utilisateurs et à permettre une personnalisation des services offerts.

Authentification et sécurité : Nous avons mis en place des mécanismes d'authentification pour permettre aux utilisateurs de créer des comptes clients et de se connecter en toute sécurité. Des protocoles d'authentification standard tels que OAuth peuvent être utilisés pour sécuriser l'accès aux informations personnelles des utilisateurs.

API REST : Le service Client expose une API REST pour permettre aux utilisateurs d'interagir avec leurs informations personnelles et leurs réservations. L'API offre des endpoints pour la création de comptes, la récupération des informations personnelles, la gestion des réservations, etc. L'API est conçue pour être facilement utilisable et intuitive pour les clients.

Gestion des préférences : Nous avons prévu des fonctionnalités de gestion des préférences pour permettre aux utilisateurs de personnaliser leur expérience cinématographique. Cela peut inclure des préférences telles que les genres de films préférés, les types de sièges, les notifications, etc. Le service Client permet aux utilisateurs de définir et de mettre à jour ces

préférences selon leurs besoins.

Intégration avec d'autres services : Le service Client peut également intégrer d'autres services, tels que le service de réservation ou le service de recommandation de films, pour offrir une expérience globale optimisée. Nous avons conçu des mécanismes d'intégration pour permettre une communication efficace et cohérente entre ces services.

1/ Entities :

```
package ma.imet.client.entities;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data @AllArgsConstructor @NoArgsConstructor @Builder
public class Client {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private Long cinemaId;


}
```

2/ Repository :

```
package ma.imet.client.repository;

import ma.imet.client.entities.Client;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface ClientRepository extends JpaRepository<Client,Long> {
    List<Client> findAllByCinemaId(Long cinemaId);
}
```

3/ Services :

```
package ma.imet.client.service;

import lombok.RequiredArgsConstructor;
import ma.imet.client.entities.Client;
import ma.imet.client.repository.ClientRepository;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
```

```java
@RequiredArgsConstructor
public class ClientServiceImpl implements ClientService {

    private final ClientRepository clientRepository;


    @Override
    public void saveClient(Client client) {
        clientRepository.save(client);
    }

    @Override
    public List<Client> allClients() {
        return clientRepository.findAll();
    }

    @Override
    public List<Client> allClientsByCinema(Long cinemaId) {
        return clientRepository.findAllByCinemaId(cinemaId);
    }
}
```

Chapitre 4 :
Conteneurisation avec Docker

La conteneurisation avec Docker est une approche couramment utilisée pour le déploiement de microservices. Elle offre de nombreux avantages en termes de facilité de déploiement, de gestion des ressources et de portabilité. Voici des informations sur l'implémentation et les avantages de la conteneurisation avec Docker pour les microservices :

Implémentation avec Docker :
L'implémentation des microservices avec Docker se fait en créant des conteneurs Docker pour chaque microservice. Un conteneur Docker est une unité légère et isolée qui contient tout ce dont un microservice a besoin pour s'exécuter, y compris son code, ses dépendances et son environnement d'exécution. Chaque conteneur est basé sur une image Docker, qui peut être créée à partir d'un fichier de configuration appelé Dockerfile.

Dans le cas des microservices, chaque microservice est généralement empaqueté dans un conteneur Docker distinct. Cela permet de gérer et de déployer chaque microservice de manière indépendante, sans affecter les autres microservices. Chaque conteneur peut être démarré, arrêté, mis à jour ou redéployé indépendamment des autres.

Avantages de la conteneurisation avec Docker pour les microservices :

Isolation : Les conteneurs Docker fournissent une isolation complète entre les microservices. Chaque microservice s'exécute dans son propre conteneur, ce qui évite les conflits de dépendances et les interférences entre les services. Cela garantit une stabilité et une fiabilité accrues de l'ensemble du système.

Portabilité : Les conteneurs Docker offrent une portabilité élevée. Ils encapsulent tout ce dont un microservice a besoin pour s'exécuter, y compris ses dépendances et son environnement d'exécution. Cela signifie qu'un conteneur Docker créé sur un système peut être facilement déployé sur un autre système, qu'il s'agisse d'un environnement de développement, de test ou de production. Cela facilite la mise en place d'une infrastructure de déploiement cohérente à travers différents environnements.

Facilité de déploiement et de mise à l'échelle : Docker simplifie considérablement le déploiement des microservices. Les conteneurs Docker peuvent être facilement déployés sur des hôtes physiques, des machines virtuelles ou des clusters de conteneurs. De plus, Docker offre des outils pour orchestrer et gérer les conteneurs à grande échelle, tels que Docker Swarm et Kubernetes. Cela facilite le déploiement et la mise à l'échelle des microservices, en permettant une gestion centralisée et automatisée.

Gestion des ressources : Docker permet une gestion efficace des ressources. Chaque conteneur Docker peut être configuré avec des limites de ressources spécifiques, telles que la mémoire et le CPU, afin de garantir une utilisation équilibrée des ressources du système. Cela permet d'optimiser l'utilisation des ressources et d'éviter les problèmes de surcharge ou de sous-utilisation.

Facilité de collaboration et de développement : Docker facilite la collaboration entre les développeurs travaillant sur des microservices. Chaque développeur peut créer son propre conteneur Docker pour son microservice, en utilisant les mêmes dépendances et configurations. Cela garantit une cohérence et une reproductibilité des environnements de développement, ce qui facilite le partage du code et la collaboration entre les équipes.

```yaml
version: '3'

services:
  postgres:
    image: postgres:latest
    container_name: my-postgres-container-sec
    environment:
      POSTGRES_DB: clients
      POSTGRES_USER: username
      POSTGRES_PASSWORD: password
    ports:
      - "5431:5432"
```

```yaml
version: '3'

services:
  postgres:
    image: postgres:latest
    container_name: my-postgres-container
    environment:
      POSTGRES_DB: cinemas
      POSTGRES_USER: mouad
      POSTGRES_PASSWORD: 1234
    ports:
      - "5432:5432"
```

Chapitre 5 :
CI/CD Jenkins

CI/CD (Intégration Continue / Livraison Continue) est une pratique de développement logiciel qui vise à automatiser les étapes de construction, de test et de déploiement d'une application. Jenkins est un outil populaire utilisé pour mettre en place des pipelines CI/CD. Voici les informations sur le processus et la configuration de CI/CD avec Jenkins :

Processus de CI/CD avec Jenkins :

Intégration Continue (CI) :

Le processus commence par la récupération du code source à partir d'un référentiel de contrôle de version tel que Git.

Jenkins surveille les modifications du référentiel et déclenche automatiquement le processus de construction lorsque des changements sont détectés.

La construction consiste à compiler le code, à résoudre les dépendances, à exécuter les tests unitaires et à générer des artefacts, tels que des fichiers binaires ou des packages d'installation.

Livraison Continue (CD) :

Une fois la construction réussie, Jenkins déploie l'application dans un environnement de test ou de pré-production.

Dans cet environnement, des tests supplémentaires, tels que des tests d'intégration, des tests de performance ou des tests de sécurité, peuvent être exécutés.

Si les tests réussissent, Jenkins peut procéder au déploiement de l'application dans l'environnement de production.

Déploiement continu :

Le déploiement continu est une étape facultative du processus de CI/CD. Il s'agit de mettre en place des mécanismes automatiques pour déployer l'application dans l'environnement de production sans intervention manuelle.

Configuration de CI/CD avec Jenkins :

La configuration de CI/CD avec Jenkins implique la création d'un pipeline qui définit les étapes à suivre pour la construction, le test et le déploiement de l'application. Voici les

aspects clés de la configuration :

Configuration du référentiel :

Jenkins doit être configuré pour surveiller le référentiel de contrôle de version (par exemple, Git) et détecter les changements.

Les informations d'authentification et les détails de connexion au référentiel doivent être configurés dans Jenkins.

Configuration du pipeline :

Le pipeline est configuré à l'aide d'un fichier de configuration appelé Jenkinsfile.

Le Jenkinsfile définit les étapes du pipeline, telles que la construction, les tests et le déploiement, ainsi que les conditions de déclenchement pour chaque étape.

Les outils et les commandes spécifiques pour chaque étape sont définis dans le Jenkinsfile.

Configuration des environnements :

Jenkins doit être configuré pour déployer l'application dans des environnements spécifiques, tels que des serveurs de test ou de production.

Les détails d'authentification et les paramètres de déploiement spécifiques à chaque environnement doivent être configurés dans Jenkins.

Configuration des notifications :

Jenkins peut être configuré pour envoyer des notifications par e-mail, des messages instantanés ou d'autres canaux de communication pour informer les équipes de développement des résultats du pipeline.

Les notifications peuvent être configurées pour les étapes de construction réussies, les échecs de test, les déploiements réussis, etc.

Conclusion générale et perspectives :

Au cours de cette discussion, nous avons exploré deux sujets importants dans le domaine du développement logiciel : la conteneurisation avec Docker pour les microservices et la mise en place de CI/CD avec Jenkins. Voici un résumé des accomplissements et des perspectives futures pour ces deux domaines :

Conteneurisation avec Docker pour les microservices :

Nous avons compris que la conteneurisation avec Docker offre une isolation, une portabilité et une facilité de déploiement accrues pour les microservices.

En utilisant Docker, chaque microservice peut être empaqueté dans un conteneur distinct, facilitant ainsi la gestion et le déploiement indépendant de chaque service.

Les avantages clés de la conteneurisation avec Docker pour les microservices incluent une meilleure isolation, une portabilité élevée, une facilité de déploiement et de mise à l'échelle, ainsi qu'une gestion efficace des ressources.

CI/CD avec Jenkins :

Nous avons appris que CI/CD avec Jenkins vise à automatiser les étapes de construction, de test et de déploiement d'une application.

Jenkins permet de mettre en place des pipelines qui suivent un processus continu, de l'intégration du code à la livraison de l'application.

La configuration de Jenkins implique la surveillance du référentiel de contrôle de version, la définition des étapes du pipeline, la configuration des environnements de déploiement et la mise en place de notifications pour informer les équipes de développement.

Perspectives futures :

Pour la conteneurisation avec Docker, nous pouvons nous attendre à des améliorations continues dans les outils et les technologies associées. De nouveaux développements pourraient inclure des solutions d'orchestration plus avancées, une sécurité renforcée des conteneurs et une intégration plus étroite avec les plateformes cloud.

Pour CI/CD avec Jenkins, nous pouvons envisager des améliorations dans l'automatisation des tests, l'intégration avec d'autres outils de développement, ainsi que des fonctionnalités d'analyse et de surveillance avancées pour les pipelines.

Dans l'ensemble, l'évolution des méthodes de développement logiciel continuera d'être axée sur l'automatisation, la collaboration et l'amélioration de l'efficacité des processus de développement et de déploiement. Les pratiques de conteneurisation et de CI/CD joueront un rôle essentiel dans cet objectif.