Atelier 1
Créer les Microservices e-commerce et leur client

1. Mettez en place l'application "Mcommerce"

Afin de vous aider à mieux comprendre les Edge Microservices, nous allons commencer par créer une petite application e-commerce **ultra minimaliste** que nous allons appeler "Mcommerce".

Cette application est composée de 3 Microservices :

1. **Microservice-produits :** ce Microservice gère le produit. Il propose 2 opérations simples : lister tous les produits et récupérer un produit par son id.
2. **Microservice-commandes :** Microservice de gestion des commandes. Il permet de passer une commande et de récupérer une commande par son id.
3. **Microservice-paiements :** ce Microservice permet de simuler le paiement d'une commande. Une fois le paiement enregistré, il fait appel au *Microservice-commandes* pour mettre à jour le statut de la commande.

L'application se trouve dans ce dépôt Github**.**

Pour chaque section du Lab, vous aurez une branche dans le dépôt qu'on vous indiquera.

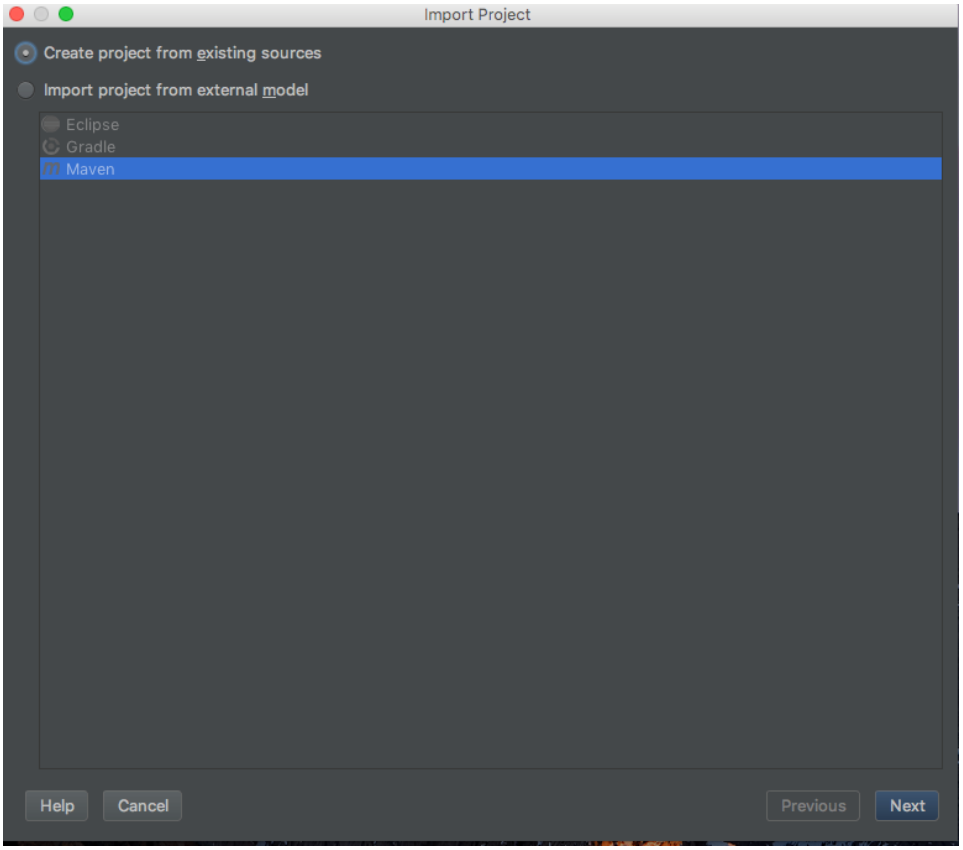
Nous allons donc commencer par mettre en place cette application dans IntelliJ IDEA.

Commencez par cloner le projet vers un dossier de votre choix.

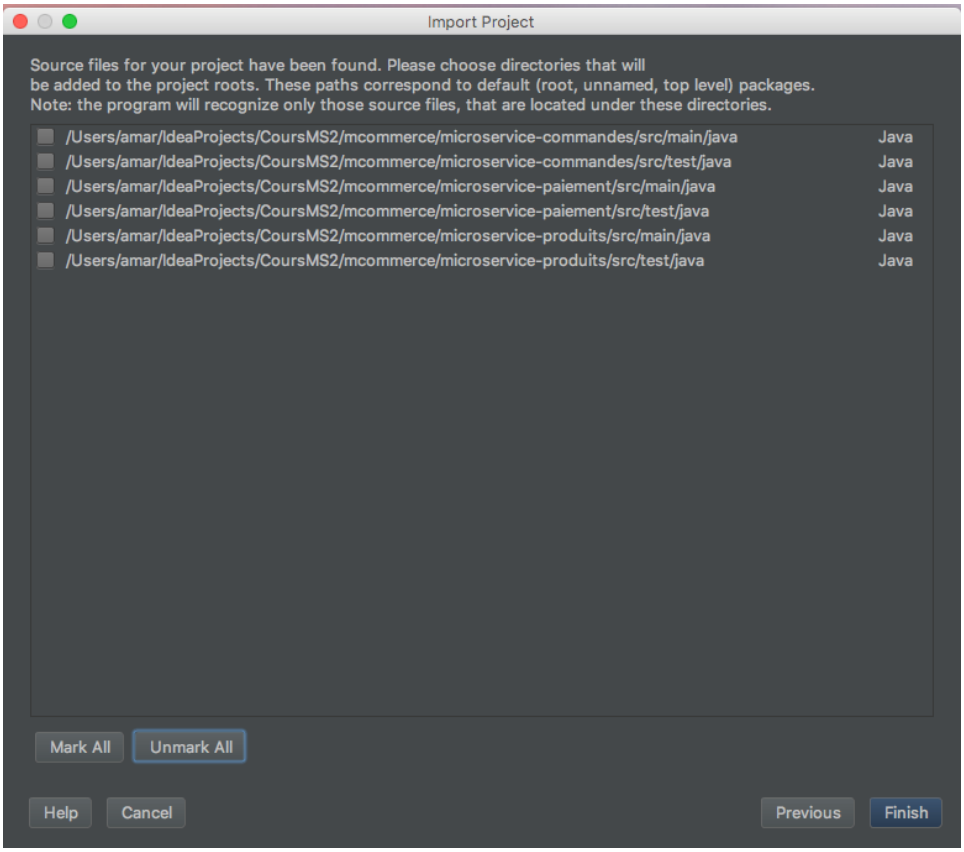
```
git clone https://github.com/AmarMicroDev/mcommerce.git
```

Une fois le projet cloné, cliquez sur "Import Project" puis sélectionnez le dossier "mcommerce" :

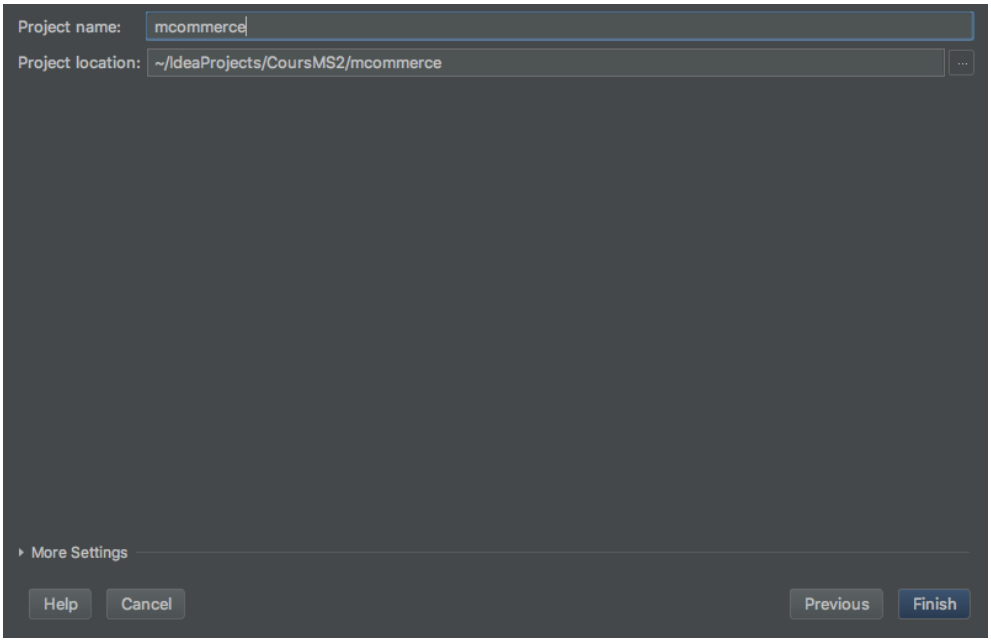
Sélectionnez "Create project from existing sources":



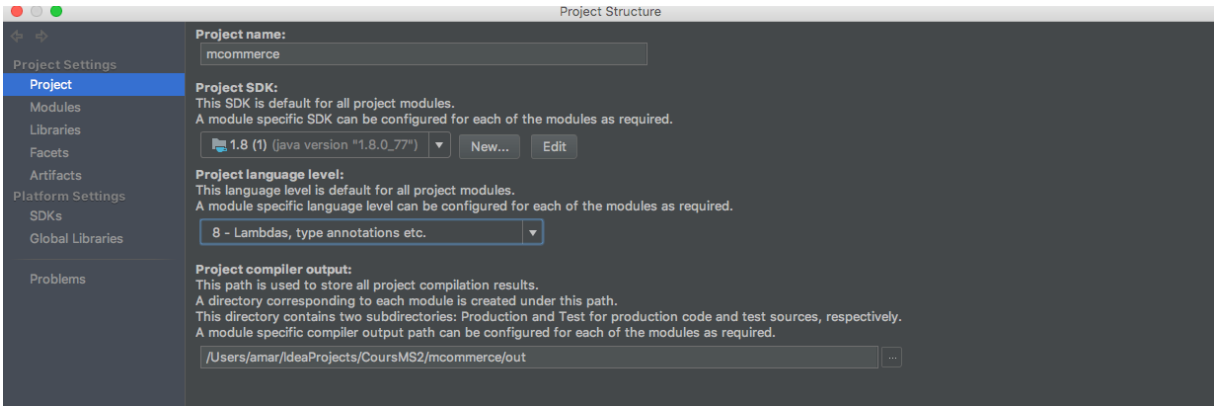
Cliquez sur "Unmark All" :



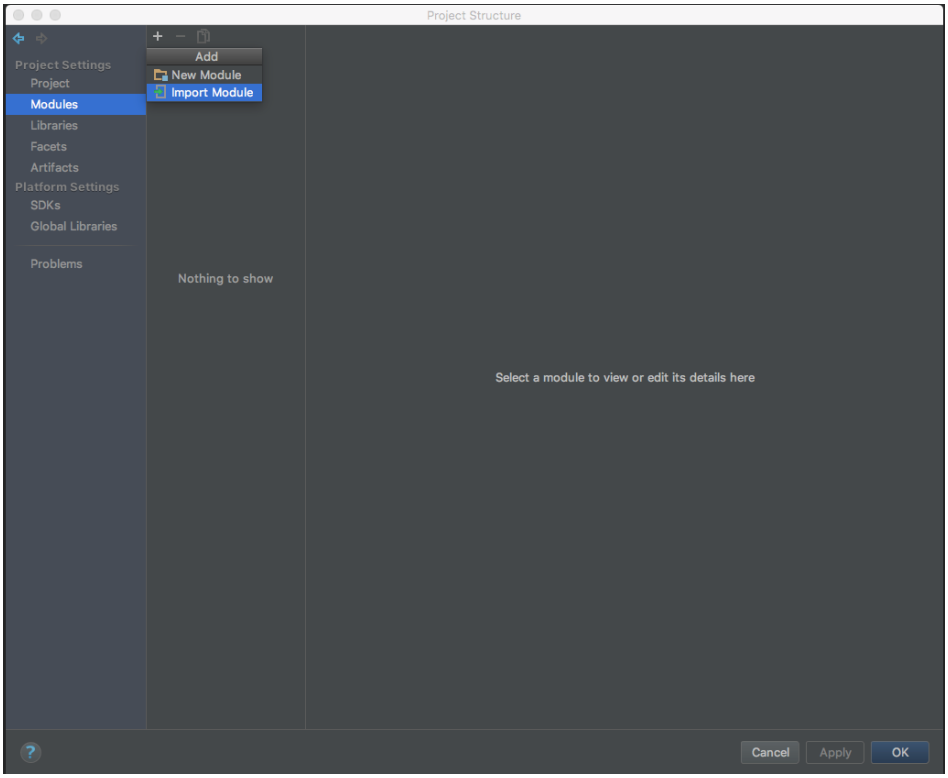
Nommez le projet "mcommerce" et terminez :



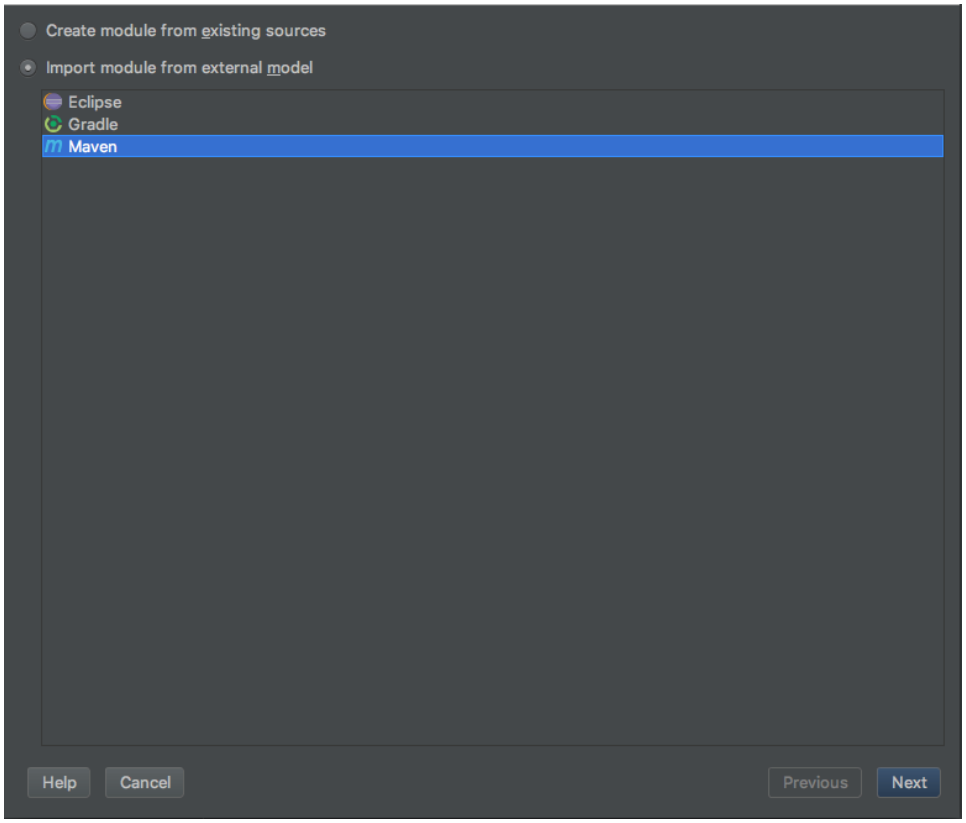
Allez dans File -> Project Structure, spécifiez le JDK 1.8 :



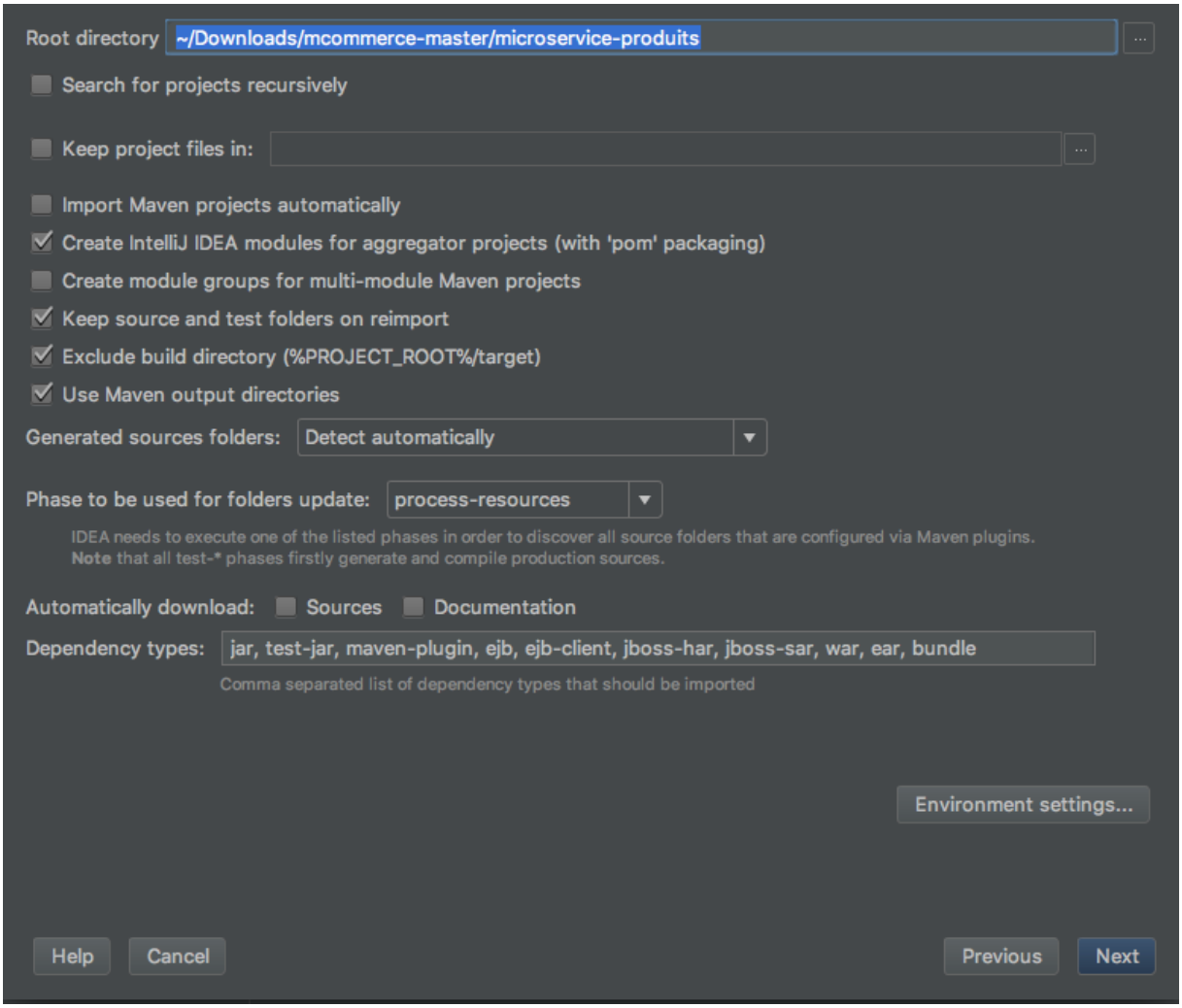
Sous "Modules", à gauche, sélectionnez le symbole "+" afin d'indiquer à IntelliJ où se trouvent les modules. Chaque module correspond à un Microservice :



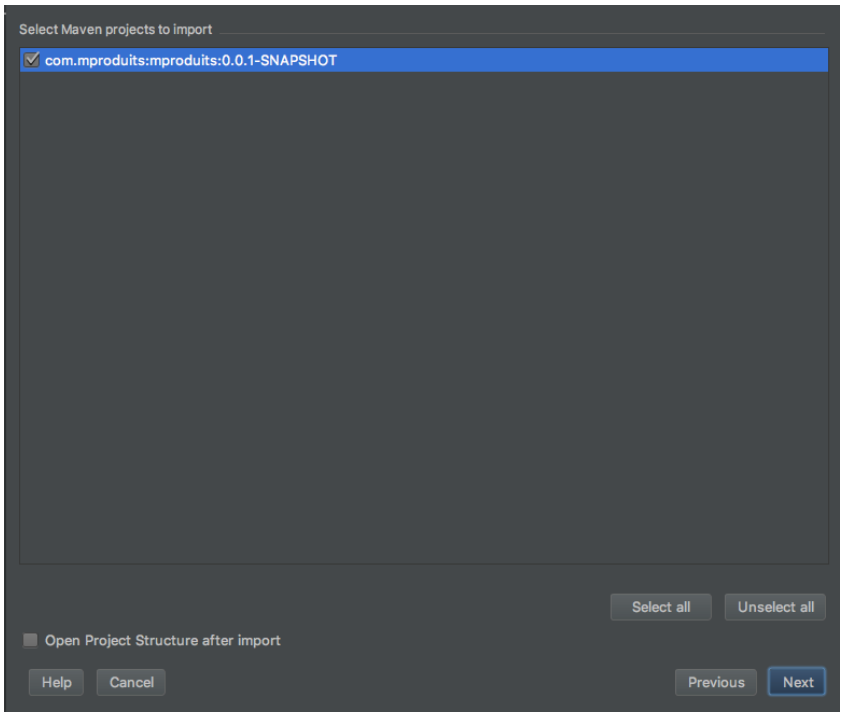
Sélectionnez "Maven" :



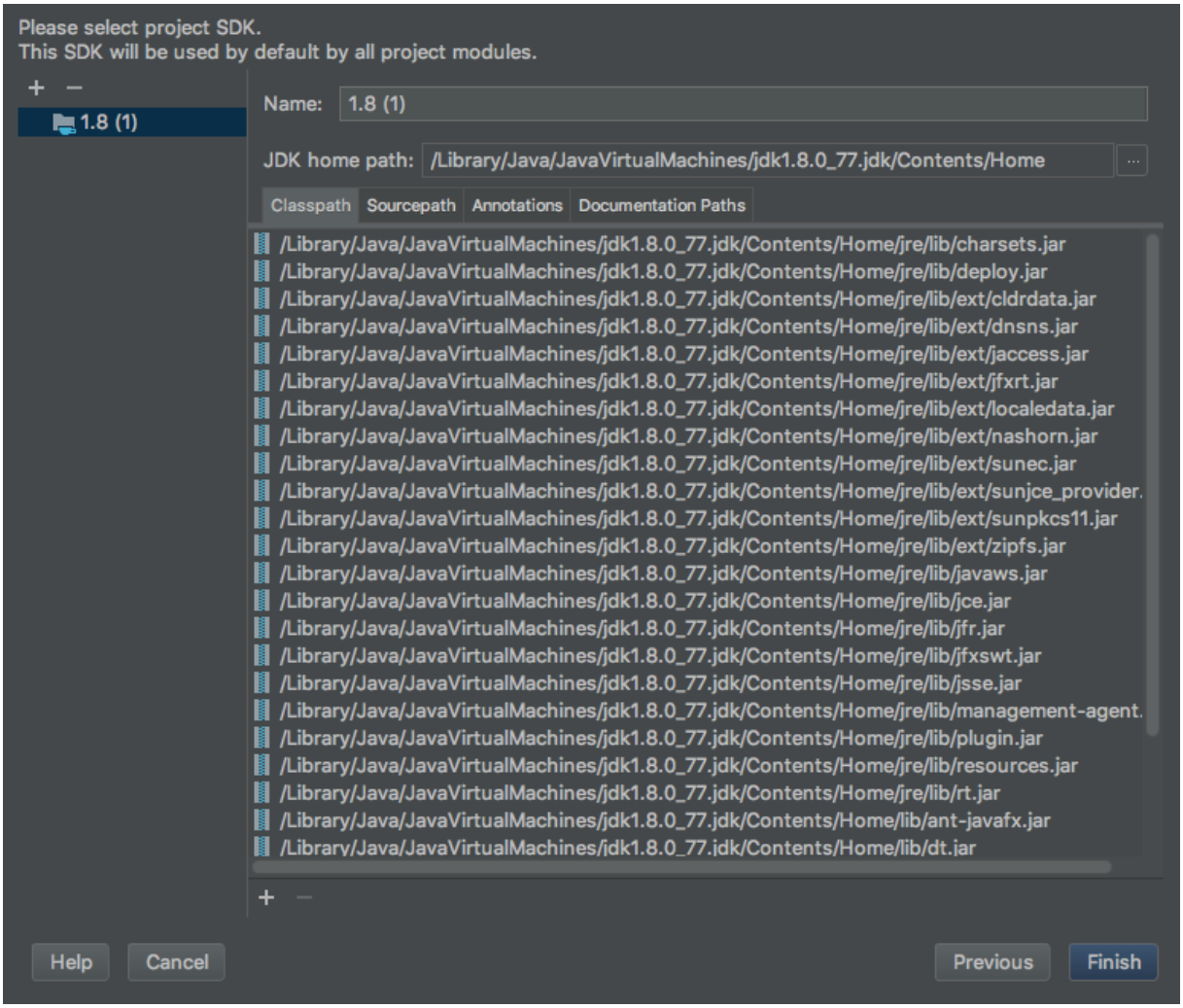
Vérifiez que vos options correspondent à la capture. Ce sont les options par défaut :



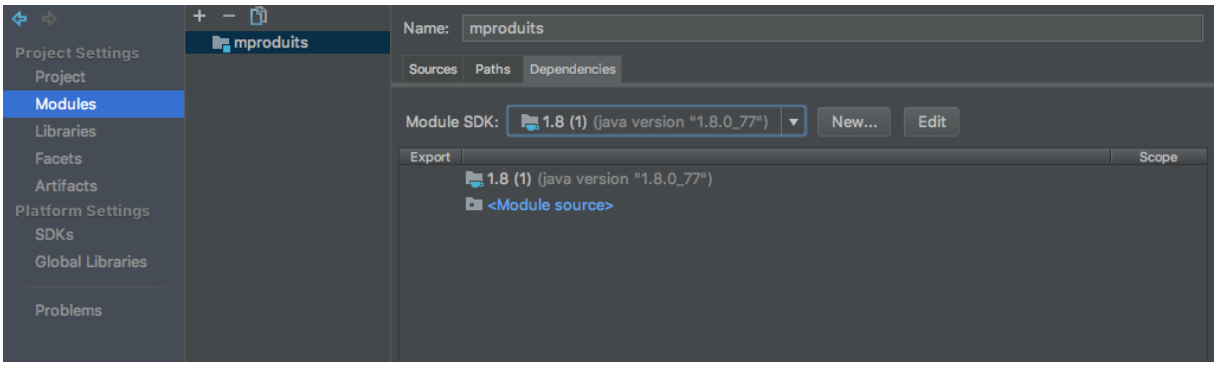
Laissez par défaut :



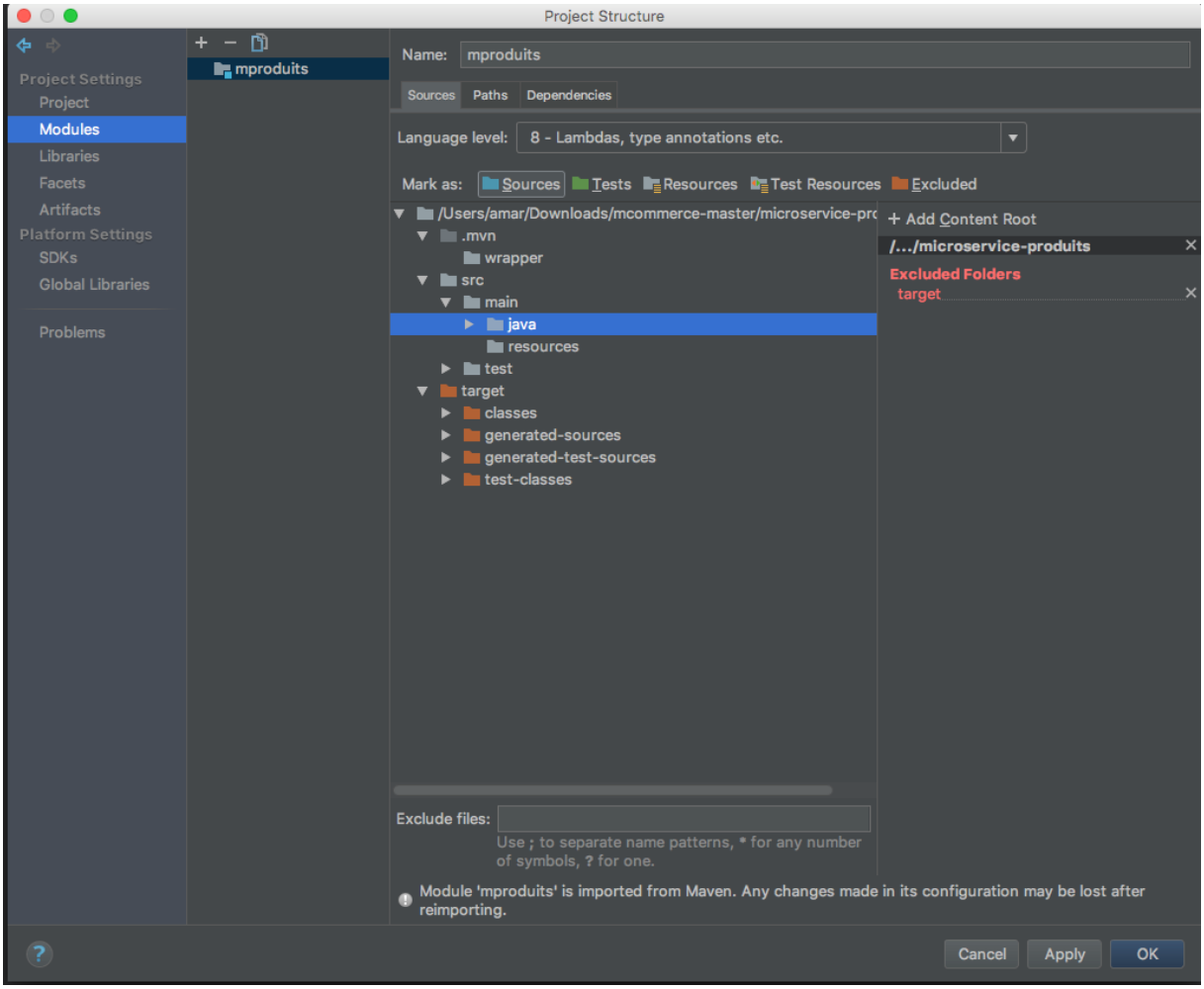
Sélectionnez le JDK 1.8 et terminez :



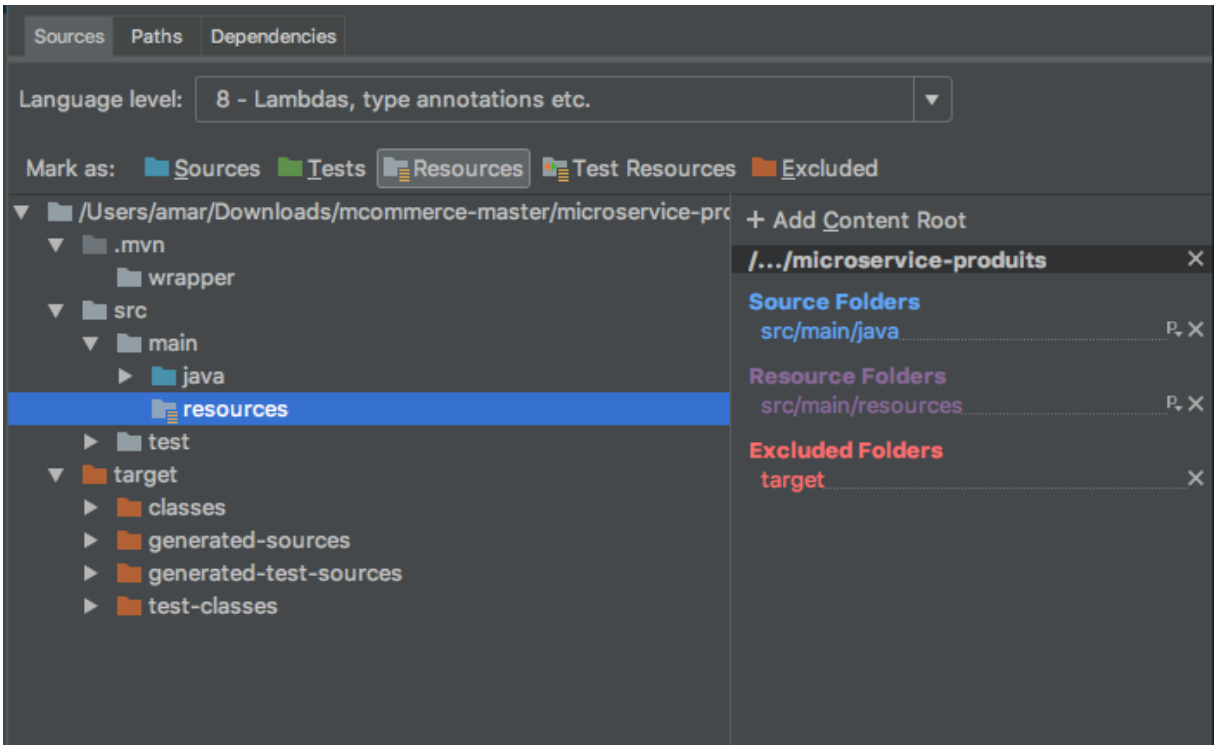
Votre Microservice est importé et apparaît sous le nom "mproduits", à gauche. Sélectionnez l'onglet "Dependencies", puis sélectionnez le JDK 1.8 :



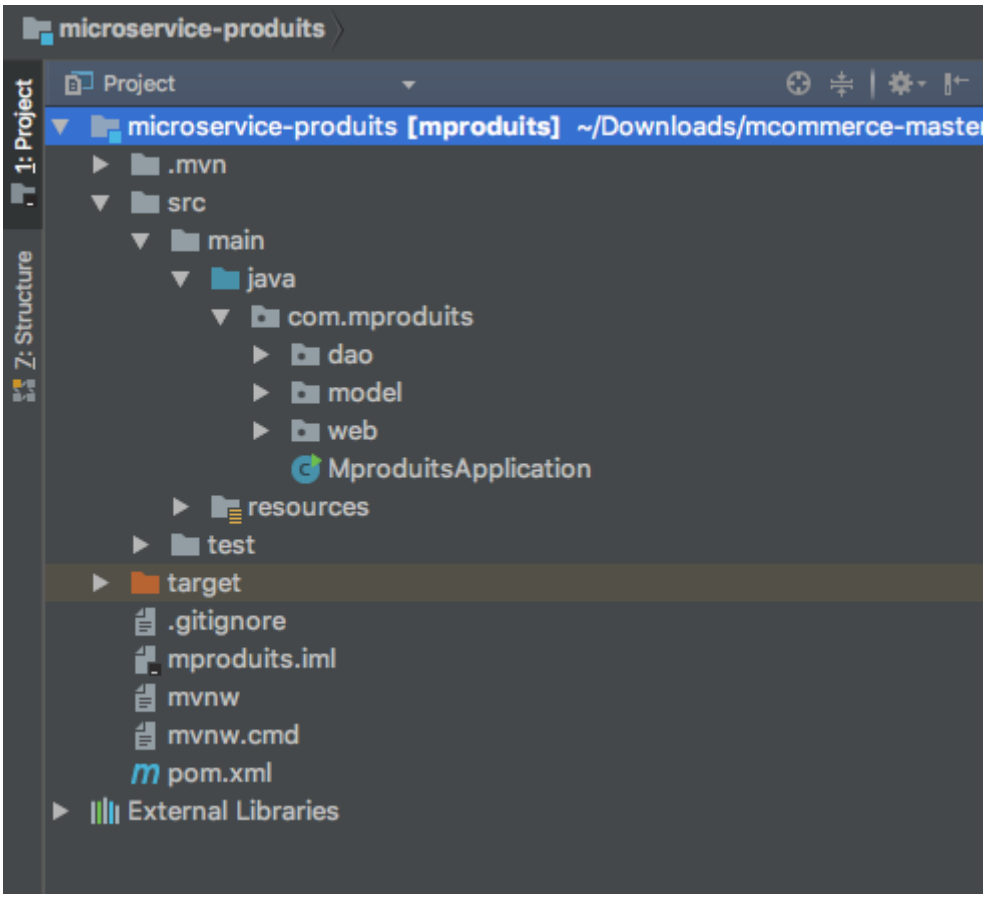
Déroulez l'arborescence du Microservice, sélectionnez le dossier "Java", puis marquez-le comme contenant le code source grâce à l'icône "Source", en haut :



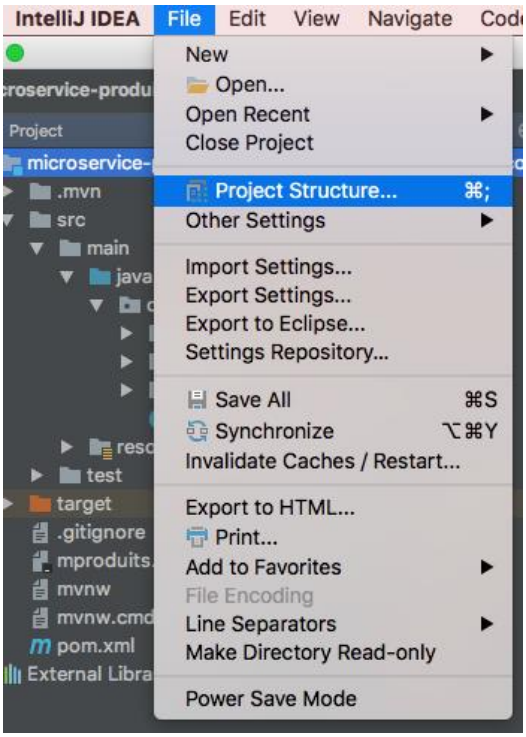
Marquez le dossier "resources" grâce à l'icône du même nom :



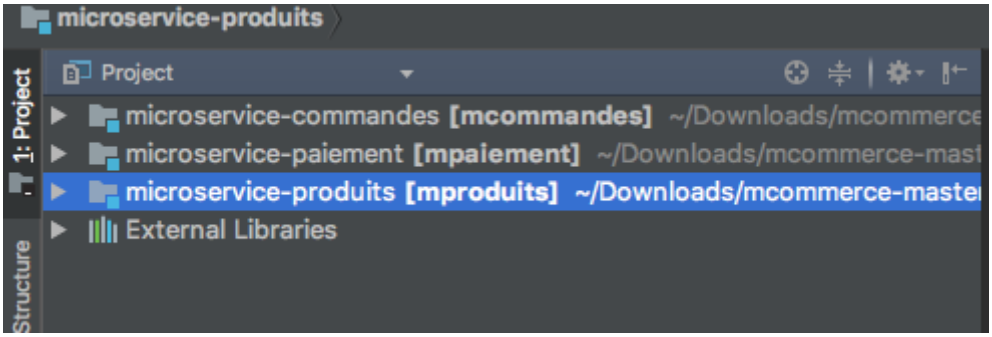
Validez et terminez. Vous devriez avoir votre premier Microservice à gauche, comme ceci :



Cliquez sur File -> Projet Structure, et recommencez la même opération pour les 2 autres Microservices :



Vous devriez obtenir cette arborescence à la fin :

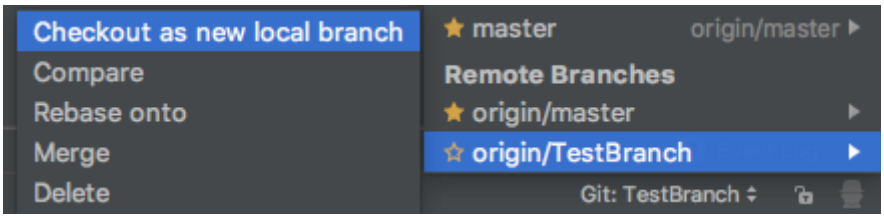


Cette forme de projet, assez spéciale dans IntelliJ, fait que le projet est constitué de plusieurs **"modules".** Ces modules sont des Microservices dans notre cas.

L'avantage est que **chaque module est un projet** en soi. Il a ses propres dépendances, ses propres fichiers de configuration et surtout, vous allez pouvoir les exécuter tous simultanément, comme si vous aviez ouvert trois projets IntelliJ différents.

Vous pourrez ainsi avoir votre projet localisé à un seul endroit tout en gardant vos Microservices parfaitement isolés. Vous pourrez également les lancer dans la même fenêtre et les faire communiquer sans jongler entre plusieurs projets IntelliJ différents.

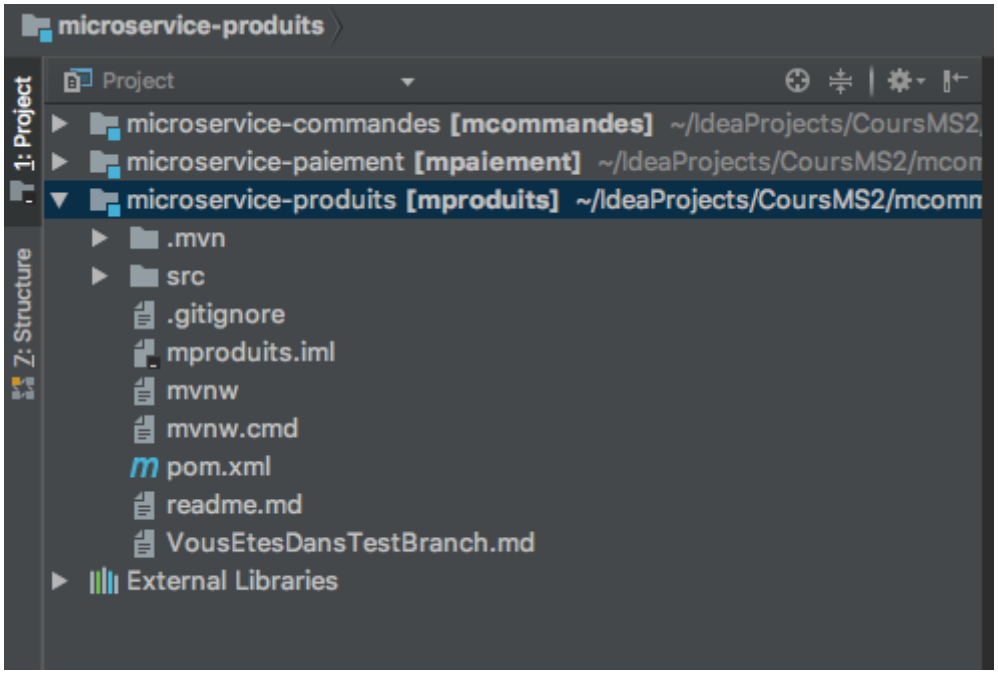
Changez de branche dans le projet vers "TestBranch" spécialement créé pour que vous puissiez vous familiariser avec le passage de branche en branche dans IntelliJ. Tout en bas à droite, sélectionnez origin/TestBranc ->Checkout as new local branch :



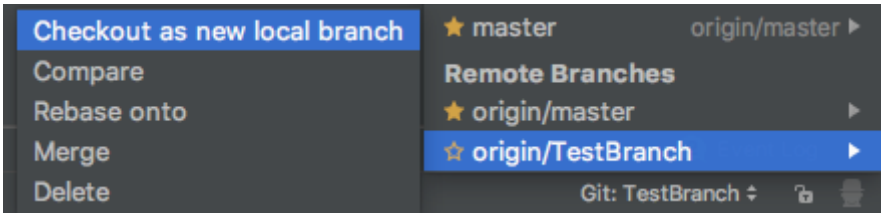
Nommez ensuite du même nom votre copie de cette branche "TestBranch" dans la boîte de dialogue qui s'ouvre.

Vous êtes désormais sur la branche "TestBranch". Vérifiez que c'est bien le cas, en bas à droite :

Vous devriez également voir apparaître, sous le Microservice-Produits, un fichier nommé "VousEtesDansTestBranch.md" :



Maintenant que vous savez changer de branche, revenez à la branche principale Master : sélectionnez "master" puis "checkout". Fermez la boîte d'avertissement qui s'affiche.



2. Explorer l'application

Nous allons analyser les différents Microservices afin que vous soyez à l'aise pour les manipuler durant le Lab.

Je vous conseille vivement de créer un autre projet à part dans lequel vous clonez le dépôt et sur lequel vous allez écrire le code qu'on va vous donner durant le Lab. Le premier projet que vous avez créé plus haut doit servir surtout à voir et à analyser la version du Lab pour pouvoir reproduire les concepts.
Tous ces Microservices ont en commun :

- L'utilisation de **Spring Data JPA** pour la communication avec la base de données
- Une base de données en mémoire de type **H2**
- Les mêmes paramètres dans le fichier **application.properties** :

```
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
```

Seul *server.port* change. Chaque Microservice a son propre port.

Microservice-produits

Ce Microservice permet une gestion très basique des produits.

```
@RestController
public class ProductController {
    @Autowired
    ProductDao productDao;

    // Affiche la liste de tous les produits disponibles
    @GetMapping(value = "/Produits")
    public List<Product> listeDesProduits(){

        List<Product> products = productDao.findAll();

        if(products.isEmpty()) throw new ProductNotFoundException("Aucun produit n'est
disponible à la vente");

        return products;

    }

    //Récuperer un produit par son id
    @GetMapping( value = "/Produits/{id}")
    public Optional<Product> recupererUnProduit(@PathVariable int id) {

        Optional<Product> product = productDao.findById(id);

        if(!product.isPresent())  throw new ProductNotFoundException("Le produit
correspondant à l'id " + id + " n'existe pas");

        return product;
    }
}
```

Microservice-produits écoute le port 9001. Il est constitué des éléments suivants :

- La méthode `listeDesProduits` : qui permet la récupération de la liste de tous les produits.
- La méthode `recupererUnProduit` : qui permet de récupérer un produit par son id.
- Une exception `ProductNotFoundException` : qui renvoie le code 404 si le ou les produits ne sont pas trouvés.
- Le fichier *data.sql* : qui permet de préremplir automatiquement la base de données avec plusieurs produits afin de pouvoir faire des tests.

Microservice-commandes

Ce Microservice permet de passer des commandes et d'en récupérer :

```
@RestController
public class CommandeController {

    @Autowired
    CommandesDao commandesDao;
```

```
    @PostMapping (value = "/commandes")
    public ResponseEntity<Commande> ajouterCommande(@RequestBody Commande commande){

        Commande nouvelleCommande = commandesDao.save(commande);

        if(nouvelleCommande == null) throw new
ImpossibleAjouterCommandeException("Impossible d'ajouter cette commande");

        return new ResponseEntity<Commande>(commande, HttpStatus.CREATED);
    }

    @GetMapping(value = "/commandes/{id}")
    public Optional<Commande> recupererUneCommande(@PathVariable int id){

        Optional<Commande> commande = commandesDao.findById(id);

        if(!commande.isPresent()) throw new CommandeNotFoundException("Cette commande
n'existe pas");

        return commande;
    }
}
```

Microservice-commandes écoute le port 9002. Il est constitué des éléments suivants :

- La méthode `ajouterCommande` : qui permet l'ajout d'une commande via un *POST.*
- La méthode `recupererUneCommande` : qui récupère une commande via son *id.*
- `Optional` : permet de ne plus vérifier si l'objet est null à chaque fois et évite les NullPointerExceptions.
- `isPresent` : vérifie que l'objet commande existe et n'est pas vide.
- L'exception : `ImpossibleAjouterCommandeException` , qui est déclenchée en dernier recours quand on n'arrive pas à enregistrer la commande pour cause d'erreur interne. Le code renvoyé est alors 500.
- L'exception `CommandeNotFoundException` : qui renvoie le code 404 lorsqu'une commande n'est pas trouvée.

Microservice-paiement

Ce Microservice reçoit l'id d'une commande, le montant et le numéro de la carte bancaire, puis enregistre le paiement dans la base de données :

```
@RestController
public class PaiementController {

    @Autowired
    PaiementDao paiementDao;

    @PostMapping(value = "/paiement")
    public ResponseEntity<Paiement>  payerUneCommande(@RequestBody Paiement paiement){


        //Vérifions s'il y a déjà un paiement enregistré pour cette commande
        Paiement paiementExistant =
paiementDao.findByidCommande(paiement.getIdCommande());
        if(paiementExistant != null) throw new PaiementExistantException("Cette commande
est déjà payée");
```

```
        //Enregistrer le paiement
        Paiement nouveauPaiement = paiementDao.save(paiement);


        if(nouveauPaiement == null) throw new PaiementImpossibleException("Erreur,
impossible d'établir le paiement, réessayez plus tard");



        //TODO Nous allons appeler le Microservice Commandes ici pour lui signifier que le
paiement est accepté

        return new ResponseEntity<Paiement>(nouveauPaiement, HttpStatus.CREATED);

    }
  }
```

Il dispose d'une seule opération : `payerUneCommande`. Une commande ne peut être payée qu'une seule fois, on vérifie alors si le paiement à effectuer n'a pas déjà eu lieu. Pour cela, nous devons chercher s'il y a un paiement correspondant à l'id de commande *idCommande*. On crée donc une méthode dans *PaiementDao* :

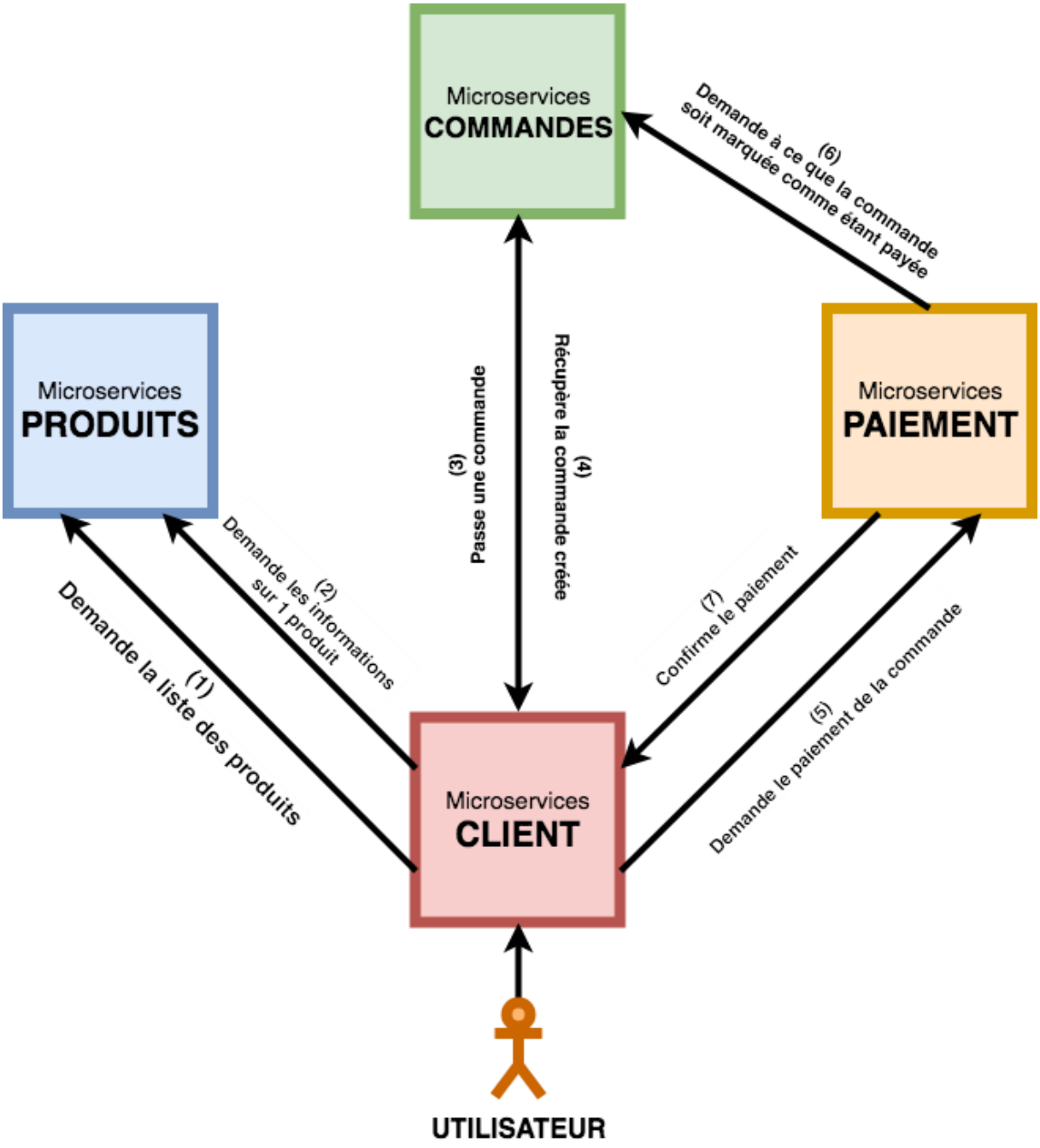
```
Paiement findByidCommande(int idCommande);
```

Si un paiement précédent est trouvé, on déclenche l'exception `PaiementExistantException`. Cette exception renvoie un code particulier : *409 CONFLICT* qui indique que les données reçues rentrent en conflit avec des données existantes.
En cas d'impossibilité d'enregistrer le paiement, le code *500* est renvoyé.

En cas de succès, le code *201 Created* est renvoyé, avec le contenu du paiement enregistré.

3. Ce que nous voulons faire avec ces Microservices

Voici un diagramme de l'application finale que nous souhaitons développer dans cette première partie du cours :



- (1) Le client propose plusieurs produits à l'utilisateur.
- (2) L'utilisateur en choisit un et affiche ses détails (description, prix, etc.).
- (3) L'utilisateur veut acheter ce produit. Le client envoie alors une requête pour passer la commande.
- (4) Il reçoit en retour un récapitulatif de la commande passée avec son id.
- (5) L'utilisateur veut payer sa commande. Le client envoie alors une requête au *Microservice-paiement* avec l'id de la commande reçu précédemment et un numéro de carte fourni par l'utilisateur.
- (6) Le paiement est enregistré en BD et le Microservice de paiement fait appel à celui des commandes afin de changer l'état de la commande en "payée" (commandePayee en TRUE).

- (7) Le *Microservice-paiement* envoie une réponse *201 Created* pour confirmer le paiement.

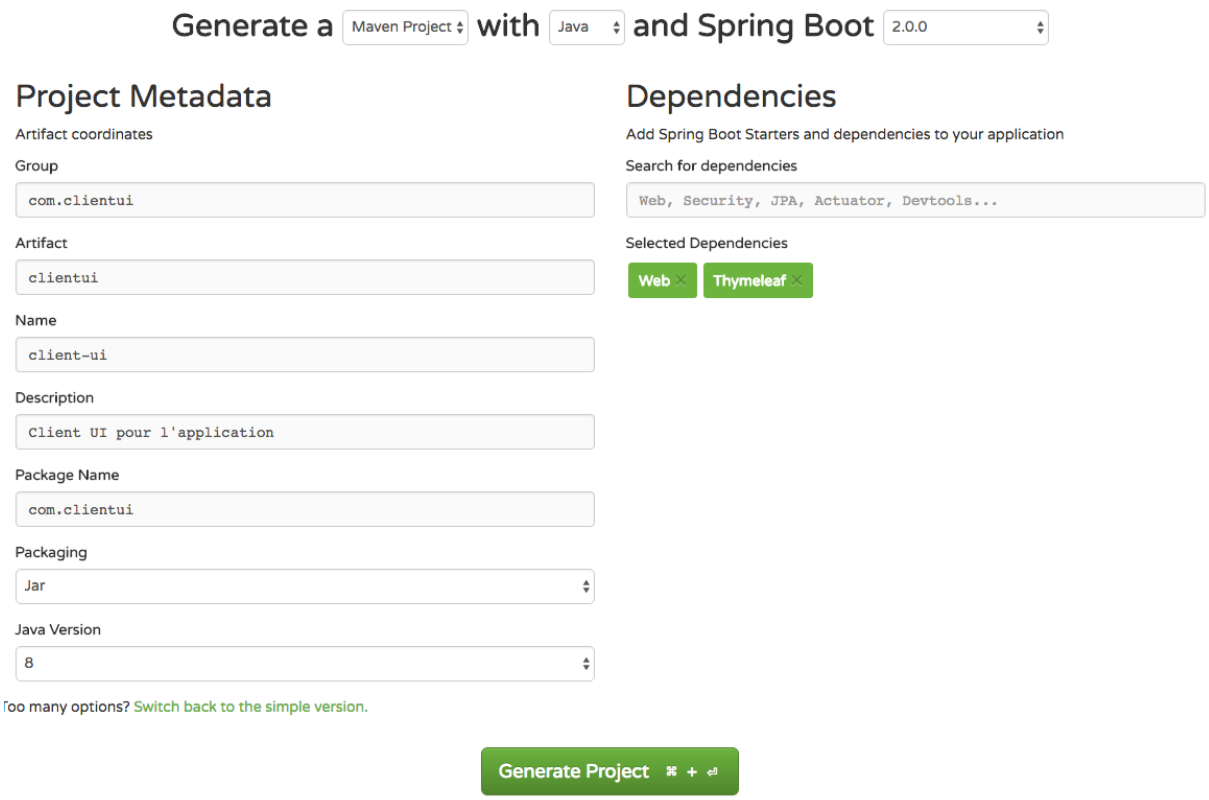
L'application que nous allons mettre en place ici est très basique, largement optimisable, mais elle a l'avantage de décortiquer, étape par étape, son fonctionnement. Nous allons faire l'impasse sur la gestion des erreurs, les validations et plein d'autres concepts.

4. Créer un client

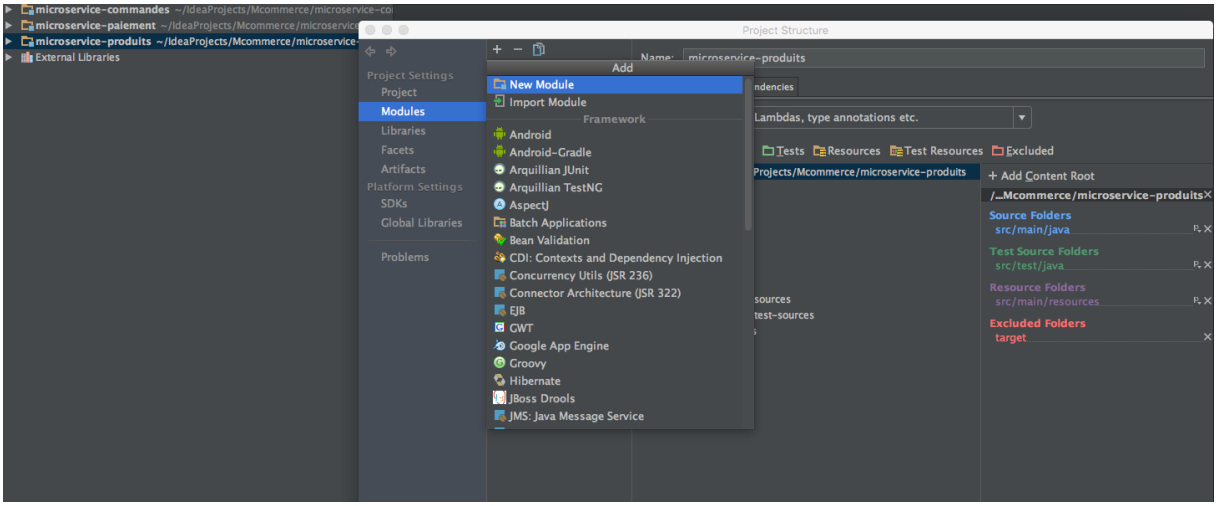
Nous allons maintenant créer notre squelette du client qui ira consommer nos Microservices. Pour créer une interface graphique, nous allons utiliser Thymeleaf comme moteur de template.

Nous allons commencer par créer un client très minimaliste afin que vous vous familiarisiez avec l'utilisation de Thymeleaf (pour ceux qui ne connaissent pas), puis nous allons l'enrichir.

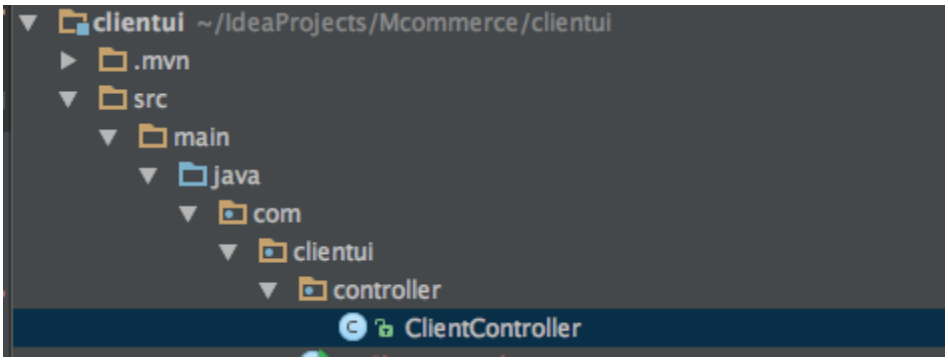
Commencez par créer un projet sur Spring Initializr, nommez-le *client-ui,* puis cochez "web" et "Thymeleaf", comme suit :



Rendez-vous dans IntelliJ, importez le projet comme nouveau module, puis sélectionnez les sources et les ressources :



Créez un nouveau contrôleur "ClientController" sous un package "controller" :



Voici donc notre contrôleur :

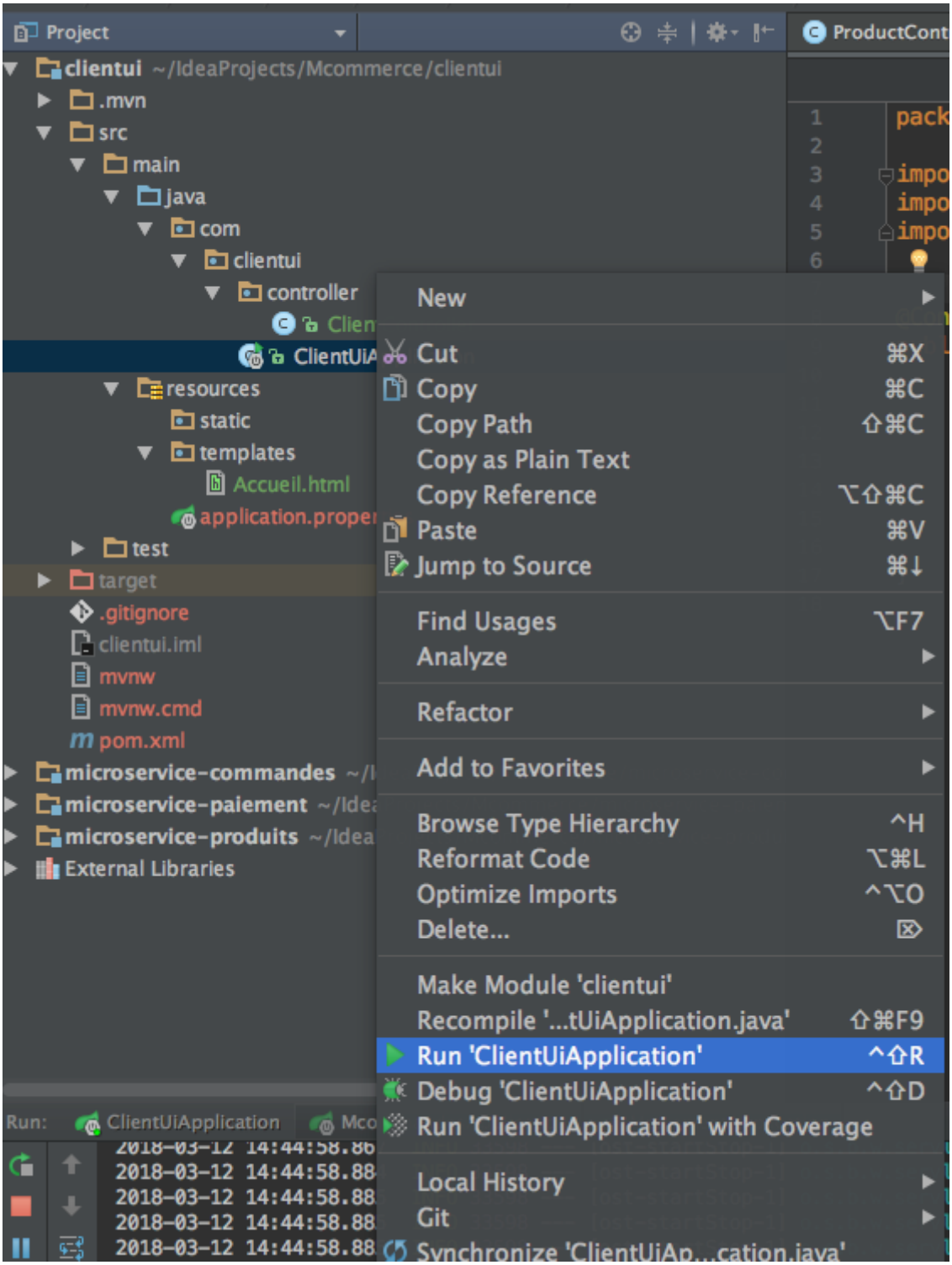
```java
package com.clientui.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;


@Controller
public class ClientController {

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

        return "Accueil";
    }

}
```

Explications :

- L'annotation @Controller est une annotation de Spring MVC qui dit au *DispatcherServlet,* qui reçoit toutes les requêtes pour le dispatcher, de chercher dans cette classe s'il y a une opération qui correspond à l'URI appelé.
- Nous créons ensuite une méthode qui répond aux URI de type "/", c'est-à-dire la page d'accueil de notre interface.

- model est une instance de la classe Model, que l'on passera en argument à notre méthode et qui nous permettra de renseigner des données à passer à la vue. Nous y reviendrons plus tard.
- return "Accueil" : Spring ira chercher dans le dossier "template", dans "resources", la page HTML du nom de *Accueil.html.*

Justement, rendez-vous dans *resources/templates* et créez un fichier *Accueil.html,* puis écrivez quelque chose à l'intérieur du fichier. Lancez ensuite votre client :



Rendez-vous à http://localhost:8080/ et vous verrez le contenu de votre HTML s'afficher.

Nous allons utiliser Bootstrap pour créer notre interface. Pour faciliter son intégration, nous allons ajouter une dépendance à notre *pom.xml :*

```
<dependency>
            <groupId>org.webjars</groupId>
            <artifactId>bootstrap</artifactId>
            <version>4.0.0-2</version>
</dependency>
```

Cette dépendance rendra les fichiers de Bootstrap disponibles pour notre HTML sans aucune configuration ou chemin à trouver.

Ajoutez ensuite ce contenu HTML à *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>


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

</body>
</html>
```

Relancez le client et vérifiez que tout fonctionne correctement.

La branche pour récupérer le projet avec ce squelette de client est **SqueletteClient**.