Formation

Java

Spring - Microservices

m2iformation.fr

Sommaire

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction

Qu'est-ce qu'un microservice ?

Dans le monde du développement actuel, il est commun de remplacer la création d'une application **monolithique** unique par une pléthore de micro-services. De la sorte, il est plus aisé de mettre en oeuvre les principes **SOLID** ainsi que de rentre notre application **maintenable** et **scalable**. Le travail est également plus facile à répartir entre une équipe, chaque groupe pouvant ainsi travailler sur un microservice (une portion de l'application concernant un élément / objectif particulier) sans perturber le travail des autres.

Projet à multiples modules

Afin de simplifier par la suite l'ajout des dépendances, il est commun dans l'univers de Mavin d'avoir recourt à un gestionnaire de dépendances au niveau de notre projet. De la sorte, les différents sous-projets seront les enfants de notre projet et hériterons de ses dépendances (Lombok et Spring Testing) et de l'accès à toutes un série de dépendances (ici les différentes dépendances Spring):

```
<modules>
    <module>Service_01</module>
    <module>Service_02</module>
</modules>
```

Dépendances

```
<dependencyManagement>
      <dependencies>
          <dependency>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-dependencies</artifactId>
              <version>${spring.boot.maven.dependencies.version}</version>
              <scope>import</scope>
              <type>pom</type>
          </dependency>
      </dependencies>
  </dependencyManagement>

  <dependencies>
      <dependency>
          <groupId>org.projectlombok</groupId>
          <artifactId>lombok</artifactId>
      </dependency>
      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-test</artifactId>
      </dependency>
  </dependencies>
```

Plugins

```
<build>
      <pluginManagement>
          <plugins>
              <plugin>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-maven-plugin</artifactId>
                  <version>${spring.boot.maven.plugin.version}</version>
              </plugin>
          </plugins>
      </pluginManagement>
  </build>
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Premier microservice

Les API REST

Un microservice n'est au final ni plus ni moins qu'une application de type API. Cette application va être consommée pour permettre l'accès et le traitement à une portion des données de notre projet, ce qui peut, si besoin, demander l'appel d'autres microservices pour alimenter les données en dépendances (par exemple un pannier de produits devra contenir les produits qui se trouvent être traités par un autre microservice).

Entities

Pour pouvoir avoir nos données présentes dans une base de données, il convient de créer des entites JPA. Pour ce faire, il nous suffit de créer des classes Java disposant des annotations **@Entity** et de spécifier leur clé primaire via l'utilisation de **@Id** ainsi que de **@GeneratedValue**:

```
@Entity
public class Client {
  @Id
  @GeneratedValue(strategy = GenerationType.UUID)
  private UUID id;
}
```

Repositories

Pour ensuite nous offrir un accès et la possibilité de faire un CRUD vis à vis de nos entitées, nous allons utiliser le Repository Pattern, implémenté dans notre cas via l'interface **JpaRepository**:

```
public interface ClientRepository extends JpaRepository<Client, UUID> { }
```

Il est ensuite possible, au moyen des **request methods**, de créer des fonction supplémentaire pour par exemple obtenir le listing des clients par nom de famille:

```
public interface ClientRepository extends JpaRepository<Client, UUID> {
  List<Client> findAllByLastname(String lastname);
}
```

Les DTOs

Afin d'éviter de perturber la base de données en permanence, il est fréquent d'avoir recourt à des objets de type **DTOs**. Ces DTOs sont généralement une copie des objets de type **Entité** mais avec des modification de typage ou l'absence / renommage de propriétés pour correspondre à la demande des clients ou les spécificités du format JSON.

```
@Data
@Builder
public class Client {
  private UUID id;
}
```

Services

Pour pouvoir réaliser les opération, nous allons centraliser les manipulation de notre répository dans un composant Spring de type **@Service** :

```
@Service
@RequiredArgsConstructor
public class ClientService {
  private final ClientRepository clientRepository;

  public ClientDTO addClient(ClientDTO clientDTO) {
    // DTO => POJO

    clientRepository.save()

    // POJO => DTO
  }
}
```

Mappers

Afin d'assurer la transformation de nos éléments depuis / vers le DTOs et depuis / vers les entités, il est fréquent d'avoir recourt au processus du **mapping**, qui est d'ailleurs aussi extériorisé de notre service afin de ne pas en réduire la lisibilité.

```
@Mapper
public interface ClientMapper {
  public ClientDTO clienttoDto(Client client);
  public Client dtoToClient(ClientDTO clientDTO);
}
```

Pour procéder au mapping, il est possible de le faire soit même dans un **@Component** prévu à cet effet ou d'avoir recourt à des librairies tierces telles que **MapStruct** qui agis via des annotation et de la génération de code tout comme **Lombok**.

Controllers

Enfin, la récupération et l'envoi des requêtes HTTP se dérouleront dans un composant de type **@RestController**. Ce contrôleur sera le lieu principal du traitement des requêtes, dont le routing sera globalement fixé via **@RequestMapping** et le logging géré via **@Slf4j**.

```
@RestController
@RequiredArgsConstructor
@Slf4j
@RequestMapping("api/v1/clients")
public class ClientController {
  private final ClientService clientService;

  public ResponseEntity<ClientDTO> addClient(@RequestBody ClientDTO newClient) {
    ClientDTO savedClient = clientService.addClient(newClient);

    return ResponseEntity.ok(savedClient);
  }
}
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Communication Synchrone

Protocole HTTP

La méthode la plus aisée d'assurer une communication entre nos composants de façon synchrone est certainement d'utiliser le protocole HTTP. Après tous, les micro-services sont des API REST prévues de base à la gestion de ce protocole.
Le protocole fonctionne d'ailleurs, peu importe le verbe, via l'envoi d'une requête et l'attente d'une réponse. De part ce constat, il suffit de créer des endpoints sur nos micro-services servant à d'autres micro-services pour la récupération des données propres à chaque section de notre application. Pour cela, Spring nous offre plusieurs façon de créer des appels REST.

RestTemplate (1/2)

La plus ancienne est l'utilisation de **RestTemplate** (qui est cependant actuellement dépréciée). Pour utiliser cette méthodologie, il va dans un premier temps falloir créer une configuration de notre Bean de sorte à pouvoir l'obtenir par injection de dépendance au niveau de nos services:

```
@Configuration
public class CustomerConfig {
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

RestTemplate (2/2)

Une fois configurée, RestTemplate devient ainsi accessible au sein de nos micro-services pour appeller d'autres micro-services et ainsi alimenter leur propre réponse des données issues de leurs "collègues":

```
AuthResponse authResponse = restTemplate.getForObject(
  "http://localhost:8082/api/v1/auth/{userId}",
  AuthResponse.class,
  user.getId()
);

if (authResponse != null && authResponse.isAutheorized()) {
  // ...
}
```

WebClient (1/2)

Le remplaçant plus moderne de RestTemplate est désormais **WebClient**. Issu du projet WebFlux, il nous offrirait, si l'on le souhaitait, la capacité de faire des appels API non bloquant. Pour nous en servir, il convient également d'alimenter notre configuration de l'ajout de ce Bean et d'injecter une dépendance dans notre service:

```
@Configuration
public class CustomerConfig {
    @Bean
    public WebClient webClient() {
        return WebClient.builder().build()();
    }
}
```

WebClient (2/2)

La méthodologie pour faire ensuite notre appel API n'est pas très différente de celle employée par RestTemplate, à ceci près qu'il faut ajouter le caractère bloquant sous peine de ne pas avoir le résultat escompté:

```
AuthResponse authResponse = webClient.get()
.uri("http://localhost:8082/api/v1/auth/" + user.getId())
.retrieve()
.bodyToMono(AuthResponse.class)
.block();

if (authResponse != null && authResponse.isAutheorized()) {
  // ...
}
```

Limitations actuelles

Dans notre manière de fonctionner actuelle, nous sommes dans l'incipacité de profiter de l'un des avantage majeur d'une architecture micro-service: la scalabilité.

En effet, en cas de surcharge d'un micro-service, il est fréquent que celui-ci soit cloné et qu'une multitudes d'instances se répartissent ensuite la charge de travail. Ceci causera cependant la création de plusieurs ensembles nom de dommaine / ports pour accéder à nos micro-services.

Il devient ainsi impossible de spécifier à nos services vers quel port ils devront envoyer leur requêtes...

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Service Discovery

A quoi ça sert ?

Il va ainsi falloir à notre application un moyen de garder une trace de tous les micro-services, de leur adresses et de leurs ports pour pouvoir permettre aux autres services de s'en servir.
Le mécanisme du Service Discovery est en réalité un micri-service dont le rôle est la synchronisation d'un carnet d'adresse en temps réel entre tous les servuces de notre application. A un interval régulier, les micro-services vont transmettre leurs nom d'hôte et leur port et ainsi alimenter un repertoire complet de notre application, qui se verra automatiquement transmis à tous les micro-services, également à interval plus ou moins régulier.

Spring Cloud

Pour accéder à ces nouvelles fonctionnalité, il va cependant nous falloir ajouter **Spring Cloud** à notre projet. Cette section de Spring est en effet celle se concentrant sur l'implémentation et la capacité de mettre en place les patterns commun d'une infrastructure Cloud.

```
 <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-dependencies</artifactId>
    <version>${spring-cloud.version}</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

Netflix Eureka

Pour mettre en place ce pattern, dans le monde du Java, il est commun de faire recourt à **Netflix Eureka**, qui n'est autre qu'une implémentation du pattern mentionné précédemment, mit en lien avec l'ecosystème Spring.
Il nous faut ainsi créer un nouveau micro-service ayant la dépendance adéquate:

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

Configuration d'Eureka

Une fois notre service créé, il va lui falloir être un serveur Eureka. Pour cela, il va falloit ajouter les bonnes configuration dans le fichier de proprietés ainsi que l'annotation **@EnableEurekaServer**:

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

Enregistrer nos clients (1/2)

Il nous faut désormais avoir dans notre serveur le listing des micro-services de notre application. Pour ce faire, nous devrons, pour chaque micro-service, ajouter le lien avec le serveur Eureka:

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

Enregistrer nos clients (2/2)

Chaqu'un de nos micro-service doit également être configuré dans son fichier de propriétés. Si l'on se sert de la version **application.yml**, voici ce que l'on peut avoir:

```
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka
    register-with-eureka: true
    fetch-registry: true
```

Modifier notre communication

Une fois Eureka mit en place, il ne nous reste plus qu'à remplacer tous les ensembles nom de domaine / port par le nom de nos services dans le serveur Eureka. Ce dernier va jouer le rôle d'un pseudo-DNS et va permettre la redirection automatique de la requête vers le service concerné.
Néanmoins, en plus de la modification de l'URL, il va falloir également informer notre RestTemplate / WebClient que derrière cet URL peut se trouver non pas un micro-service, mais plusieurs. C'est alors que l'on va ajouter une annotation pour mettre en place le mécanisme du **Load Balancing**.

Load Balancing

Ce mécanisme est en réalité la capacité d'intérroger l'un de nos X replicat de service en fonction de sa charge de travail et / ou de son activité récénte. Pour le mettre en place, il suffit d'ajouter une annotation, **@LoadBalanced**:

```
@Configuration
public class CustomerConfig {
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Limitations actuelles

Nous atteignons désormais encore une fois une des limite de notre fonctionnement actuel: Il nous est nécessaire de poséder la classe de sérialisation / dsésérialisation dans chacun de nos microservices. De plus, la configuration de notre requête se retrouvera également dans tous les micro-services. Ceci cause un gros problème de scalabilité, en plus de nous éviter la mise en pratique du DRY...

Pour remédier à cela, nous allons désormais utiliser **OpenFeign**.

Ajout de la dépendance vers OpenFeign

Pour avoir accès aux fonctionnalités d'OpenFeign, il nous faut avoir un nouveau module possédant les bonnes dépendances:

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

Ce module sera rendu disponible en tant que dépendance pour nos autres modules, de sorte à ce qu'il serve à centraliser les différentes classes de transformation des données ainsi que les liens vers les endpoints.

Création de l'interface

Une fois notre dépendance ajouté, il suffit de créer des interfaces. Ces interfaces auront, sous la forme de méthodes, la représentation des différents endpoints que l'on cherche à atteindre pour un micro-service:

```
@FeignClient(value = "auth-service", path = "api/v1/auth")
public interface AuthClient {
    @GetMapping("{userId}")
    AuthResponse isAllowed(@PathVariable UUID userId);
}
```

Ajout du client OpenFeign

Une fois l'ensemble des dépendance et l'interface créé, il ne nous reste plus qu'à ajouter une annotation dans notre projet de micro-service ayant besoin de réaliser l'appel, **@EnableFeignClients**:

```
@SpringBootApplication
@EnableFeignClients(basePackages = "org.example.clients")
public class CustomerApplication {
    public static void main(String[] args) {
        SpringApplication.run(CustomerApplication.class, args);
    }
}
```

Changement des appels

Dans les méthodes faisant présemment appel à notre WebClient ou à RestTemplate, il esuffit désormais d'ajouter une dépendance vers notre interface et de faire appel à la méthode adéquate:

```
@Service
@RequiredArgsConstructor
public class CustomerService {
    private final CustomerRepository customerRepository;
    private final AuthClient authClient;
    public void registerCustomer(CustomerDTO customerDTO) {
        // ...

         AuthResponse authResponse = fraudClient.isAllowed(customer.getId());
    }
}
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Distributed Tracing

L'objectif

Maintenant que l'on a réussit à faire communiquer plus efficacement nos micro-services, il devient également important d'améliorer notre mécanisme de suivi des erreurs. En effet, dans le cas où une erreur se produit dans un appel API, il est actuellement difficile de savoir de à quel endroit et à quel moment l'erreur a eu lieu.

Pour résoudre ce problème, on va implémenter le mécanisme du Distributed Tracing en faisant en sorte de récupérer en temps réel le passage de nos requêtes via des IDs uniques pour chaque requête (**Trace Id**) mais également pour chaque micro-service (**Span Id**).

Micrometer Tracing

Afin de mettre en place cette nouvelle implémentation, il est possible d'avoir recourt à **Micrometer Tracing**, qui, si mit en commun avec **Zipkin**, va permettre le suivi de l'avancement de nos requête au sein d'une interface graphique sur notre navigateur préféré !
Pour mettre en place ce système, il va simplement nous falloir ajouter des dépendances à notre projet ainsi que de modifier légèrement les fichiers de configuration de nos micro-services.

Dependances MAVEN

```
<dependency>
            <groupId>io.github.openfeign</groupId>
            <artifactId>feign-micrometer</artifactId>
        </dependency>
        <dependency>
            <groupId>io.micrometer</groupId>
            <artifactId>micrometer-tracing</artifactId>
        </dependency>
        <dependency>
            <groupId>io.micrometer</groupId>
            <artifactId>micrometer-tracing-bridge-brave</artifactId>
        </dependency>
        <dependency>
            <groupId>io.zipkin.reporter2</groupId>
            <artifactId>zipkin-reporter-brave</artifactId>
        </dependency>
```

Configuration

Il nous faut donc désormais avoir, au sein de notre fichier **application.yml**, la configuration suivante:

```yaml
management:
  tracing:
    sampling:
      probability: 1.0 # 0.1 si non spécifié

logging:
  pattern:
    level: '%5p [${spring.application.name:},%X{traceId:-},%X{spanId:-}]'
```

La probabilité est modifiée à des fins de débugging, il faudra la remettre à sa valeur de base (0.1) pour la production.

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Spring Cloud Gateway

Load Balancing

Dans le cas d'une application à grande echelle, il est fort à parier qu'un nombre important de clients vont chercher à se connecter à nos micro-services. Pour éviter des coups initiaux trop importants et avoir quelque chose de résistant dans le temps, nous allons avoir plusieurs machines faisant tourner une ou plusieurs instances de nos micro-services. L'accès à ces machines passera par un **Load Balancer** pour que personne n'ait pas à connaître par coeur les adresse IP ou les ports à atteindre. Le Load Balancing se chargera de diriger les requêtes vers les X instances de nos services en fonction de leur charge et activité, que ce soit en interne ou en externe.

Round Robin

Pour que notre Load Balancer soit capable de connaitre vers quels instance il doit rediriger les requêtes, il possède par défaut une redirection de type séquentielle (l'un après l'autre). L'algorithme utilisé pour ce mécanisme est nommé le **Round Robin**.
Bien entendu, d'autres algorithmes existent, tels que le **Least Connection** qui envera la requête vers le service le moins solicité, ou le **Hash** qui se basera sur une clé fournie.

Health Checks

Pour que les micro-services fassent partie du listing géré par l'algorithme de sélection, il faut cependant qu'ils communiquent leur état de santé. Pour ce faire, il est souvent nécessaire de mettre en place les **Health Checks**. Ces Health checks ne sont ni plus ni moins que des contrôles (souvent un simple endpoint appellé par protocole HTTP) réalisés à interval régulier.

Circuit Breaker

Le **Circuit Breaker** est un autre pattern utilisé dans le but d'éviter les timeouts ou les réponses absentes pour nos requêtes.
Le principe est d'avoir une gestion de trois états pour nos méthodes de récupération de données par une requête:

- **Fermé**
- **Ouvert**
- **Semi-Ouvert**

Circuit Breaker: Closed

Dans cet état, les requêtes passent et les réponses sont traitées normalement. Cet état est conservé du temps qu'un **pourcentage d'échec** à nos requêtes n'est pas atteint. Ce pourcentage est appellé le **Threshold**.

Du temps que le threshold n'est pas atteint, notre breaker reste en état fermé et notre application continue de fonctionner comme avant l'implémentation du pattern. Si les requêtes commencent à échouer, on va alors atteindre un score d'échec de plus en plus élevé, jusqu'à potentiellement dépasser le threshold.

Circuit Breaker: Open

En cas de dépassement du threshold, le circuit breaker passera en état **ouvert**. Les requêtes ne seront alors plus envoyées.

Cependant, en cas de micro-services dépendant de la réponse qu'aurait provoqué notre requête, il convient de ne pas bloquer le reste des requêtes. Dans ce cas de figure, on va idéalement générer des **fallbacks** (simulâcres de réponses) afin de leur permettre un fonctionnement similaire. Cet état dure un **temps déterminé**, le **timeout**.

Circuit Breaker: Half-Open

Cet état est un état de transition servant au test de nouvelles requêtes. On va ainsi retenter de faire les requêtes prévues un certains nombre de fois (un nombre de **retries**). En fonction de l'issue, on repasse en **Ouvert** ou en **Fermé**.

- Si les requêtes échouent après un certain nombre de renlance, alors on va repasser en état **Ouvert**.
- Si les requêtes parviennent à leur destination, on va repasser en état **Fermé**.

Spring Cloud Gateway

Si l'on a envie de mettre en place une partie des mécaniques mentionnées précédemment dans un projet de type Spring, il est possible d'avoir recourt à **Spring Cloud Gateway**. Via l'ajout d'une Gateway, il devient possible de rediriger les requêtes vers l'une des instances de notre cluster de façon transparente pour l'utilisation. Sa capacité à gérer aussi le **Service Discovery** et le **Circuit Breaker** la rende intéressante pour un projet de moyenne échelle, mais il reste tout de même préférable d'avoir recourt à un Load Balancer issu d'un fournisseur Cloud en cas d'application à très grande echelle, ces derniers pouvant s'étendre en cas de besoin (forte demande client).

Mise en place

Pour créer une Gateway, il nous suffit de recréer un nouveau projet possédant les bonnes dépendances puis de configurer la chose dans son fichier **application.yml**.

Pour les dépendances Maven, la dépendance importante à ajouter à notre module est la suivante:

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

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Docker

Présentation

Docker est un outil actuellement indispensable au monde du développement et qui sert à créer un lien entre les opérateurs systèmes et l'équipe de développement. Son arrivée à rendu populaire le **DevOps** et son objectif principal est la création d'images les plus petites possible mais contenant à la fois notre application et toutes ses dépendances.
De part l'utilisation d'une image Docker, il n'est plus nécessaire d'installer Java sur un ordinateur, ni toutes les dépendances Maven. L'image est déjà lançable dés la fin du téléchargement, peu importe l'OS sur lequel on travaille.

Terminologie

Dans l'écosystème Docker, il existe certains termes qu'il convient de connaître et de comprendre afin d'éviter les confusions:

- **Image**: Une sauvegarde des fichiers d'applicatif ainsi que de toutes ses dépendances.
- **Conteneur**: Un élément Docker servant au lancement d'une image
- **Volume**: Un dossier dans le système Docker permettant la sauvegarde des fichiers de notre conteneur
- **Réseau**: Une connexion rendue possible entre les conteneurs et / ou avec l'ordinateur hôte.

Installer Docker

Dans le cas de l'utilisation de Docker sur un environnement Windows, l'installation passe en premier par la mise à jour de la **WSL** (Windows SubSystem for Linux). Pour ce faire, il faut d'abord ouvrir un terminal:

```
wsl --install
```

Une fois fait, on peut aller sur le site de Docker Desktop pour télécharger puis installer l'outil. Dans les paramètres lors de l'installation, nous choisirons d'utiliser la **WSL 2** en lieu et place d'**Hyper-V**.

Les commandes de base

- **docker ps**: Permet de voir les conteneurs en court d'utilisation
- **docker images**: Permet de voir les images disponibles
- **docker pull**: Permet de récupérer une image depuis un registery
- **docker run**: Permet de démarrer un conteneur
- **docker exec**: Permet d'exécuter une commande dans un conteneur
- **docker build**: Permet de créer une image à partir d'un Dockerfile
- **docker save**: Permet de sauvegarder l'état d'une image
- **docker prune**: Permet de libérer l'espace de stockage

Dockerfile

Un **Dockerfile** est un type de fichier servant à la création d'image. Il détaille, au moyen de mot-clés et de lignes d'instructions, les étapes de création d'une image et les manipulation à effectuer avant la sauvegarde de l'état final dans une image déposable sur **Docker Hub** ou tout autre registery tel qu'**Azure Container Registery**.

Google Cloud Tools: Jib

Dans le cadre de Maven, il est possible d'avoir recourt à **Jib** pour la création et l'envoi d'images Docker. Ces images se voient crées via notre projet et seront automatiquement envoyée sur le service désiré à condition d'avoir les bons crédentials. Pour utiliser Jib, il nous faut ajouter la dépendance Maven requise.
Dans le cadre de l'utilisation de Docker Desktop, si l'on veut pouvoir utiliser Jib, il va nous falloir changer notre méthode de gestion des identitfiants Docker. Pour cela, il nous faut modifier le fichier **config.json** et y mettre une autre valeur pour **credStore**:

```
"credStore": "wincred"
```

Dépendance Maven Jib

```
<plugin>
      <groupId>com.google.cloud.tools</groupId>
      <artifactId>jib-maven-plugin</artifactId>
      <version>3.4.0</version>
      <configuration>
        <to>
          <image>docker.io/my-docker-id/my-app</image>
        </to>
      </configuration>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>build</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Message Broker

Communication asynchrone

Dans le cas d'une communication entre micro-services, il est possible que certains micro-services se voient être ralentis. Dans ce genre de moment, l'ensemble de l'application, en cas de communication synchrone pour tout, va se voir ralentie.
Il convient donc de se demander s'il est obligatoire d'attendre l'appel API vers telle ou telle partie de l'application pour générer le retour final à la requête utilisateur, ou s'il ne serait pas plus judicieux d'avoir recourt à un envoie de requête non bloquant pouvant provoquer des effets annexes via la communication **asynchrone** entre nos micro-services.

AMQP

Pour résoudre un problème de bottleneck dans notre réseau d'API, il est possible d'avoir recourt à un mécanisme de **Message Queuing** qui va se charger de stocker l'ensemble des messages dans le but de pouvoir les faire parvenir selon la règle du FIFO (First In First Out).

Pour ce faire, le protocole que l'on va utiliser se nomme le **AMQP** (Advanced Message Queuing Protocol), qu'il est possible d'implémenter via RabbitMQ ou Kafka. Ces deux services sont des **Message Brokers**, stockant nos messages pour les envoyer, soit immédiatement, soit en cas d'impossibilité technique, par la suite.

Terminologie de RabbitMQ

Dans l'univers RabbitMQ, il existe plusieurs termes:

- **Producer**: L'élément générant les messages
- **Consumer**: L'élément allant récupérer les messages
- **Exchange**: Un point d'entrée des messages
- **Queue**: Une entité chargée de récupérer, stocker et distribuer les messages
- **Binding**: Le processus de liaison des queues et des exchanges.

Ajout d'un conteneur RadditMQ

Pour obtenir rapidement RabbitMQ, il va nous falloir l'installer. Pour ce faire, il est possible de suivre ce lien. De notre côté, nous allons simplement ajouter un conteneur via l'utilisation de Docker compose:

```
services:
  rabbitmq:
    image: rabbitmq:management
    ports:
      - "5672:5672"
      - "15672:15672"
```

Dépendance MAVEN

Une fois le serveur RabbitMQ rendu disponible, il nous faudra ajouter la dépandance adéquate à nos projets devant se servir de RabbitMQ. Celle ci se trouve être:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Configuration

Pour utiliser RabbitMQ, il va falloir idéalement créer un module Maven qui va servir de dépendance commune à tous nos projets devant communiquer via le message broker. Cette dépendance contiendra dans un premier temps un fichier de configuration:

```
@Configuration
public class RabbitMQConfig {
    @Bean
    public MessageConverter jacksonConverter() {
        return new Jackson2JsonMessageConverter();
    }
}
```

Création d'un Producer

Vient ensuite la création dans cette dépendance de notre méthode permettant l'accès au Producer:

```
@Component
@Slf4j
@RequiredArgsConstructor
public class MessageProducer {
    private final RabbitTemplate rabbitTemplate;

    public void publish(Object payload, String exchange, String routingKey) {
        log.info("Publishing to {} using routingKey {}. Payload: {}",
          exchange, routingKey, payload);
        rabbitTemplate
          .convertAndSend(exchange, routingKey, payload);
        log.info("Published to {} using routingKey {}. Payload: {}",
          exchange, routingKey, payload);
    }
}
```

Accès aux Beans

Pour permettre l'accès à ce producer dans les micro-services devant envoyer un message, l'ajout de la propriété adéquate dans le fichier de configuration de Spring est également nécessaire. Celle-ci se trouve être **spring.rabbitmq.addresses** et se nomme **spring.rabbitmq.addresses**. Elle doit contenir le lien vers le serveur RabbitMQ. De plus, il nous faut, en plus de la dépendance vers le module Maven, informer Spring que les Beans peuvent provenir de ces packages:

```
@SpringBootApplication(scanBasePackages =
  {"org.example.userservice", "org.example.rabbitmq"}
)
```

Envoi du message

Pour ensuite envoyer le message, rien de plus simple. Ilm suffit d'ajouter à notre service concerné la dépendance et de faire appel à la méthode du producer. Il ne faudra pas oublier de bien spécifier l'échange et la clé de routage pour diriger le message vers la bonne queue:

```
private final MessageProducer messageProducer;

    public void register(RegisterRequest registerRequest) {
        // ...

        messageProducer.publish(registerRequest,
                "internal.exchange", "internal.messages.routing-key");
    }
```

Configuration de la réception

Pour la partie réception, il va nous falloir créer plusieurs choses. Dans un premier temps, nous allons obtenir les informations nécessaire au Binding de la queue avec l'échange via les propriétés issues d'**application.yml**:

```
@Value("${rabbitmq.exchanges.internal}")
private String internalExchange;

@Value("${rabbitmq.queues.messages}")
private String messagesQueue;

@Value("${rabbitmq.routing-keys.internal-messages}")
private String internalMessagesRoutingKey;
```

Binding Exchange-Queue

Il est ensuite possible d'avoir recourt à la définition de Beans pour procéder à la mise en place des liaisons entre les queues et les échanges:

```
@Bean
public Binding internalToNotificationBinding() {
    return BindingBuilder
            .bind(new Queue(this.messagesQueue))
            .to(new TopicExchange(this.internalExchange))
            .with(this.internalMessagesRoutingKey);
}
```

Receptionner les messages

Enfin, pour ajouter le déclenchement de méthodes à la réception des messages et ainsi en extraire les informations, il nous suffit d'ajouter une annotation au niveau de nos méthodes Java:

```
@RabbitListener(queues = "${rabbitmq.queues.messages}")
    public void consumer(RegisterRequest registerRequest) {
        log.info("Consumed {} from queue", registerRequest);
    }
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Kubernetes

Présentation

Kubernetes est un outil permettant l'orchestration des conteneurs. Via son utilisation, il est possible de déployer des conteneurs de façon distribuées, de les mettre en lien les uns avec les autres, que cela soit au sein d'un même système ou dans un environnement Cloud.
De part l'utilisation de Kubernetes, il est aussi possible d'avoir recourt au mécanisme du **Load Balancing**, de **communication internet / externe** de nos éléments et ce que cela soit sur un ordinateur ou sur plusieurs ordinateurs réparti dans des zones géographiques différentes.

Terminologie

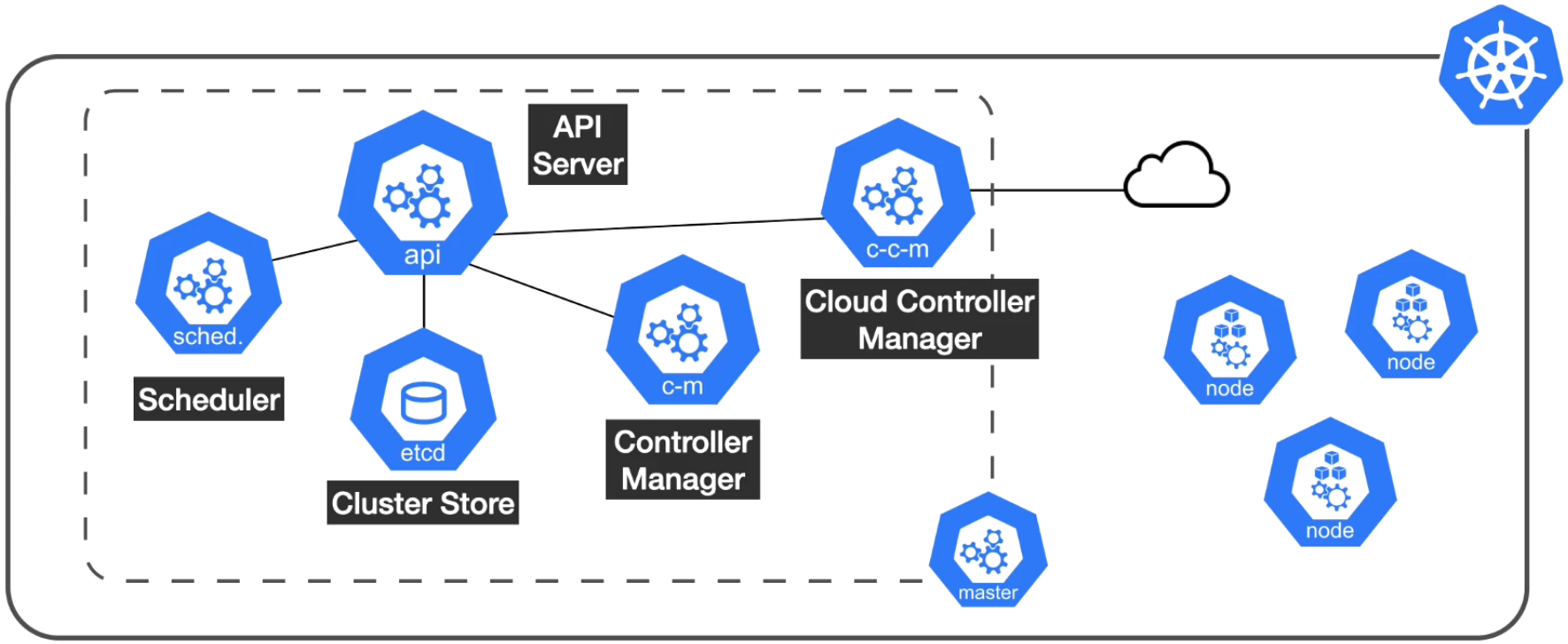
- **Node**: Une machine, virtuelle ou non
  - **Worker Node**: Une node servant à l'exécution de pods
  - **Master Node**: Une node dédiée à la gestion du cluster
- **Cluster**: Un ensemble de nodes
- **Pod**: Une abstraction de conteneur

Master Node

- **API Server**: Ce serveur va ensuite relayer l'informations aux services concernés pour provoquer les changements demandés.
- **Cluster Store**: Il se charge de stocker l'état du cluster via une association clés-valeurs.
- **Scheduler**: Observe la charge de travail des nodes et se charge de l'assignation du pod pour les services K8s.
- **Controller Manager**: Daemon gérant les contrôleurs. Il permet d'atteindre, par modification du cluster, l'état demandé dans le serveur API via une boucle d'observation des contrôleurs.

Control Plane

- Made of several components



Worker Node

La **Worker Node** est en général une machine (virtuelle ou non) utilisant Linux et qui va posséder l'environnement nécessaire au lancement de notre application.

- **kubelet**: Agent principal qui va intéragir avec le Container Runtime et le serveur API
- **Container runtime**: Va récupérer les images des registeries et gérer les conteneurs via la Container Runtime Interface (**CRI**)
- **Kube Proxy**: Agent lancé sur chaque node via **DaemonSets** pour gérer le networking, en particulier si un load balancing a lieu.

Découvrir Kubernetes

Dans le cas où l'on voudrait réaliser une mise en production d'application, il faudrait bien entendu passer par l'utilisation d'un cluster géré par un cloud provider du type EKS, AKS, GKE. Malheureusement, cette solution, dans un but d'apprentissage, est bien trop couteuse. Nous allons alors avoir plusieurs façon d'obtenir un cluster local sur notre machine:

- Minikube
- Kind
- Docker

Minikube

Dans notre cas, nous allons utiliser **Minikube**, qui peut s'installer facilement via ce lien.
Sur Windows, en cas d'utilisation de chocolatey, il suffit d'entrer la commande requise et le tour est joué:

```
choco install minikube
```

Une fois installé, il est possible de lancer notre cluster personnel via la commande:

```
minikube start
```

Deployment

Quand bien même il est possible de créer manuellement nos pods dans notre cluster K8s, il est conseillé d'éviter. A la place, nous créons ce que l'on appelle un **deployment** qui va définir la structure des pods. De part cette surcouche, il est possible à K8s de savoir que les pods doivent en permanence être présent et dans un état acceptable au sein d'un **replicaset**. Un replicaset, de son côté, est un ensemble de **pods**.

Service

Dans le but d'accéder ensuite à notre application, il va falloir utiliser une autre ressource K8s dédiée à la communication interne / externe de l'application. Ce type de ressource se nomme un **service** et il en existe de plusieurs type en fonction du besoin:

- **ClusterIP** Pour la communication au sein du cluster uniquement
- **NodePort** Permet l'ajout de port joignable par l'exterieur par dessus un ClusterIP
- **LoadBalancer**: Permet l'ajout d'une adresse IP fixe et d'un mécanisme de liaison vers les pods d'un replicaset.

Service Discovery

Dans l'écosystème K8s, il existe un mécanisme de **Service Registery** permettant la récupération des adresses IP des différents services afin d'informer les autres services au sein d'un réseau de noms qu'ils peuvent intéroger en lieu et place d'une adresse IP. Pour fonctionner, K8s va utiliser **CoreDNS** en tant qu'outil de Service Discovery.

De son côté, **Kube Proxy** va avoir pour objectif la gestion des règles de communication interieur / extérieur en observant le serveur API et en créant des règles **IPVS** pour intercepter le traffic à destination du ClusterIP et rediriger via l'utilisation des labels de pods.

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Kafka

Un autre message broker

Kafka joue à peu près le même rôle au sein d'une application que RabbitMQ.
Cependant, contrairement à lui, il s'agit d'un système distribué qui peut agir depuis un cluster plus ou moins large de nodes. Il est aussi possible, au moyen de Kafka, de créer des application de type **Event Driven** (comme par exemple une application dont le rôle est le suivit du livreur de pizza en temps réel).
De plus, contrairement à RabbitMQ, les messages ne sont pas automatiquement supprimé une fois distribués. Il est possible d'injecter des données, d'en extraire ou de les modifier en temps réel.

Ajout de Kafka

Pour pouvoir de notre côté utiliser Kafka, il va dans un premier temps créer un serveur. Tout comme RabbitMQ, il est faut un endroit centralisant les données pour que nos producers et nos consumers puissent communiquer entre eux.
Pout installer Kafka, il suffit de suivre ce guide.

Sur **Windows**, attention à bien utiliser les fichier portant l'extension **.bat** et se trouvant dans le sous-dossier windows de l'archive téléchargeable à cette adresse.

Dépendance Maven

Pour pouvoir utiliser Kafka, une dépendance Maven sera bien entendu nécessaire. Celle-ci se présente de la sorte:

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

Configuration de l'application

application.yml:

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
```

KafkaTopicConfig.java:

```
@Configuration
public class KafkaTopicConfig {
  @Bean
  public NewTopic myTopic() {
    return TopicBuilder.name("topicName").build();
  }
}
```

Configuration du Consumer

La configuration d'une application Kafka passe en majorité par la configuration des Topics:

```
@Configuration
public class ConsumerConfig {
    @Bean
    public NewTopic topic() {
        return TopicBuilder.name("topic1")
                .partitions(10)
                .replicas(1)
                .build();
    }
}
```

Réception des messages

Les méthodes possédant l'annotation **@KafkaListener** se voient être déclenchées en cas de publication d'un message du topic visé:

```
@KafkaListener(id = "myId", topics = "topic1")
public void listen(String in) {
    System.out.println(in);
}
```

Configuration du Producer

Si l'on souhaite envoyer des POJO en tant que message, il va nous falloir modifier le template Kafka de sorte à user des bons outils de sérialisation:

```
@Bean
public ProducerFactory<String, Object> producerFactory() {
    return new DefaultKafkaProducerFactory<>(producerConfigs());
}

@Bean
public Map<String, Object> producerConfigs() {
    Map<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);
    return props;
}

@Bean
public KafkaTemplate<String, Object> kafkaTemplate() {
    return new KafkaTemplate<String, Object>(producerFactory());
}
```

Envoi de messages

L'envoi de message se fait via **KafkaTemplate**, de la sorte:

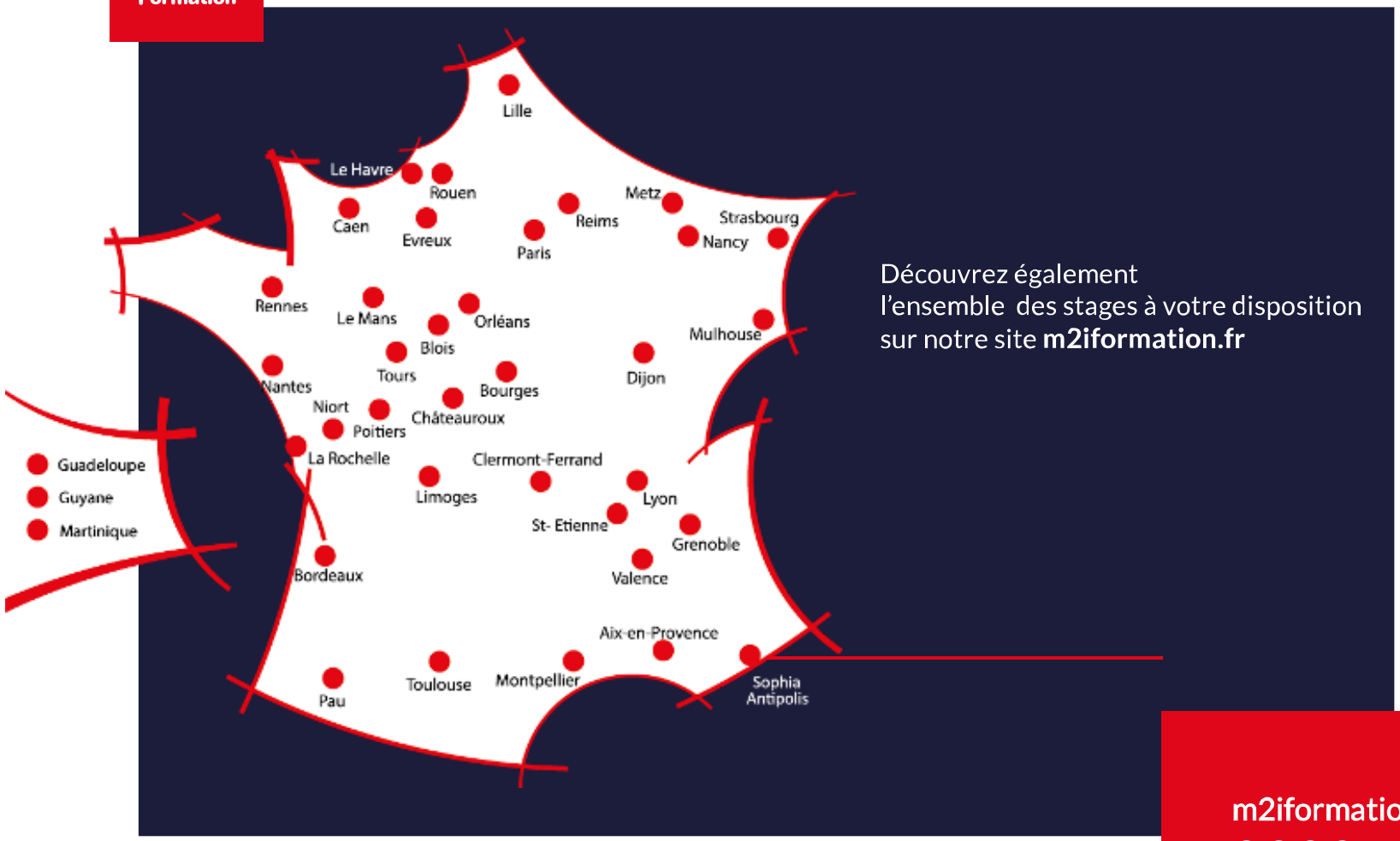
```
private final KafkaTemplate<String, String> template;

public void sendMessage(String message) {
  template.send("topic1", message);
}
```

- Ou si l'on veut envoyer un objet:

```
private final KafkaTemplate<String, Object> template;

public void sendUser(User user) {
  template.send("topic2", user);
}
```



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr