Spring Boot : Microservice

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

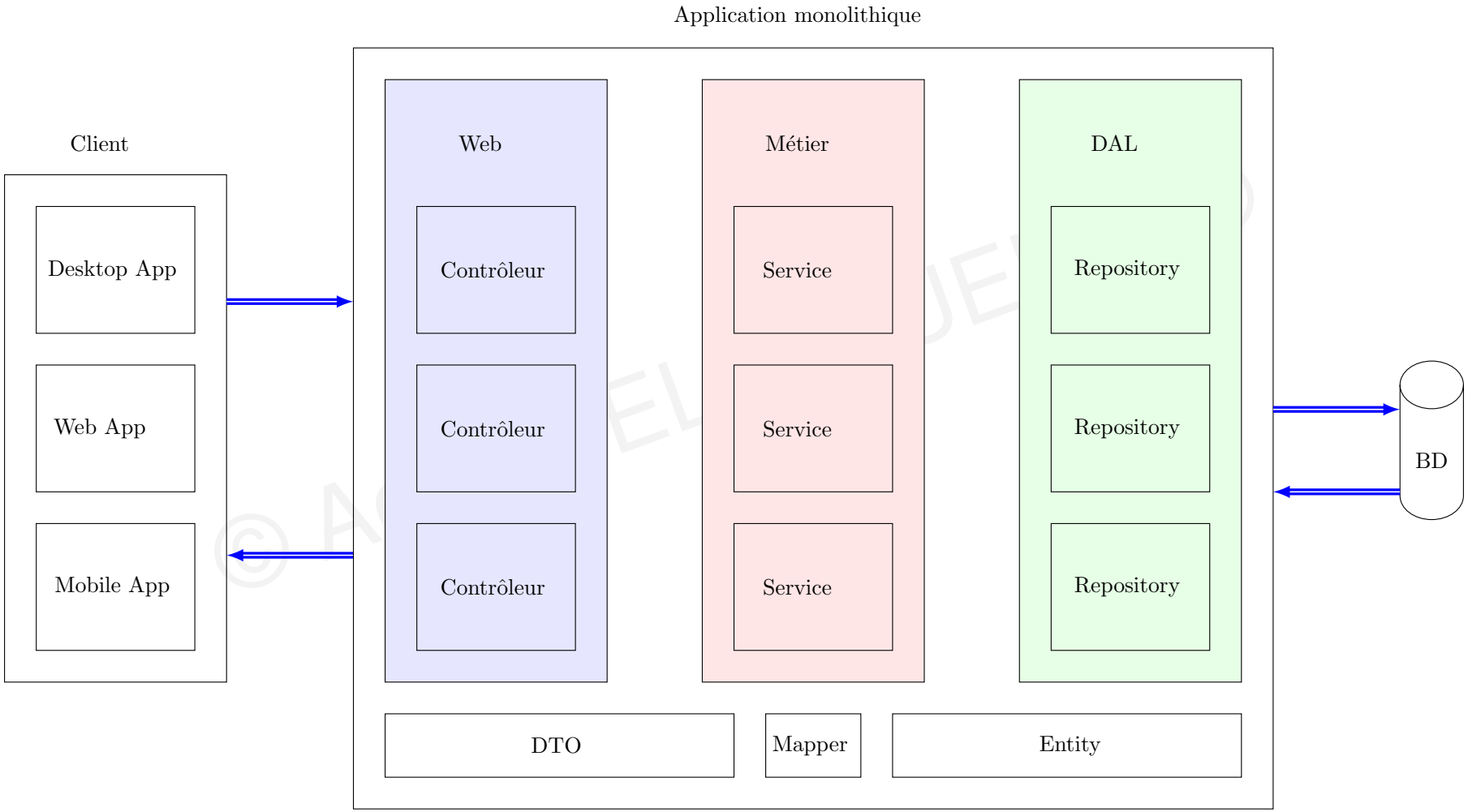
Plan

Spring Boot

Applications Web classiques : Architecture Monolithique

- Volumineuses
- Trop d'interdépendance entre les fichiers
- S'exécutant sur un serveur en respectant l'architecture (requête-réponse)
- Format de réponse : **HTML**, **XML** ou **JSON**

Spring Boot


Application monolithique
Client
Desktop App
Web App
Mobile App
Web
Contrôleur
Contrôleur
Contrôleur
Métier
Service
Service
Service
DAL
Repository
Repository
Repository
BD
DTO
Mapper
Entity

Spring Boot

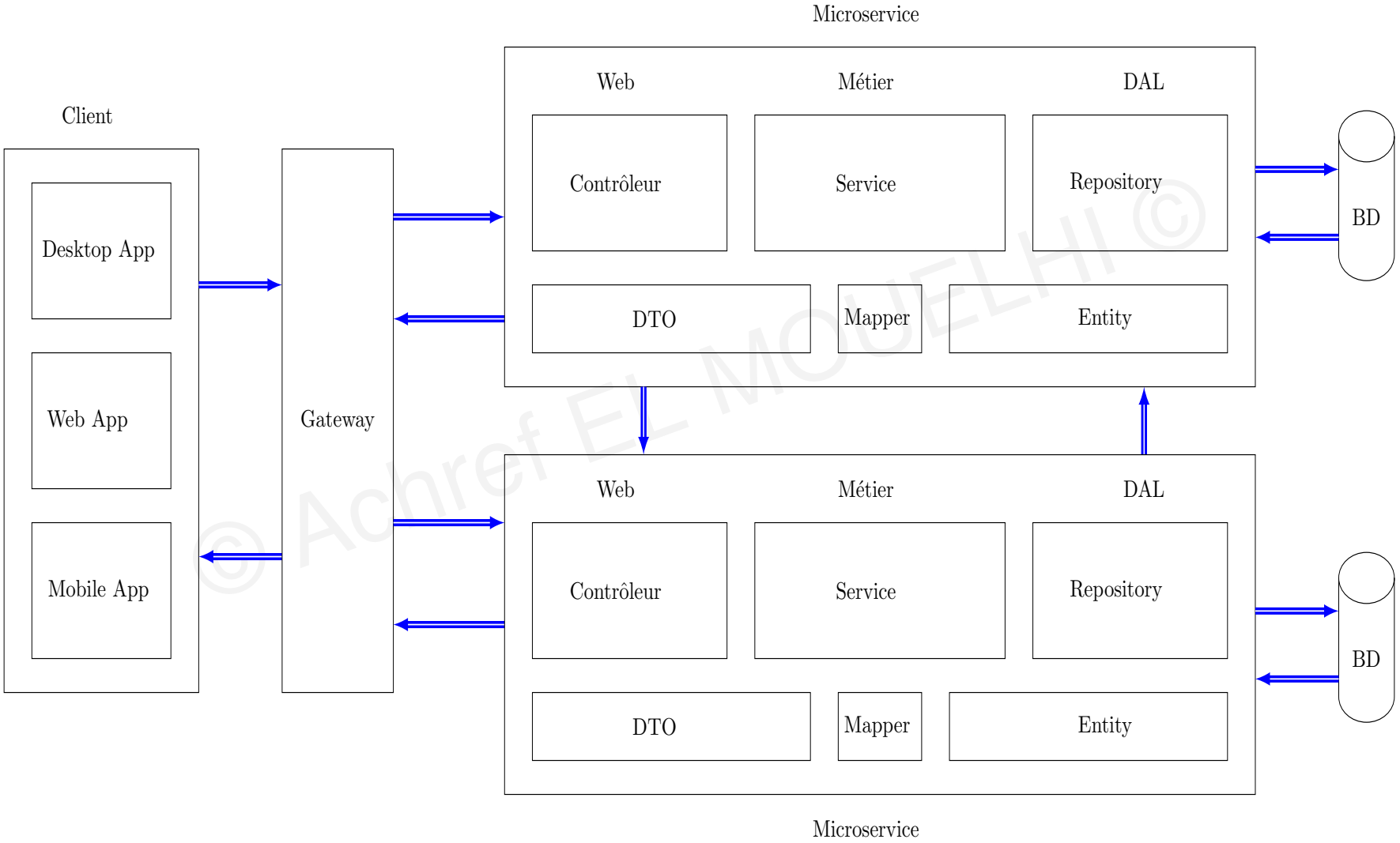
Problématiques

- Évolution
  - Taille $\nearrow \Rightarrow$ complexité $\nearrow$
  - Après mise-à-jour : faut-il garder le code précédent ou le supprimer ?
- Déploiement
  - Échec de déploiement $\Rightarrow$ toutes les briques de code sont remises en question
  - Une bonne maîtrise de tout le code est nécessaire pour la correction
- Scalabilité : extensibilité sur plus de ressources
  - verticale $\Rightarrow$ remplacer un serveur acceptant x nombre d'utilisateur par un autre acceptant y nombre d'utilisateur (avec $x < y$) : $\surd$
  - horizontale $\Rightarrow$ placer l'application sur plusieurs serveurs en parallèle : X

Architecture MicroServices

- Diviser une application en plusieurs fragments intégralement autonomes
- Chaque fragment a un objectif bien particulier
- Un fragment est appelé **Web Service**
- Un **Web Service** peut être utilisé par
  - notre application monolithique existante,
  - la partie serveur
  - la partie client
  - d'autres services Web.
- Une Architecture MicroService (**MSA**) est implémentée par un ensemble de **Web Service** exposant une **API REST**
- Le terme **Microservice** peut désigner l'architecture ou un **Web Service**
- L'architecture **MSA** permet de réaliser des applications **Cloud-native** : ce qui permet d'augmenter et de diminuer le nombre de ressources à la demande en fonction du besoin.

Spring Boot



Spring Boot

MSA vs **SOA** (**S**ervice-**O**riented **A**rchitecture)

- Dans une **MSA**, un service s'occupe d'une fonctionnalité. Dans une **SOA**, un service s'occupe d'un domaine.
- La taille des services dans une **MSA** $<$ La taille des services dans une **SOA**.
- Chaque Microservice possède sa propre base de données. Dans une **SOA**, une base de données est très souvent utilisée par plusieurs services.
- Les **SOA** utilisent souvent **SOAP**. Les **MSA** privilégient **REST**.

Spring Boot

MSA : 3 options pour les bases de données

- `Private-tables-per-service` : chaque service possède un ensemble de tables accessibles uniquement via ce service.
- `Schema-per-service` : chaque service possède un schéma de base de données privé pour ce service.
- `Database-server-per-service` : chaque service possède son propre serveur de base de données.

Spring Boot

MSA : 3 options pour les bases de données

- `Private-tables-per-service` : chaque service possède un ensemble de tables accessibles uniquement via ce service.
- `Schema-per-service` : chaque service possède un schéma de base de données privé pour ce service.
- `Database-server-per-service` : chaque service possède son propre serveur de base de données.

Avantages d'une base de données par Microservice

- Garantir l'autonomie des services et le couplage faible.
- Chaque service peut utiliser le type de base de données le mieux adapté à ses besoins.

Spring Boot

Inconvénients d'une base de données par Microservice

- Difficulté de mise en place de transactions couvrant plusieurs services.
- Complexité de requête de jointure sur des données situées dans plusieurs base de données.
- Compétences et connaissances de plusieurs base de données **SQL** et **NoSQL**

MSA vs Architecture Monolithique

- Utiliser **MSA** si :
  - Nombreuses personnes travaillent sur le système à long terme (couplage faible).
  - Possibilité d'avoir plusieurs millions d'utilisateurs simultanés (scalabilité horizontale)
  - Besoin d'un système hautement disponible avec redondance
- Utiliser une Architecture Monolithique si :
  - Système développé par une personne ou une petite équipe
  - Système pas trop évolutif : utilisé que par quelques centaines voire des milliers de personnes (scalabilité verticale)
  - Besoin d'une architecture simple, facile à comprendre, à maintenir, à déployer et à surveiller.

Spring Boot

Création de projet Spring Boot

- Aller dans `File > New > Other`
- Chercher `Spring`, dans `Spring Boot` sélectionner `Spring Starter Project` et cliquer sur `Next >`
- Saisir
  - `micro-service-personne` dans `Name`,
  - `com.example` dans `Group`,
  - `micro-service-personne` dans `Artifact`
  - `com.example.demo` dans `Package`
- Cliquer sur `Next`
- Chercher et cocher les cases correspondantes aux `Spring Data JPA`, `MySQL Driver`, `Lombok`, `Spring Web`, `Spring Boot DevTools` et `Eureka Discovery Client`
- Cliquer sur `Next` puis sur `Finish`

Spring Boot

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Spring Boot

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Pour la suite, nous considérons

- une entité `Personne` à définir dans `com.example.demo.model`
- une interface DAO `PersonneRepository` à définir dans `com.example.demo.dao`
- un contrôleur REST `PersonneController` à définir dans `com.example.demo.dao`

Spring Boot & REST

Créons une entité `Personne` **dans** `com.example.demo.model`

```
package com.example.demo.model;

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

@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long num;
    @NonNull
    private String nom;
    @NonNull
    private String prenom;
}
```

Spring Boot & REST

Préparons notre interface DAO `PersonneRepository`

```
package com.example.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne,
   Long> {

}
```

Créons le contrôleur REST suivant

```
@RestController
@AllArgsConstructor
public class PersonneRestController {

        private PersonneRepository personneRepository;

        @GetMapping("/personnes")
        public List<Personne> getPersonnes() {
                return personneRepository.findAll();
        }

        @GetMapping("/personnes/{id}")
        public Personne getPersonne(@PathVariable("id") long id) {
                return personneRepository.findById(id).orElse(null);
        }

        @PostMapping("/personnes")
        public Personne addPersonne(@RequestBody Personne personne) {
                return personneRepository.save(personne);
        }
}
```

Spring Boot

Dans `application.properties`, ajoutons les données permettant la connexion à la base de données et la configuration de `Hibernate`

```
server.servlet.context-path=/ws
server.port=8081
spring.datasource.url = jdbc:mysql://localhost:3306/microservice_personne?
  createDatabaseIfNotExist=true
spring.datasource.username = root
spring.datasource.password =
spring.jpa.hibernate.ddl-auto = create-drop
spring.jpa.show-sql = true
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.model.naming.
  PhysicalNamingStrategyStandardImpl
```


L'ajout de la propriété `spring.jpa.hibernate.naming.physical-strategy` permet de forcer **Hibernate** à utiliser les mêmes noms pour les tables et les colonnes que les entités et les attributs.

Spring Boot

Dans `application.properties`, définissons un nom pour notre application

```
spring.application.name=PERSONNE-SERVICE
```

Pour commencer, on ne publie pas le microservice

```
spring.cloud.discovery.enabled=false
```

Spring Boot

Commençons par modifier le point d'entrée (qui implémentera l'interface `ApplicationRunner`) pour alimenter la base de données avec quelques données de test

```
@SpringBootApplication
public class MicroServicePersonneApplication {

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

  @Bean
  ApplicationRunner start(PersonneRepository repository) {
    return args -> {
      repository.save(new Personne("Wick", "John"));
      repository.save(new Personne("El Mouelhi", "Achref"));
    };
  }
}
```

Spring Boot

Pour tester, allez à

- `http://localhost:8081/ws/personnes`
- Ou `http://localhost:8081/ws/personnes/1`

Spring Boot

Création de projet Spring Boot

- Aller dans `File > New > Other`
- Chercher `Spring`, dans `Spring Boot` sélectionner `Spring Starter Project` et cliquer sur `Next >`
- Saisir
  - `micro-service-adresse` dans `Name`,
  - `com.example` dans `Group`,
  - `micro-service-adresse` dans `Artifact`
  - `com.example.demo` dans `Package`
- Cliquer sur `Next`
- Chercher et cocher les cases correspondantes aux `Spring Data JPA`, `MySQL Driver`, `Lombok`, `Spring Web`, `Spring Boot DevTools`, `Eureka Discovery Client` et `OpenFeign`
- Cliquer sur `Next` puis sur `Finish`

Spring Boot

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Spring Boot

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Pour la suite, nous considérons

- une entité `Adresse` à définir dans `com.example.demo.model`
- une interface DAO `AdresseRepository` à définir dans `com.example.demo.dao`
- un contrôleur REST `AdresseController` à définir dans `com.example.demo.dao`

Spring Boot & REST

Créons une entité `Adresse` **dans** `com.example.demo.model`

```
package com.example.demo.model;

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

@NoArgsConstructor
@AllArgsConstructor
@RequiredArgsConstructor
@Data
@Entity
public class Adresse {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @NonNull
    private String rue;
    @NonNull
    private String codePostal;
    @NonNull
    private String ville;
}
```

Spring Boot & REST

Comme ce microservice ne persiste pas les personnes, on va créer une classe `Personne` (qui n'a pas l'annotation `@Entity`) dans `com.example.demo.model`

```
package com.example.demo.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
@Data
@RequiredArgsConstructor
public class Personne {
    private Long num;
    @NonNull
    private String nom;
    @NonNull
    private String prenom;
}
```

Spring Boot & REST

Modifions l'entité `Adresse` en ajoutant l'association avec `Personne` dans

```
@NoArgsConstructor
@AllArgsConstructor
@RequiredArgsConstructor
@Data
@Entity
public class Adresse {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @NonNull
    private String rue;
    @NonNull
    private String codePostal;
    @NonNull
    private String ville;
    @NonNull
    private Long idPersonne;
    @Transient
    private Personne personne;
}
```

Spring Boot & REST

Préparons notre interface DAO `AdresseRepository`

```
package com.example.demo.dao;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Adresse;

public interface AdresseRepository extends JpaRepository<Adresse, Long>
    {

}
```

Spring Boot

Dans `application.properties`, ajoutons les données permettant la connexion à la base de données et la configuration de `Hibernate`

```
server.servlet.context-path=/ws
server.port=8082
spring.datasource.url = jdbc:mysql://localhost:3306/microservice_adresse?
   createDatabaseIfNotExist=true
spring.datasource.username = root
spring.datasource.password =
spring.jpa.hibernate.ddl-auto = create-drop
spring.jpa.show-sql = true
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.model.naming.
   PhysicalNamingStrategyStandardImpl

# microservice
spring.application.name=ADRESSE-SERVICE
spring.cloud.discovery.enabled=false
```

L'ajout de la propriété `spring.jpa.hibernate.naming.physical-strategy` permet de forcer **Hibernate** à utiliser les mêmes noms pour les tables et les colonnes que les entités et les attributs.

Spring Boot

Commençons par modifier le point d'entrée (qui implémentera l'interface `ApplicationRunner`) pour alimenter la base de données avec quelques données de test

```
@SpringBootApplication
public class MicroServiceAdresseApplication {

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

  @Bean
  ApplicationRunner start(AdresseRepository repository) {
    return args -> {
      repository.save(new Adresse("prado", "13008", "Marseille", 1L));
      repository.save(new Adresse("plantes", "75014", "Paris", 2L));
    };
  }
}
```

Spring Boot

Pour activer Open Feign, ajoutons l'annotation `@EnableFeignClients` à la classe de démarrage

```
@EnableFeignClients
@SpringBootApplication
public class MicroServiceAdresseApplication {

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

  @Bean
  ApplicationRunner start(AdresseRepository repository) {
    return args -> {
      repository.save(new Adresse("prado", "13008", "Marseille", 1L));
      repository.save(new Adresse("plantes", "75014", "Paris", 2L));
    };
  }
}
```

Spring Boot

Créons une interface cliente `PersonneRestClient` qui va assurer la communication avec le premier microservice

```
package com.example.demo.openfeign;

import java.util.List;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

import com.example.demo.model.Personne;

@FeignClient(name = "PERSONNE-SERVICE")
public interface PersonneRestClient {

        @GetMapping("/ws/personnes")
        public List<Personne> getPersonnes();

        @GetMapping("/ws/personnes/{id}")
        public Personne getPersonne(@PathVariable("id") long id);

}
```

Spring Boot

Créons un contrôleur REST `AdresseRestController` dans lequel nous injectons `AdresseRepository`

```
@RestController
@AllArgsConstructor
public class AdresseRestController {

    private AdresseRepository adresseRepository;

}
```

Spring Boot

Pour lier les adresses aux personnes, injectons aussi `PersonneRestClient`

```
@RestController
@AllArgsConstructor
public class AdresseRestController {

    private PersonneRestClient personneRestClient;
    private AdresseRepository adresseRepository;

}
```

Spring Boot

Ajoutons maintenant les trois actions suivantes

```
@GetMapping("/adresses")
public List<Adresse> getAdresses() {
  var adresses = adresseRepository.findAll();
  for (Adresse a : adresses) {
    a.setPersonne(personneRestClient.getPersonne(a.getIdPersonne()));
  }
  return adresses;
}

@GetMapping("/adresses/{id}")
public Adresse getAdresse(@PathVariable("id") long id) {
  var a = adresseRepository.findById(id).orElse(null);
  a.setPersonne(personneRestClient.getPersonne(a.getIdPersonne()));
  return a;
}

@PostMapping("/adresses")
public Adresse addAdresse(@RequestBody Adresse adresse) {
  return adresseRepository.save(adresse);
}
```

Spring Boot

Pour tester

- Démarrer l'application
- Vérifier, dans la console, que les tuples de la classes de démarrage ont été insérés
- Faire une requête `GET` à `http://localhost:8082/ws/adresses` lance une exception car les microservices ne sont pas déclarés

Spring Boot

Pour tester

- Démarrer l'application
- Vérifier, dans la console, que les tuples de la classes de démarrage ont été insérés
- Faire une requête `GET` à `http://localhost:8082/ws/adresses` lance une exception car les microservices ne sont pas déclarés

Pour résoudre le problème précédent

Il faut créer un projet **Eureka Discovery** (Spring Cloud).

Spring Boot

Création de projet Spring Boot

- Aller dans `File > New > Other`
- Chercher `Spring`, dans `Spring Boot` sélectionner `Spring Starter Project` et cliquer sur `Next >`
- Saisir
  - `eureka-service` dans `Name`,
  - `com.example` dans `Group`,
  - `eureka-service` dans `Artifact`
  - `com.example.demo` dans `Package`
- Cliquer sur `Next`
- Chercher et cocher les cases correspondantes à `Eureka Server` et `DevTools`
- Cliquer sur `Next` puis sur `Finish`

Spring Boot

Commençons par activer `Eureka Server` dans la classe de démarrage

```
package com.example.demo;

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

@SpringBootApplication
@EnableEurekaServer
public class EurekaServiceApplication {

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

}
```

Spring Boot

Les propriétés à ajouter dans `application.properties`

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

Spring Boot

Les propriétés à ajouter dans `application.properties`

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

Explication

- `8761` : numéro de port par défaut d'**Eureka**
- `eureka.client.fetch-registry=false` : pour ne pas enregistrer le serveur en tant que client
- `eureka.client.register-with-eureka=false` : pour ne pas l'enregistrer dans le registre des services

Spring Boot

Dans `application.properties` **de** `micro-service-personne` **et** `micro-service-adresse`**, mettons la propriété suivante à** `true`

```
spring.cloud.discovery.enabled=true
```

Spring Boot

Dans `application.properties` **de** `micro-service-personne` **et** `micro-service-adresse`**, mettons la propriété suivante à** `true`

```
spring.cloud.discovery.enabled=true
```


Pour tester

1. démarrer le projet `eureka-service`
2. démarrer le projet `micro-service-personne`
3. démarrer le projet `micro-service-adresse`

Spring Boot

Allez à `http://localhost:8082/ws/adresses` et vérifiez qu'on récupère les adresses avec les personnes


```
[
  {
    "id": 1,
    "rue": "paradis",
    "codePostal": "13006",
    "ville": "Marseille",
    "idPersonne": 1,
    "personne": {
      "num": 1,
      "nom": "Wick",
      "prenom": "John"
    }
  },
  {
    "id": 2,
    "rue": "plantes",
    "codePostal": "75014",
    "ville": "Paris",
    "idPersonne": 2,
    "personne": {
      "num": 2,
      "nom": "El Mouelhi",
      "prenom": "Achref"
    }
  }
]
```

Spring Boot

Remarque

1. Allez à `http://localhost:8761/`
2. Vérifiez la présence de deux services dans `Instances currently registered with Eureka`

Spring Boot

Constat

Chaque microservice a sa propre **URL** : host + port + ...

Spring Boot

Constat

Chaque microservice a sa propre **URL** : host + port + ...

Comment utiliser la même URL avec un path unique pour chaque microservice ?

Utiliser le routage dynamique avec le **Gateway**

Spring Boot

Création de projet Spring Boot

- Aller dans File > New > Other
- Chercher Spring, dans Spring Boot sélectionner Spring Starter Project et cliquer sur Next >
- Saisir
  - gateway-routing dans Name,
  - com.example dans Group,
  - gateway-routing dans Artifact
  - com.example.demo dans Package
- Cliquer sur Next
- Chercher et cocher les cases correspondantes aux Reactive Gateway, DevTools et Eureka Discovery Client
- Cliquer sur Next puis sur Finish

Spring Boot

Ajoutons le bean suivant dans la classe de démarrage pour activer et utiliser le routage dynamique

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.ReactiveDiscoveryClient;
import org.springframework.cloud.gateway.discovery.DiscoveryClientRouteDefinitionLocator;
import org.springframework.cloud.gateway.discovery.DiscoveryLocatorProperties;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class GatewayRoutingApplication {

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

        @Bean
        DiscoveryClientRouteDefinitionLocator dynamicRoute(
            ReactiveDiscoveryClient rdc,
            DiscoveryLocatorProperties dlp) {

                return new DiscoveryClientRouteDefinitionLocator(rdc, dlp);
        }

}
```

Spring Boot

Les propriétés à ajouter dans `application.properties`

```
server.port=8080
spring.application.name=GATEWAY
spring.cloud.discovery.enabled=true
eureka.instance.prefer-ip-address=true
```

Spring Boot

Ordre de lancement des projets

1. démarrer le projet `eureka-service`
2. démarrer le projet `micro-service-personne`
3. démarrer le projet `micro-service-adresse`
4. démarrer le projet `gateway`

Spring Boot

Ordre de lancement des projets

1. démarrer le projet `eureka-service`
2. démarrer le projet `micro-service-personne`
3. démarrer le projet `micro-service-adresse`
4. démarrer le projet `gateway`

Pour tester

1. Pour récupérer la liste d'adresses :
   `http://localhost:8080/ADRESSE-SERVICE/ws/adresses`
2. Pour récupérer la liste de personnes :
   `http://localhost:8080/PERSONNE-SERVICE/ws/personnes`