LaTeX Exercise 1

April 15, 2019

1 Introduction

Pour ce tutoriel nous allons créer deux microservices:

1. Service Échange
2. Service Convertisseur monnaie

1.1 Service Échange - SE

Ce service va nous fournir les valeurs de conversions. Prenons une requête simple:

```
GET to http://localhost:8000/currency-exchange/from/BIT/to/EUR
```

doit retourner:

```
{
  id: 10002,
  from: "BIT",
  to: "EUR",
  conversionMultiple: 5000,
  port: 8000,
}
```

Cette requête retourne le multiple de conversion entre le Bitcoin et l'euro.

1.2 Service convertisseur monnaie - SCM

Ce service peut convertir une somme d'argent vers une autre monnaie. Par exemple pour la requête:

```
GET to http://localhost:8100/currency-converter/from/BIT/to/EUR/
    quantity/10
```

```
{
  id: 10002,
  from: "EUR",
  to: "BIT",
  conversionMultiple: 5000,
  quantity: 10,
  totalCalculatedAmount: 50000,
  port: 8000,
}
```

Pour faire ce calcul on utilise le microservice précédent. SCM utilise l'API de SE pour avoir le multiple de conversion et faire le calcul. Une veux de l'architecture de microservices:

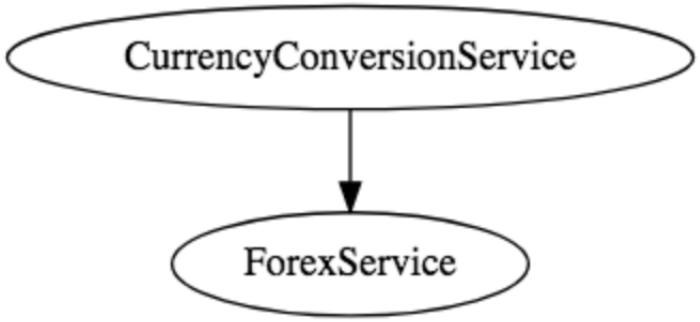

Figure 1: Architecture microservice

2 Spring Cloud

Pour faire nos microservices, nous allons utiliser le Spring Cloud framework [1] avec Maven. Maven est un outil de gestion et d'automatisation de production des projets logiciels Java. Pour notre premier microservice nous allons générer un projet Maven avec toutes les dépendances nécessaires en utilisant SpringInitializr. Sur le site on peut sélectionner les dépendances (les librairies que l'on souhaite utiliser : Web, DevTools, JPA, H2, Feign, Ribbon et Netflix Discovery. On nomme le projet service-converteur-monnaie. Enfin on génère le projet.

[1] SpringCloud

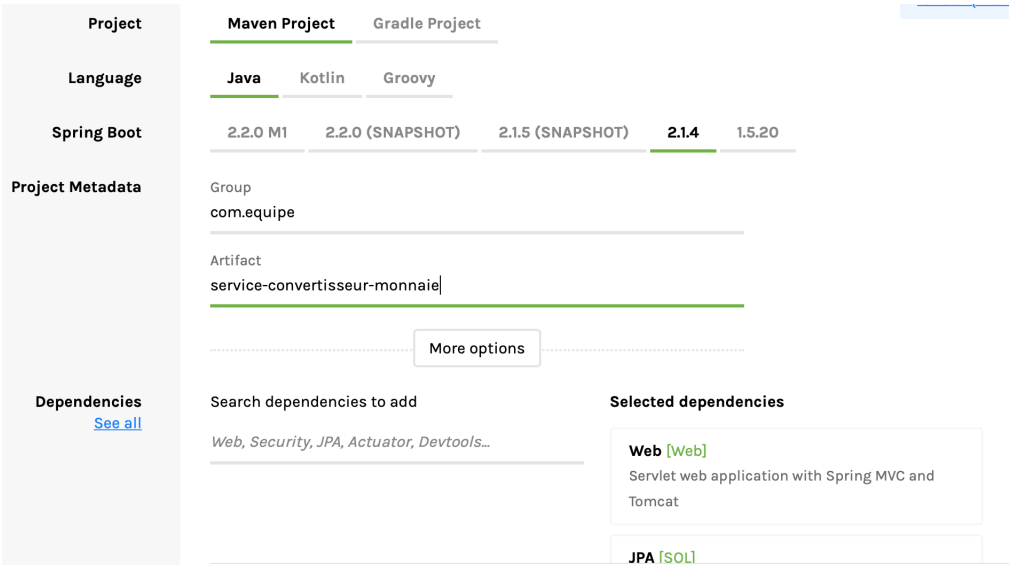

Figure 2: SpringInitialzr

2.1 Le premier microservice

Maintenant, il faut importer le projet sur Eclipse en tant que projet Maven. Pour ce microservice il faut définir une entité qui va représenter notre valeur d'échange.

```
@Entity
public class ExchangeValue {

  @Id
  private Long id;

  @Column(name="currency_from")
  private String from;

  @Column(name="currency_to")
  private String to;

  private BigDecimal conversionMultiple;
  private int port;

  public ExchangeValue() {

  }
```

```
  public ExchangeValue(Long id, String from, String to,
   BigDecimal conversionMultiple) {
    super();
    this.id = id;
    this.from = from;
    this.to = to;
    this.conversionMultiple = conversionMultiple;
  }

  public Long getId() {
    return id;
  }

  public String getFrom() {
    return from;
  }

  public String getTo() {
    return to;
  }

  public BigDecimal getConversionMultiple() {
    return conversionMultiple;
  }

  public int getPort() {
    return port;
  }

  public void setPort(int port) {
    this.port = port;
  }

}
```

- @Entity : Cette annotation nous permet de spécifier une table dans la base de données de H2 avec la librairie JPA (JAVA Persistence API).

- @Id : Cette annotation permet de spécifier la clef primaire.

On définit aussi une interface pour interagir avec la base de données:

```
public interface ExchangeValueRepository extends
    JpaRepository<ExchangeValue, Long>{
  ExchangeValue findByFromAndTo(String from, String to);
}
```

Enfin nous définissons le controller REST qui nous permettra d'accéder aux valeurs ExchangeValue pour les retourner :

```
@RestController
public class ForexController {

  @Autowired
  private Environment environment;

  @Autowired
  private ExchangeValueRepository repository;

  @GetMapping("/currency-exchange/from/{from}/to/{to}")
  public ExchangeValue retrieveExchangeValue
    (@PathVariable String from, @PathVariable String to){

    ExchangeValue exchangeValue =
        repository.findByFromAndTo(from, to);

    exchangeValue.setPort(
        Integer.parseInt(environment.getProperty("local.server.
    port")));

    return exchangeValue;
  }
}
```

- @RestController : Cette annotation définit le controller.

- @GetMapping : Cette annotation définit notre API GET.

Maintenant que le logiciel métier est écrit, il faut configurer l'application. Il faut mettre ce code dans "application.properties":

```
spring.application.name=service-converteur-monnaie
server.port=8000

spring.jpa.show-sql=true
spring.h2.console.enabled=true
```

Puis nous allons rentrer des valeurs dans la base de données. Il faut créer un fichier "data.sql" avec des données qui seront chargée lors du lancement du microservice.

```
insert into exchange_value(id,currency_from,currency_to,
    conversion_multiple,port)
values(10001,'BIT','EUR',5000,0);
insert into exchange_value(id,currency_from,currency_to,
    conversion_multiple,port)
values(10002,'BIT','USD',4000,0);
insert into exchange_value(id,currency_from,currency_to,
    conversion_multiple,port)
values(10003,'BIT','AUD',7000,0);
```

il faut lancer le projet sur un serveur Tomcat. Pour tester l'API, n'importe quel navigateur suffit avec l'URL suivant:

```
GET to http://localhost:8000/currency-exchange/from/BIT/to/EUR
```

3 Le deuxième microservice

En utilisant SpringInitializr, on crée le deuxième microservice avec les mêmes dépendances. On définit notre JAVA Bean pour manipuler les valeurs de retour du premier microservice.

```
public class CurrencyConversionBean {
  private Long id;
  private String from;
  private String to;
  private BigDecimal conversionMultiple;
  private BigDecimal quantity;
  private BigDecimal totalCalculatedAmount;
  private int port;

  public CurrencyConversionBean() {

  }

  public CurrencyConversionBean(Long id, String from, String to,
    BigDecimal conversionMultiple, BigDecimal quantity,
      BigDecimal totalCalculatedAmount, int port) {
    super();
    this.id = id;
    this.from = from;
    this.to = to;
    this.conversionMultiple = conversionMultiple;
    this.quantity = quantity;
    this.totalCalculatedAmount = totalCalculatedAmount;
    this.port = port;
  }
}
```

Il faut définir le controller REST avec une methode GET qui retourne un CurrencyConversionBean :

```
@RestController
public class CurrencyConversionController {
@GetMapping("/currency-converter/from/{from}/to/{to}/quantity/{
    quantity}")
  public CurrencyConversionBean convertCurrency(@PathVariable
    String from, @PathVariable String to,
      @PathVariable BigDecimal quantity) {

    Map<String, String> uriVariables = new HashMap<>();
    uriVariables.put("from", from);
    uriVariables.put("to", to);

    ResponseEntity<CurrencyConversionBean> responseEntity = new
    RestTemplate().getForEntity(
        "http://localhost:8000/currency-exchange/from/{from}/to
    /{to}", CurrencyConversionBean.class,
        uriVariables);

    CurrencyConversionBean response = responseEntity.getBody();

    return new CurrencyConversionBean(response.getId(), from, to
    , response.getConversionMultiple(), quantity,
        quantity.multiply(response.getConversionMultiple()),
    response.getPort());
}
```

Il faudra aussi définir la propriété du microservice :

```
spring.application.name=service-echange
server.port=8100
```

3.1 Feign Client

Pour simplifier la consommation d'API on peut utiliser la technologie Feign. Avec Feign on peut définir une interface qui décrit le service web que l'on souhaite consommer.

```
@FeignClient(name="service-converteur-monnaie" url="localhost
    :8000")
public interface CurrencyExchangeServiceProxy {
  @GetMapping("/currency-exchange/from/{from}/to/{to}")
  public CurrencyConversionBean retrieveExchangeValue
```

```
    (@PathVariable("from") String from, @PathVariable("to")
    String to);
}
```

Maintenant on peut consommer le service dans notre controller en utilisant cette ligne de code :

```
CurrencyConversionBean response = proxy.retrieveExchangeValue(
    from, to);
```

et en definissant le proxy en tant qu'attribut:

```
@Autowired
private CurrencyExchangeServiceProxy proxy;
```

Dans la classe principale, il faut autoriser les clients Feign en ajoutant l'annotation suivante:

```
@SpringBootApplication
@EnableFeignClients("<package>.currencyconversion")
@EnableDiscoveryClient
public class SpringBootMicroserviceCurrencyConversionApplication
    {

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

4 Statique vs Dynamique

Un des aspects importants des microservices c'est la facilité de dupliquer des instances de microservices. En fonction de nos besoins nous pouvons lancer plusieurs instances de nos microservices. Jusqu'à présent nos microservices se connectent de manière statique en allant sur une adresse statique. Pour correctement distribuer les appels vers les multiples instances on peut utiliser la technologies de Ribbon. Ribbon est un équilibreur de charge côté client. Une fois installé, il va pouvoir aller consulter la liste des instances disponibles pour un microservice pour les choisir à tour de rôle afin d'équilibrer la charge.

Ajoutons l'annotation a l'interface Feign dans le microservice échange:

```
@RibbonClient(name="service-converteur-monnaie")
```

Il faudra ensuite lancer deux instances de service-converteur-monnaie avec le port 8000 et 8001. Puis nous ajoutons a la propriété du microservice echange la ligne suivante :

```
service-converteur-monnaie.ribbon.listOfServers=localhost:8000,
    localhost:8001
```

Si nous voulons augmenter le nombre d'instances nous devons relancer le microservice service-echange avec les ports de chaque nouvelles instances de service-converteur-monnaie. Pour éviter ce problème et rendre cette architecture plus dynamique nous pouvons utiliser un équilibreur de charge côté serveur. Une technologie qui facilite cette tache est Eureka par Netflix.

Retournons sur SpringInitializr pour générer un microservice spécialisé. Ce microservice doit avoir pour dépendances Eureka et DevTools.

Ce microservice doit contenir l'annotation suivante dans sa classe principale :

```
@SpringBootApplication
@EnableEurekaServer
public class SpringBootMicroserviceEurekaNamingServerApplication
    {
...
}
```

et les propriétés suivantes:

```
spring.application.name=netflix-eureka-naming-server
server.port=8761

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Maintenant, lancez l'application et utilisez votre navigateur pour consulter la page du serveur Eureka. Si tous fonctionne correctement, le serveur devrait décrire les instances connectées sur Eureka. Vous pouvez ajouter aux fichiers de configuration des deux microservices la propriete suivante pour indiquer l'url du serveur eureka :

```
eureka.client.service-url.default-zone=http://localhost:8761/
    eureka
```

Enfin on peut supprimer la liste statique des ports dans l'application service-echange :

```
service-converteur-monnaie.ribbon.listOfServers=localhost:8000,
    localhost:8001
enlever
```