Optimisation des Communications Microservices

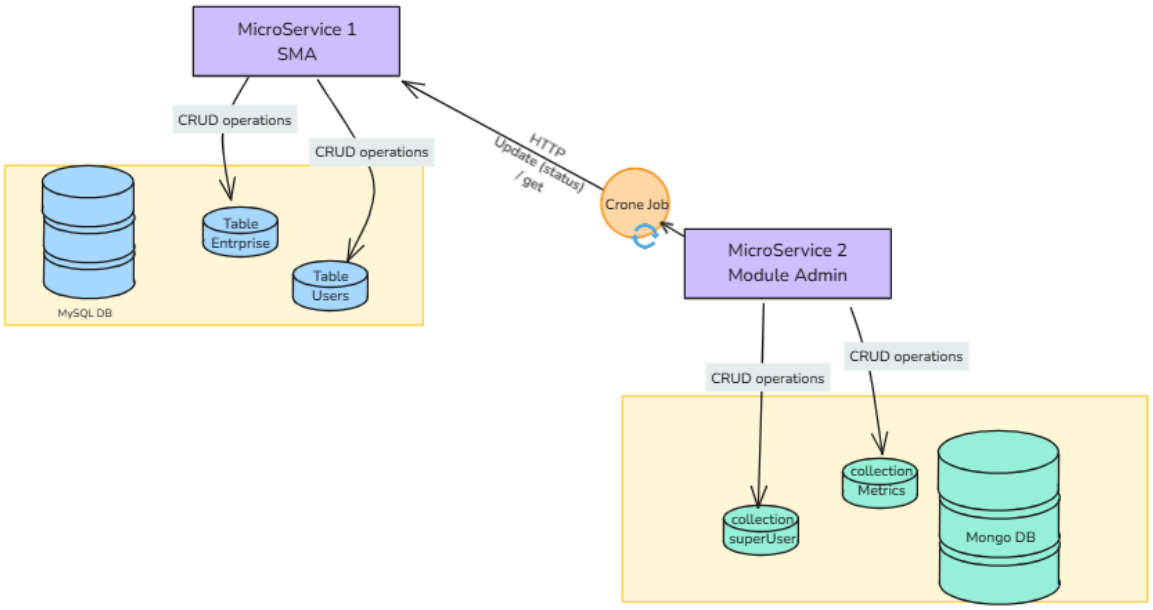
Introduction

Les deux microservices principaux :

1. **MicroService 1 (SMA) :** Une application dédiée à la gestion des chantiers de construction. Elle utilise sa propre base de données MySQL et assure la gestion complète des données.
2. **MicroService 2 (SMA-SUPER-USER) :** Une application connectée à MongoDB pour gérer des collections comme **SuperUser** et des métriques, et effectuant des appels HTTP vers SMA pour échanger des données spécifiques.

La gestion de la communication entre SMA et SMA-SUPER-USER présente des défis en raison de l'authentification JWT distincte pour chaque microservice. SMA-SUPER-USER doit récupérer les JWT de SMA pour chaque requête, ce qui ajoute de la complexité et peut entraîner des risques de sécurité.

Dans cette situation, la mise en place d'une API gateway en tant que point d'entrée unique pourrait être une solution. La passerelle centralise l'authentification et le routage, gère la validation des JWT et éliminerait le besoin d'échanges directs de JWT. Cela permettrait de réduire la complexité de la sécurité, de simplifier la communication et d'améliorer la scalabilité et la maintenabilité.



Architecture sans API Gateway (Requêtes HTTP Simples):

Fonctionnement

Dans une architecture sans API Gateway, chaque client communique directement avec les microservices. Chaque microservice expose ses endpoints à l'extérieur, et les clients doivent interagir avec eux individuellement.

Avantages

1. **Simplicité initiale** :
   - Aucun composant intermédiaire à configurer.
   - Les clients interagissent directement avec les services concernés.
2. **Performances accrues pour des architectures simples** :
   - Pas de latence ajoutée par un composant supplémentaire (API Gateway).
3. **Facilité de débogage** :
   - Les communications étant directes, les erreurs sont plus faciles à tracer dans les premiers stades du développement.

Inconvénients

1. **Couplage fort**:
    - Les services deviennent interdépendants, ce qui complique leur gestion et leur mise à jour.
2. **Manque de sécurité** :
    - Chaque service doit gérer l'authentification, augmentant le risque d'erreurs.
3. **Maintenance difficile** :
    - Toute modification d’un service peut nécessiter des ajustements dans tous les autres.
4. **Absence de centralisation** :
    - Pas de point unique pour surveiller, contrôler ou limiter les requêtes.
5. **Scalabilité limitée** :
    - À mesure que le système grandit, les connexions directes deviennent rapidement complexes et peu performantes.

Architecture avec API Gateway

Fonctionnement

L’API Gateway agit comme un point d’entrée unique pour toutes les requêtes des clients. Elle reçoit les requêtes, les transmet aux microservices appropriés, orchestre les appels si nécessaire, et renvoie une réponse consolidée au client.

Avantages

1. **Centralisation**
    - **Authentification unique** : L'API Gateway valide tous les JWTs et simplifie la gestion des droits.
    - **Routage** : Les requêtes sont automatiquement dirigées vers le service destiner .
    - **Gestion des erreurs** : Les messages d'erreur sont standardisés et centralisés.
2. **Découplage des services** :
    - Les services communiquent via l'API Gateway, réduisant leur interdépendance.
    - Les modifications d’un service n'affectent pas directement les autres.
3. **Sécurité renforcée** :
    - L'API Gateway gère les politiques de sécurité
    - Les services internes ne sont pas exposés directement au public.
4. **Observabilité et suivi** :
    - Des outils de monitoring et de logging intégrés permettent de suivre toutes les requêtes.
    - Les métriques et les journaux offrent une vue globale des performances.
5. **Scalabilité et performance** :
    - Une API Gateway peut équilibrer la charge entre plusieurs instances d’un service.
    - Elle permet l'ajout ou la suppression de services sans perturber le système global.
6. **Flexibilité** :

    - Possibilité d’ajouter des fonctionnalités comme la mise en cache, la compression des réponses ou encore des transformations de requêtes.

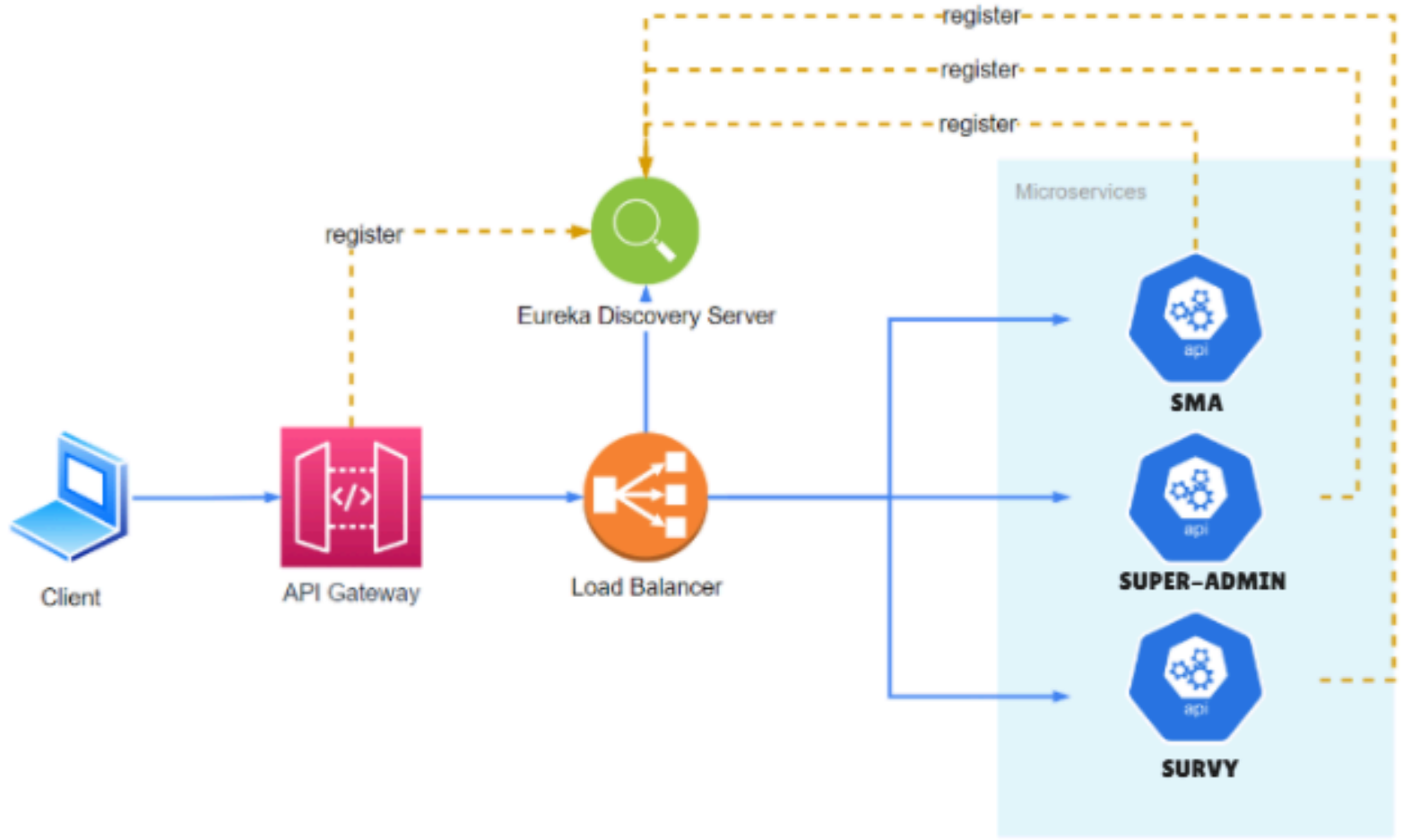
Inconvénients

1. **Complexité initiale** :
    - Mise en œuvre de l'API Gateway nécessite une configuration initiale et peut nécessiter une courbe d'apprentissage.
2. **Latence supplémentaire** :
    - Les requêtes passent par un composant intermédiaire, ce qui peut ajouter une légère latence.
3. **Point unique de défaillance** :
    - Si l'API Gateway rencontre un problème, cela peut bloquer toutes les requêtes, sauf si un mécanisme de haute disponibilité est mis en place.

Critères	Requêtes HTTP Simples	API Gateway
Complexité initiale	Faible	Moyenne
Gestion de la sécurité	Décentralisée, difficile	Centralisée, robuste
Évolutivité	Limitée	Excellente
Couplage	Élevé	Faible
Observabilité	Absente	Intégrée
Performances	Excellentes (petit système)	Excellentes (grand système)
Maintenance	Difficile	Simplifiée

Vue détaillée de l'implémentation technique

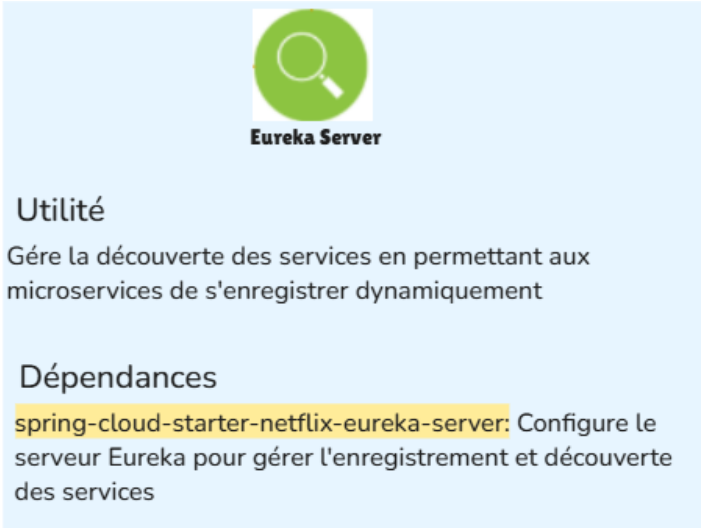
Architecture avec API gateway



Configuration des différents composant pour un API Gateway

---

1. Eureka Server




```xml
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
        </dependency>
```

EurekaServerApplication.java | application.properties | pom.xml (eureka-server)

```
spring.application.name=eureka-server

server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

EurekaServerApplication.java | application.properties | pom.xml (eureka-server)

```java
package com.devwise.eureka_server;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApplication {

    public static void main(String[] args) { SpringApplication.run(EurekaServerApplication.class, args); }

}
```

2. Microservices


```xml
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
```

```
spring.application.name=SUPERUSER
# MongoDB configuration
# Eureka Server
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
```

pom.xml (SMAsuperUser) | application.properties | SuperUserSmaApplication.java

```java
package com.example.SMAsuperUser;

import ...

@EnableDiscoveryClient
@SpringBootApplication
public class SuperUserSmaApplication {

    public static void main(String[] args) { SpringApplication.run(SuperUserSmaApplication.class, args); }


}
```

3. API Gateway

API Gateway

Utilité

Point d'entrée unique pour les clients externes
Gére le routage, la sécurité (authentification, autorisation) et éventuellement la transformation des requêtes/réponses

Dépendances

-spring-cloud-starter-gateway : Implémente l'API Gateway pour gérer le routage et les fonctionnalités associées

-spring-cloud-starter-netflix-eureka-client: Pemet a l'API Gateway de récupérer dynamiquemnt la liste des microservices via le serveur Eureka

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

```java
@EnableDiscoveryClient
@SpringBootApplication
public class ApiGatewayApplication {

    public static void main(String[] args) {
        SpringApplication.run(ApiGatewayApplication.class, args);
    }
    @Bean
    public HttpClient httpClient() {
        return HttpClient.newBuilder()
                .version(HttpClient.Version.HTTP_2) // Optional: Set HTTP version (e.g., HTTP/2)
                .build();
    }
}
```