Atelier 2
Faire communiquer les Microservices grâce à Feign

Feign est un client HTTP qui facilite grandement l'appel des API exposées par les différents Microservices. Il est donc capable de créer et d'exécuter des requêtes HTTP basées sur les annotations et informations que l'on fournit. C'est un peu l'équivalent en code de Postman.

Il se présente sous forme de dépendance à ajouter au Microservice.

Commençons par ajouter Feign à notre *pom.xml.* Dans ce cas, il ne suffira pas d'ajouter le starter Feign. Il faudra ajouter certaines modifications afin d'assurer les compatibilités. Pour obtenir un *pom.xml* avec toutes les dépendances nécessaires, rien de mieux qu'un **Spring Initializr.**

Vous obtenez alors ce *pom.xml :*

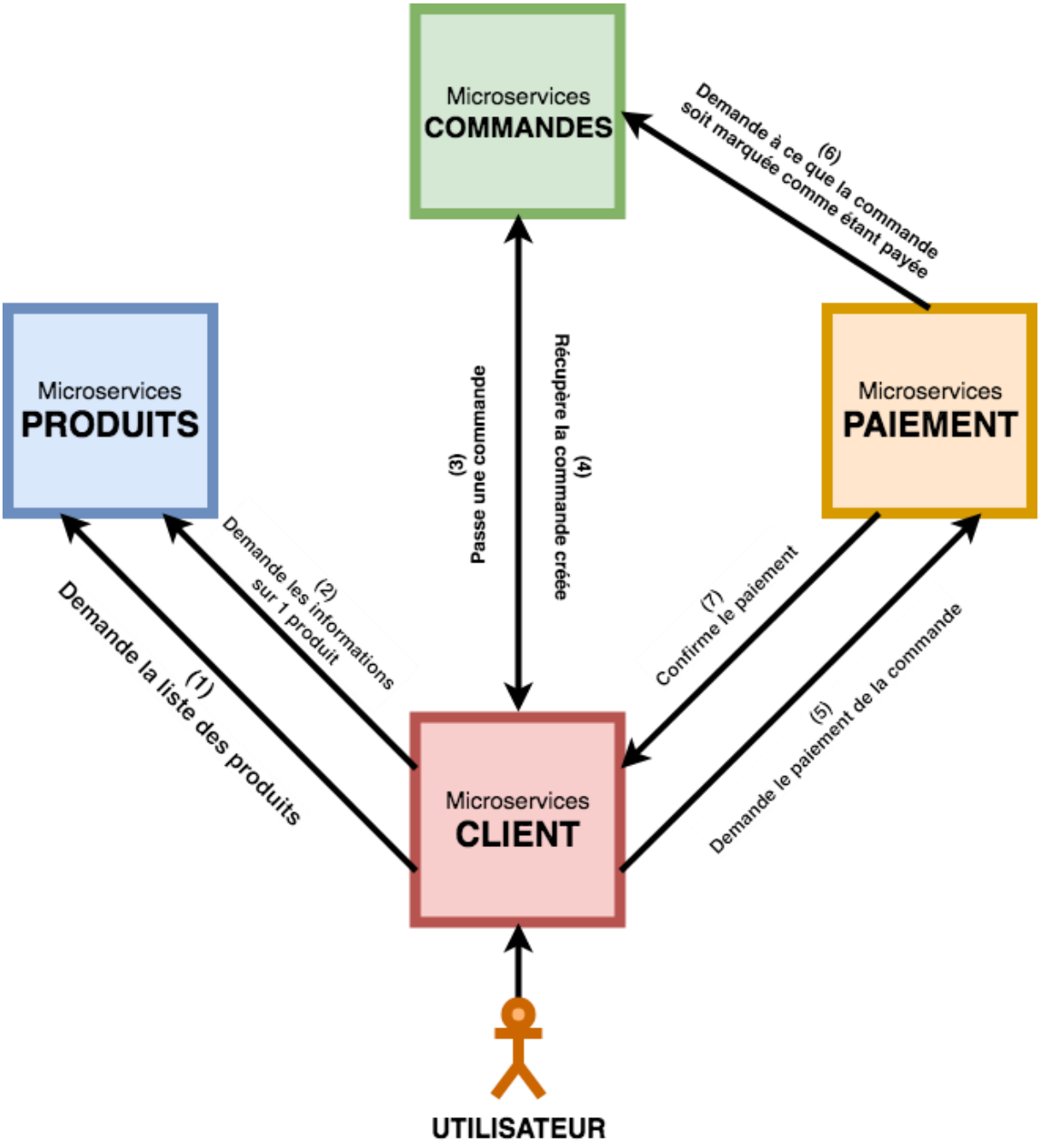
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.clientui</groupId>
   <artifactId>clientui</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <packaging>jar</packaging>
   <name>client-ui</name>
   <description>Client UI de l'application</description>
   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.0.0.RELEASE</version>
      <relativePath />
      <!-- lookup parent from repository -->
   </parent>
   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
      <java.version>1.8</java.version>
      <spring-cloud.version>Finchley.M8</spring-cloud.version>
   </properties>
   <dependencies>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-thymeleaf</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
      <dependency>
         <groupId>org.webjars</groupId>
         <artifactId>bootstrap</artifactId>
         <version>4.0.0-2</version>
      </dependency>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-starter-openfeign</artifactId>
      </dependency>
```

```
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-test</artifactId>
         <scope>test</scope>
      </dependency>
   </dependencies>
   <dependencyManagement>
      <dependencies>
         <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
         </dependency>
      </dependencies>
   </dependencyManagement>
   <build>
      <plugins>
         <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
         </plugin>
      </plugins>
   </build>
   <repositories>
      <repository>
         <id>spring-milestones</id>
         <name>Spring Milestones</name>
         <url>https://repo.spring.io/milestone</url>
         <snapshots>
            <enabled>false</enabled>
         </snapshots>
      </repository>
   </repositories>
</project>
```

Afin d'activer Feign dans ce Microservice, rendez-vous à *ClientUiApplication* et ajoutez l'annotation ***@EnableFeignClients :***

```
@SpringBootApplication
@EnableFeignClients("com.clientui")
public class ClientUiApplication {

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

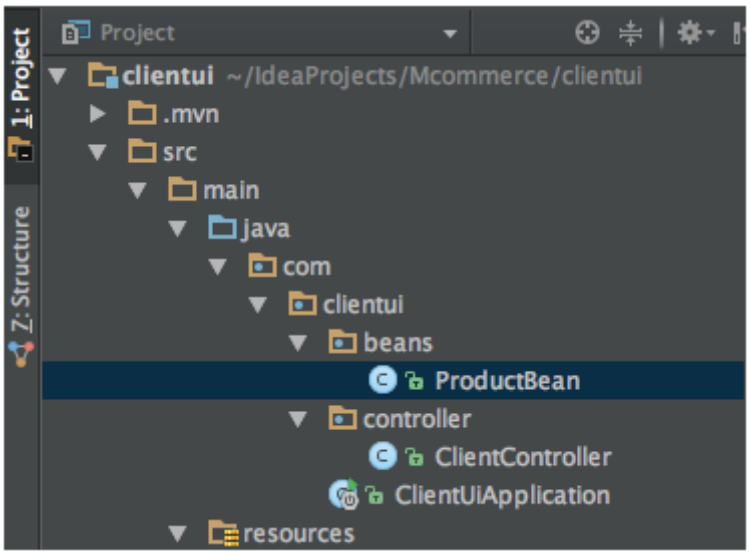
L'annotation `@EnableFeignClients` demande à Feign de scanner le package *"com.clientui"* pour rechercher des classes qui se déclarent clients Feign. Nous allons justement en créer une plus tard.
Voici, pour rappel, les étapes que nous avons définies pour passer une commande :



1. Récupérer la liste des produits (Étape 1)

Quand Feign fera appel à *Microservice-produits* afin de récupérer la liste des produits, il lui faudra stocker chaque produit dans un objet de type `Product` afin que nous puissions les manipuler facilement plus tard (vous conviendrez que si Feign nous retourne le JSON brut, il ne nous sert pas à grand-chose).
Nous allons donc créer un bean qui reprend les mêmes champs que *Product.java.*

Créez une classe `ProductBean` sous un package **"beans" :**



Voici notre `ProductBean` :

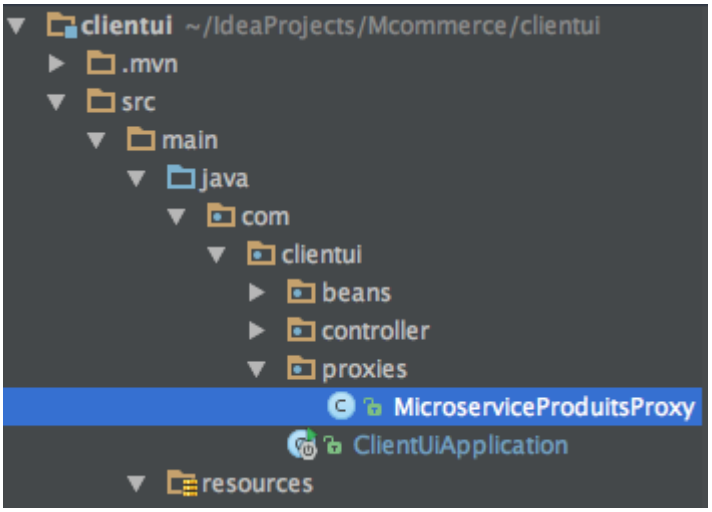
```
package com.clientui.beans;


public class ProductBean {

  private int id;

  private String titre;

  private String description;

  private String image;

  private Double prix;

  public ProductBean() {
  }

  public int getId() {
      return id;
  }

  public void setId(int id) {
      this.id = id;
  }

  public String getTitre() {
      return titre;
  }

  public void setTitre(String titre) {
      this.titre = titre;
  }

  public String getDescription() {
      return description;
  }

  public void setDescription(String description) {
      this.description = description;
  }

  public String getImage() {
```

```
        return image;
    }

    public void setImage(String image) {
        this.image = image;
    }

    public Double getPrix() {
        return prix;
    }

    public void setPrix(Double prix) {
        this.prix = prix;
    }

    @Override
    public String toString() {
        return "ProductBean{" +
                "id=" + id +
                ", titre='" + titre + '\'' +
                ", description='" + description + '\'' +
                ", image='" + image + '\'' +
                ", prix=" + prix +
                '}';
    }
}
```

Nous avons repris les mêmes champs que dans *Product.java,* puis nous avons généré les *Getters* et *Setters.*

Nous allons maintenant créer une **interface qui va regrouper les requêtes** que nous souhaitons passer au *Microservice-produits.* Cette interface est ce que nous appelons un **proxy,** car elle se positionne comme une classe intermédiaire qui fait le lien avec les Microservices extérieurs à appeler.

Créez une classe `MicroserviceProduitsProxy` sous un package **"proxies" :**



Voici le code que nous allons utiliser dans ce proxy :

```
package com.clientui.proxies;

import com.clientui.beans.ProductBean;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

import java.util.List;
import java.util.Optional;

@FeignClient(name = "microservice-produits", url = "localhost:9001")
public interface MicroserviceProduitsProxy {

@GetMapping(value = "/Produits")
List<ProductBean> listeDesProduits();

@GetMapping( value = "/Produits/{id}")
ProductBean recupererUnProduit(@PathVariable("id") int id);

}
```

Explications :

- `@FeignClient` déclare cette interface comme client Feign. Feign utilisera les informations fournies ici pour construire les requêtes HTTP appropriées afin d'appeler le *Microservice-Produits.*
  On donne à cette annotation 2 paramètres : le premier est "name", il s'agit du nom du Microservice à appeler. Il ne s'agit pas ici de n'importe quel nom, mais du nom "officiel" qui sera utilisé plus tard par des Edge Microservices comme Eureka et Ribbon.
  Celui-ci est à renseigner dans *application.properties* du Microservice à appeler grâce à *`spring.application.name`*.

  Voici donc à quoi ressemble ce fichier dans Microservice-produits :

  ```
  spring.application.name=microservice-produits

  server.port 9001

  #Configurations H2
  spring.jpa.show-sql=true
  spring.h2.console.enabled=true

  #défini l'encodage pour data.sql
  spring.datasource.sql-script-encoding=UTF-8
  ```

  Le deuxième paramètre est l'URL du Microservice (localhost:9001). Vous comprenez maintenant pourquoi nos Microservices écoutent des ports différents. Même s'ils partagent le même domaine, Feign (et d'autres composants) pourra les différencier grâce aux ports.

- Dans cette interface créée, il faut déclarer les signatures des opérations à appeler dans le Microservice "produits". Dans notre cas, comme nous avons accès au code source de *Microservice-produits,* il suffit de copier ses signatures.

Néanmoins, même sans avoir accès aux sources, il suffit de préciser les types de retour, par exemple *List*, un nom quelconque pour votre méthode et un URI. Toutes ces informations sont normalement disponibles dans la documentation qui accompagne chaque Microservice.

Si vous copiez les signatures, il faut veiller à remplacer *Product* par son équivalent dans ce client `ProductBean` que nous avons créé.

Remarquez que j'ai changé la signature avec *Optional* par *ProductBean* dans la deuxième méthode, afin de simplifier son utilisation.

Très bien ! Feign a désormais tout ce qu'il faut pour déduire qu'il faut une requête HTTP de type GET (grâce à GetMapping), à quelle URL l'envoyer et une fois la réponse reçue, dans quel objet la stocker (ProductBean).

Il ne reste plus alors qu'à utiliser ce proxy. **Revenez dans le contrôleur :**

```
@Controller
public class ClientController {

    @Autowired
    private MicroserviceProduitsProxy ProduitsProxy;

    @RequestMapping("/")
    public String accueil(Model model){

        List<ProductBean> produits =  ProduitsProxy.listeDesProduits();

        model.addAttribute("produits", produits);

        return "Accueil";
    }

}
```

Explications :

- Nous créons une variable de type `MicroserviceProduitsProxy` qui sera instanciée automatiquement par Spring.
- Nous avons donc maintenant accès à toutes les méthodes que nous avons définies dans `MicroserviceProduitsProxy`. Il suffit de faire appel à *listeDesProduits*. Feign ira exécuter la requête HTTP et nous renverra une liste de `ProductBean`.
- Nous utilisons la méthode `addAttribute` de model afin de passer en revue la liste des produits.

Rendez-vous maintenant dans *Accueil.html :*

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Mcommerce</title>

    <link rel="stylesheet" type="text/css" href="webjars/bootstrap/4.0.0-
2/css/bootstrap.min.css" />

</head>
<body>
```

```
<div class="container">

    <h1>Application Mcommerce</h1>

    <div class="row">
        <div th:each="produit : ${produits}" class="col-md-4 my-1">
            <a th:href="@{|/details-produit/${produit.id}|}" >
                <img th:src="${produit.image}"  class="card-img-top" />
                <p th:text= "${produit.titre}"></p>
            </a>
        </div>
    </div>

</div>


<script type="text/javascript" src="webjars/bootstrap/4.0.0-
2/js/bootstrap.min.js"></script>

</body>
</html>
```

Explications :

Dans le HTML, nous recevons grâce à *`model`* la variable *`produits`* avec la liste de tous les produits. Il suffit d'utiliser la syntaxe **Thymeleaf** pour parcourir cette liste et afficher les images des produits et leurs titres.

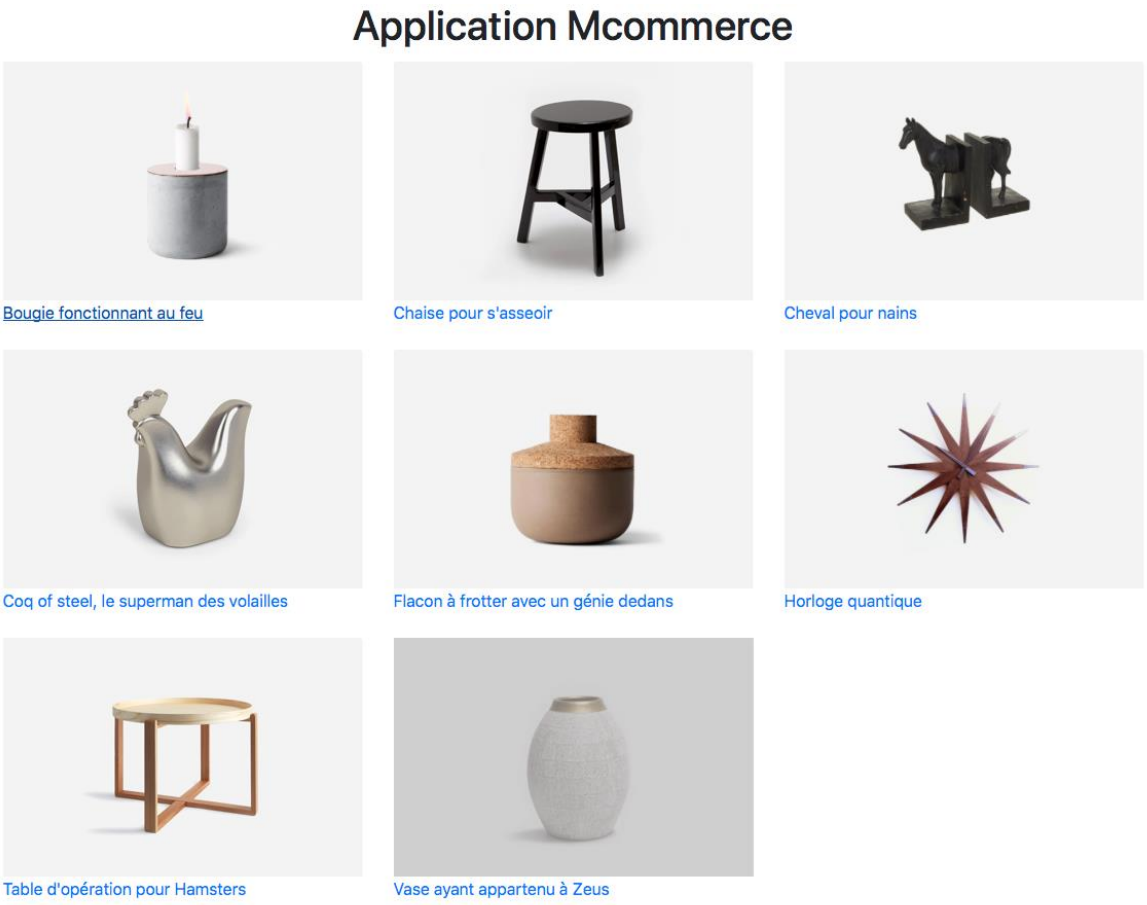
`th:each="produit : ${produits}"` parcourt la liste *produits* et stocke à chaque fois un objet de type ProductBean dans la variable `produit` .

On a accès ensuite aux attributs de chaque objet pour créer le lien vers chaque produit sous le format *"/details-produit/id_produit_ici"*. Nous créerons ensuite la méthode nécessaire pour cet URI dans notre contrôleur.

On ajoute également les images et les titres.

Si vous voulez aller plus loin, vous pouvez vous familiariser avec Thymeleaf grâce à sa documentation.

Vous devriez obtenir ceci :



N'oubliez pas de démarrer d'abord le *Microservice-produits !*

Félicitations ! Vous avez un client fonctionnel capable de faire appel à un autre Microservice, de récupérer et de formater les données reçues, et de les présenter dans une page web.

Nous venons donc de réaliser l'étape (1) de notre diagramme d'application.

2. Ajoutez la page de produit et de commande : étape (2) du diagramme

Vous avez maintenant tous les outils nécessaires pour ajouter une page qui affiche les détails d'un produit avec un bouton "commander", et une autre pour le retour après la commande.
Nous avons défini l'URL vers chaque produit dans le HTML grâce à :

```
<a th:href="@{|/details-produit/${produit.id}|}" >
```

Il faut donc créer une méthode dans le contrôleur qui répond aux URI de type : */details-produit/{id}*

ClientController.java

```java
  @RequestMapping("/details-produit/{id}")
   public String ficheProduit(@PathVariable int id,  Model model){

       ProductBean produit = ProduitsProxy.recupererUnProduit(id);

       model.addAttribute("produit", produit);

       return "FicheProduit";
   }
```

Explications :

Nous récupérons donc l'id passé dans l'URL du produit pour faire appel au *Microservice-produits* grâce à ProduitsProxy qui nous retourne les détails du produit en question (`recupererUnProduit(id)`).
Nous passons ensuite classiquement l'objet *`produit`* à *`model`*.
Puis nous demandons à ce que l'on affiche la page *FicheProduit.html*.

Voici la page HTML :

```html
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Mcommerce</title>

    <link rel="stylesheet" type="text/css"
          href="http://localhost:8080/webjars/bootstrap/4.0.0-2/css/bootstrap.min.css" />

</head>
<body>


<div class="container">

    <h1 class="text-center">Application Mcommerce</h1>

    <div class="row">
        <div class="col-md-4 mx-auto mt-5 text-center">

            <img th:src="${produit.image}" class="card-img-top" />

            <p th:text="${produit.titre}" class="font-weight-bold"></p>

            <p th:text="${produit.description}"></p>

            <p>
                <a th:href="@{|/details-produit/commander-produit/${produit.id}|}"
class="font-weight-bold">COMMANDER</a>
            </p>

        </div>
    </div>

</div>


<script type="text/javascript" src="http://localhost:8080/webjars/bootstrap/4.0.0-
2/js/bootstrap.min.js"></script>

</body>
</html>
```

Explications :

On affiche les détails du produit en accédant aux propriétés de celui-ci via la notation : `${produit.PROPRIÉTÉ}`.
On insère ensuite le lien de commande qui fera appel à une méthode dans notre contrôleur. Celui-ci s'occupera d'envoyer la requête GET vers le Microservice de commande, grâce à : `@{|/details-produit/commander-produit/${produit.id}|}`.

Vous devriez obtenir ce résultat :

3. Étapes (3) à (7) du diagramme

Pour les prochaines étapes, nous allons réutiliser les **mêmes principes** pour faire communiquer tous les Microservices.

Je vous invite à récupérer le code commenté qui explique étape par étape tout le processus dans la branche **ClientEtMSCommuniquant** de l'application.

4. Gestion et propagation des erreurs

Que se passe-t-il si vous demandez la récupération d'un produit qui n'existe pas ?

Faisons en sorte que *Microservice-produits* renvoie un code 400 Bad Request si le produit n'existe pas (nous allons éviter le 404, car son cas est particulier). **Pour ce faire, rendez-vous dans `ProductNotFoundException.java` et changez le code à renvoyer :**

```java
@ResponseStatus(HttpStatus.BAD_REQUEST)
public class ProductNotFoundException extends RuntimeException {


    public ProductNotFoundException(String message) {
        super(message);
    }
}
```

Essayons maintenant en appelant cette URL, par exemple sur Postman : http://localhost:8080/details-produit/20.

Vous recevez le code 500 en réponse, avec un corps de réponse comme celui-ci :

```
{
    "timestamp": "2018-04-15T23:29:07.112+0000",
    "status": 500,
    "error": "Internal Server Error",
    "message": "status 400 reading MicroserviceProduitsProxy#recupererUnProduit(int);
content:\n{\"timestamp\":\"2018-04-15T23:29:06.999+0000\",\"status\":400,\"error\":\"Bad
Request\",\"message\":\"Le produit correspondant à l'id 20 n'existe
pas\",\"path\":\"/Produits/20\"}",
    "path": "/details-produit/20"
}
```

Vous pouvez donc constater 2 choses :

- Le *Microservice-produit* a bien répondu comme prévu en renvoyant un code 400, mais Feign a tout simplement constaté que le code n'était pas au format 2XX. Feign a renvoyé un code 500 générique pour indiquer qu'un problème était survenu au niveau du serveur.

- Vous pouvez voir que le message d'erreur renvoyé par notre Microservice est stocké dans "message" et qu'il comporte le bon code.

Si vous vous rendez dans la console de ClientUI, vous trouvez cette erreur :

```
FeignException: status 400 reading MicroserviceProduitsProxy#recupererUnProduit(int);
```

L'exception que Feign a renvoyée est donc `FeignException`. Cette exception est celle que renvoie Feign à chaque fois que le code de retour est différent de 2XX.
Vous pouvez me dire que l'on peut se contenter dans ce cas du code 500, mais il faut penser aux cas où des Microservices appellent d'autres Microservices à la chaîne. Prenons l'exemple de *ClientUI* qui appelle *Microservice-paiement* pour enregistrer un paiement et qui, à son tour, appelle *Microservice-commande* pour passer le statut de la commande à "payée".

Si *Microservice-commande* renvoie par exemple un code disant que la commande est déjà payée, Feign générera le fameux code 500 et n'aura aucune chance d'informer

ClientUI de la nature du problème. ClientUI se retrouvera réduit à annoncer à l'utilisateur qu'un problème inconnu est survenu.

Or, si nous arrivons, dans *Microservice-produits,* à **décrypter l'exception générée par Feign** pour retomber sur les bons codes HTTP renvoyés, nous pourrons transmettre ce code en réponse à ClientUI qui affichera au client que le paiement n'a pas abouti, car la commande est déjà payée ! C'est ce que l'on appelle la propagation des erreurs à travers les Microservices.

Nous allons donc maintenant nous atteler à décoder l'exception générique de Feign afin de retomber sur les bons codes renvoyés.

Heureusement, Feign propose une interface nommée `ErrorDecoder` spécialement dédiée au décodage de la réponse HTTP afin de lancer l'exception de notre choix en fonction de la nature de l'erreur.

Créez donc votre propre décodeur qui viendra hériter de `ErrorDecoder` :

```
package com.clientui.exceptions;

import feign.Response;
import feign.codec.ErrorDecoder;


public class CustomErrorDecoder implements ErrorDecoder {

    private final ErrorDecoder defaultErrorDecoder = new Default();

    @Override
    public Exception decode(String invoqueur, Response reponse) {

        if(reponse.status() == 400 ) {
            return new ProductBadRequestException(
                    "Requête incorrecte "
            );
        }

        return defaultErrorDecoder.decode(invoqueur, reponse);
    }

}
```

Explications :

- On hérite de `ErrorDecoder`.
- On récupère une instance de `ErrorDecoder` . Si nous n'avons pas la capacité d'identifier le code d'erreur ou que, tout simplement, nous n'avons aucune exception prévue pour un code en particulier, on demande à ce que l'erreur soit traitée par le décodeur par défaut : `ErrorDecoder`.
- On implémente la méthode `decode` qui nous permet de récupérer le code d'erreur envoyé afin de lancer des exceptions en fonction de celui-ci.
  Cette méthode nous donne accès à un premier paramètre que j'ai appelé `invoqueur` et qui contient la classe et la méthode qui a généré la requête.
  Dans notre cas, par exemple, le contenu de `invoqueur` est

: `MicroserviceProduitsProxy#recupererUnProduit(int)`. En effet, c'est la méthode `recupererUnProduit` qui a été utilisée pour appeler `microservice-produits`.

- `reponse` contient donc la réponse de celui-ci. C'est la partie la plus importante, car elle va nous permettre de récupérer le code d'erreur. C'est exactement ce que l'on fait juste après.
- `reponse.status()` nous donne donc accès au code d'erreur renvoyé par le Microservice distant. Dans ce cas, nous vérifions s'il est égal à 400. Si c'est le cas, on lance une exception que nous allons créer et appeler `ProductBadRequestException`.
- Si l'erreur ne rentre pas dans nos critères, on la passe tout simplement au décodeur par défaut qui s'occupera de lancer l'exception par défaut vue plus haut : `FeignException`.

Créez enfin l'exception à renvoyer `ProductBadRequestException` :

```
package com.clientui.exceptions;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.BAD_REQUEST)
public class ProductBadRequestException extends RuntimeException{

    public ProductBadRequestException(String message) {
        super(message);
    }
}
```

Il s'agit là d'une exception classique, équivalente à celle créée dans *Microservice-produits,* par exemple. Elle renvoie tout simplement l'erreur *400 Bad Request* avec le message qui lui a été passé précédemment en argument.

Si vous testez maintenant, vous remarquerez que vous avez toujours la fameuse erreur 500. C'est normal, car il faut informer Spring de l'existence de notre propre décodeur `CustomErrorDecoder`. Nous allons donc **déclarer notre décodeur** afin que celui-ci soit utilisé à la place de celui par défaut.
Pour cela, nous allons tout simplement le mettre dans un **bean.**

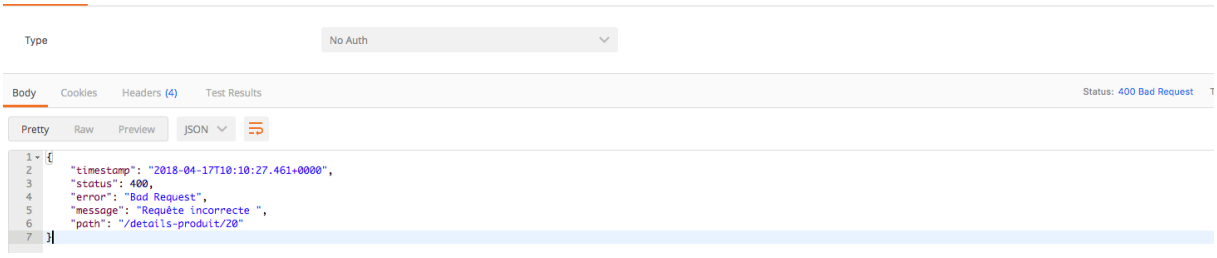
Créez une classe de configuration et appelez-la `FeignExceptionConfig` dans un package `configuration` :

```
package com.clientui.configuration;

import com.clientui.exceptions.CustomErrorDecoder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class FeignExceptionConfig {

    @Bean
    public CustomErrorDecoder mCustomErrorDecoder(){
        return new CustomErrorDecoder();
    }
}
```

Nous créons tout simplement une méthode **mCustomErrorDecoder** qui renvoie notre décodeur et le tour est joué.

Attention, il est tout à fait possible de déclarer notre décodeur de manière plus simple en ajoutant par exemple `@Component` à celui-ci. Néanmoins, nous allons préférer garder les déclarations des beans séparément, dans un package `configuration`, pour une meilleure lisibilité du code.

Lancez ClientUI et Microservice-produits. Vous obtenez alors cette réponse :



Vous avez bien votre code d'erreur 400 renvoyé par votre propre exception, au lieu du code 500 générique.

Vous avez également le message d'erreur que vous avez indiqué.

Maintenant que le mécanisme de propagation d'erreur est en place, vous pouvez **générer des exceptions facilement pour tous les cas et codes d'erreurs,** par exemple :

```
else if(reponse.status() > 400 && reponse.status() <=499 ) {
            return new Product4XXException(
                    "Erreur de au format 4XX "
            );
        }
```

Ce code va vous permettre de lancer une exception pour tous les cas où le code HTTP est entre 401 et 499.

Enfin, vous pouvez même récupérer le message renvoyé dans le corps de la réponse par *Microservice-produit* en y accédant via `reponse.body()`.

Le cas 404 Not Found

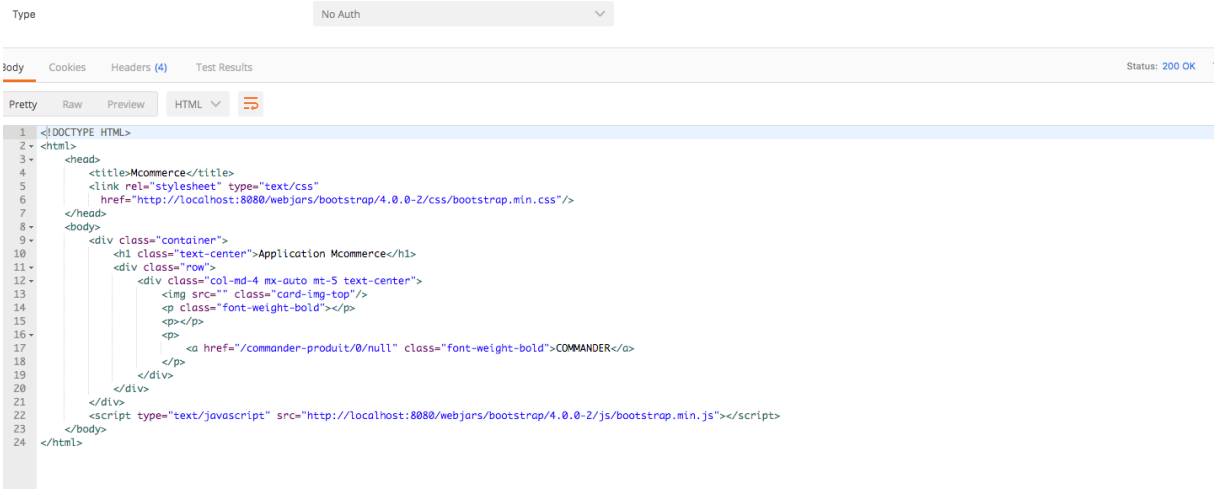
Au début, nous avons changé `ProductNotFoundException` afin qu'elle nous renvoie *400* au lieu de *404.* L'objectif était de lever une ambiguïté sur cette erreur.

En effet, Feign propose un **argument decode404** qui s'utilise comme ceci :

```
@FeignClient(name = "microservice-produits", url = "localhost:9001", decode404 = true)
```

decode404 = true est souvent utilisé à tort afin de gérer automatiquement les cas de ressources non trouvées.

Si vous lancez ClientUI avec `decode404` à true, et que vous appelez de nouveau http://localhost:8080/details-produit/20, vous obtenez ceci :



Vous avez un code *200 OK* et du HTML sans contenu.

En effet, cet argument `decode404` permet simplement de passer l'erreur et donc d'éviter de lancer la fameuse `FeignException`. Le but dépasse le cadre de ce Lab, mais si vous êtes curieux, c'est simplement pour éviter le déclenchement des circuits breakers comme Hystrix.

Vous devez donc toujours traiter l'erreur 404 exactement comme nous l'avons fait avec l'erreur 400, grâce à une condition dans votre décodeur.

Voici donc le code pour gérer l'erreur 404 :

```
package com.clientui.exceptions;

import feign.Response;
import feign.codec.ErrorDecoder;


public class CustomErrorDecoder implements ErrorDecoder {

    private final ErrorDecoder defaultErrorDecoder = new Default();

    @Override
    public Exception decode(String invoqueur, Response reponse) {

        if(reponse.status() == 400 ) {
            return new ProductBadRequestException(
                    "Requête incorrecte "
            );
        }

        else if (reponse.status() == 404 ) {
            return new ProductNotFoundException(
                    "Produit non trouvé "
            );
        }

        return defaultErrorDecoder.decode(invoqueur, reponse);
    }

}
```

Remettez `HttpStatus.NOT_FOUND` dans `ProductNotFoundException` de *Microservice-produit* et testez !

La branche pour récupérer le code de ce chapitre est : `ClientEtMSCommuniquant`.

En résumé

- Feign est un outil qui permet la simplification de la communication entre Microservices, en générant automatiquement les requêtes HTTP à partir des données fournies dans des classes appelées proxies.
- Une fois la réponse du Microservice distant reçue, Feign l'associe à un bean local que l'on aura créé. On obtient alors en retour directement des objets java prêts à l'emploi.
- Quand Feign rencontre un autre code de réponse que *2XX*, il génère une exception `FeignException`. Afin de pouvoir décoder la réponse et extraire les bons codes d'erreur, il faut créer une classe qui hérite de `ErrorDecoder` et qui implémente `decode`.