

Cours Spring-Core / Spring-Boot

- Live Coding -



Intervenant : Marwén Saidi (marwen.saidi@orange.com)

Décembre 2018 / Janvier 2019



Cours Spring-Core / Spring-Boot	1
Initialiser le projet	5
Comment est organisé le projet ?	5
Module ticket-model	7
Module ticket-consumer	8
Module ticket-business	8
Module ticket-webapp	8
Dessine-moi une inversion de contrôle (IoC)	9
Est-ce que c'est SOLID !	10
Utiliser le design pattern Factory	11
Procéder à une injection de dépendances	14
Améliorons la classe ManagerFactory	16
Ajouter de l'abstraction	18
L'inversion des dépendances c'est quoi ?	18
Les abstractions ne doivent pas dépendre des détails... ?	20
Mais en vrai, je fais quoi ?	20
Questions, problèmes, angoisses... ?	22
Conclusion	23
Une injection de dépendances plus sophistiquée	23
L'injection de dépendances avec une approche séquentielle	23
Utiliser un « IoC container »	24
Avec une approche séquentielle (sans IoC container)	26
Avec une approche déclarative (avec IoC container)	26
L'inversion de contrôle avec Spring	28
Créer un premier fichier de configuration	29
Initialiser Spring dans l'application web	30
Initialiser Spring dans les batches	30
Configurer l'IoC avec des annotations	31
Définir ses dépendances	32
Se déclarer en tant que bean	33
Initialiser l'IoC container	33
Ajout de contexte dans la configuration XML	33
Création d'une classe Java de configuration	34
Annotation vs XML	36
Mieux utiliser Spring	37
Organiser la configuration de Spring IOC	37
Compartimenter la configuration	37
Séparer le bootstrapping	37
Séparer la configuration de chaque module	38

Opter pour d'autres types d'injection de dépendances	40
L'injection par constructeur	40
L'injection par setter	42
L'injection par champs	43
Factoriser la configuration avec l'héritage de définition	44
Utiliser d'autres scopes pour les beans	45
Les logs	45
Utiliser Spring pour injecter d'autres éléments	47
Utiliser des valeurs de fichiers properties	47
Injecter une ressource JNDI	48
Conclusion	48
Implémenter des DAO	49
Création et configuration des DAO	49
Configuration des DAO	50
Configuration de la DataSource	50
Drivers JDBC	50
Module ticket-batch	51
Module ticket-webapp	51
Configuration de l'application web	52
Configuration des batches	54
Simplifier l'exécution de requêtes SQL avec Spring JDBC	57
Travail préparatoire	57
Ajout de la dépendance vers Spring JDBC	57
Requête SQL de sélection	58
Définir et exécuter une requête SQL de sélection	58
Récupérer le résultat d'une requête	60
Requête SQL de mise à jour	62
Passer des paramètres	62
Utiliser les attributs d'un JavaBean	62
Spécifier les types de données	63
Gérer les exceptions	65
Conclusion	65
Gérer les transactions avec Spring TX	67
Gérer le contexte transactionnel	67
La pilule bleue	68
Utiliser un TransactionTemplate	69
Définir le PlatformTransactionManager	69
Silence, moteur... action !	69
Demander vous-même l'annulation	70
Renvoyer un objet en sortie	70
Partager le TransactionTemplate	70

La pilule rouge	71
La propagation du contexte transactionnel	74
PROPAGATION_REQUIRED	75
PROPAGATION_REQUIRES_NEW	76
Ajuster les paramètres des transactions	76
Si vous utilisez le TransactionTemplate	76
Si vous utilisez le PlatformTransactionManager	77
Synchroniser les transactions hors de Spring JDBC	77
Conclusion	77
Récapitulatif général	79
Partie 1 : Mettre en œuvre une inversion de contrôle	79
Partie 2 : Faciliter le développement avec Spring	80
Spring Boot(Créez un Microservice grâce à Spring Boot)	81
Spring Initializr	81
Création et importation à partir de Spring Initializr	81
Analyse du code obtenu	90
MicrocommerceApplication.java	95
application.properties	96
MicrocommerceApplicationTests.java	96
Exécuter l'application	96
Créez l'API REST	101
Quels sont les besoins ?	101
Créez le contrôleur REST	102
Renvoyez une réponse JSON	106
Créez le DAO	110
Interagir avec les données	113

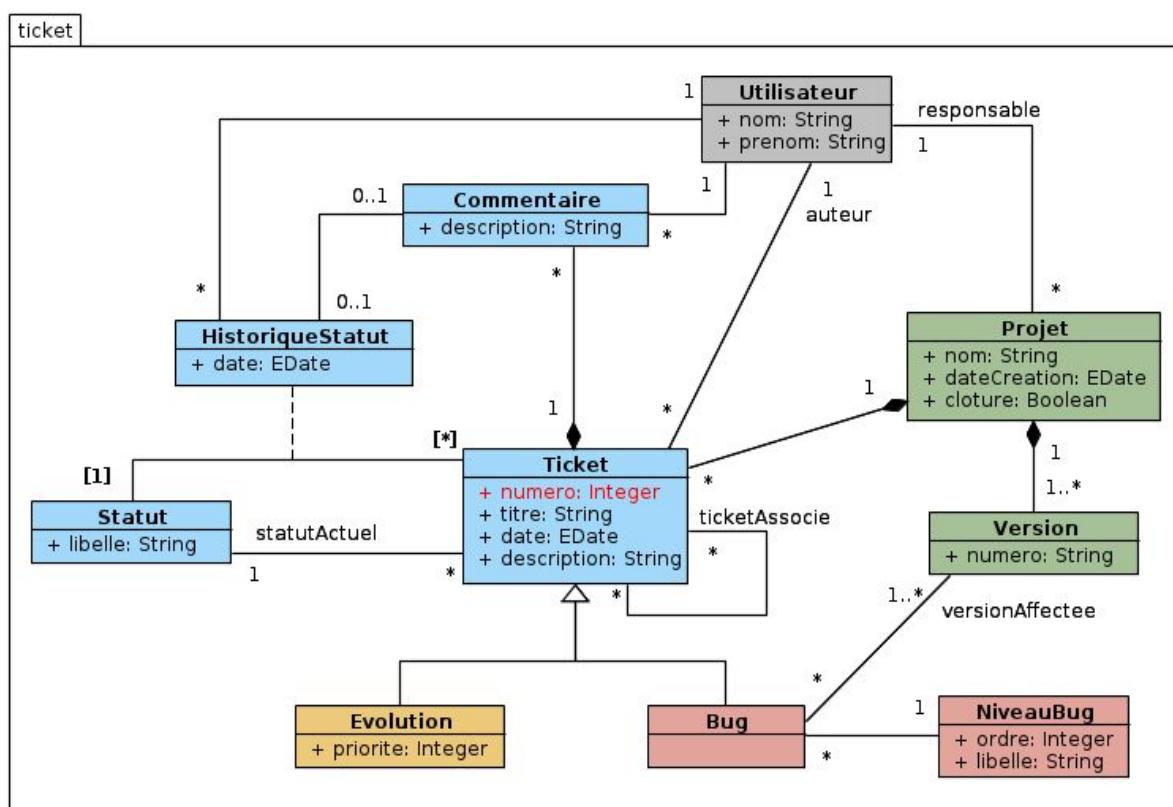
Initialiser le projet

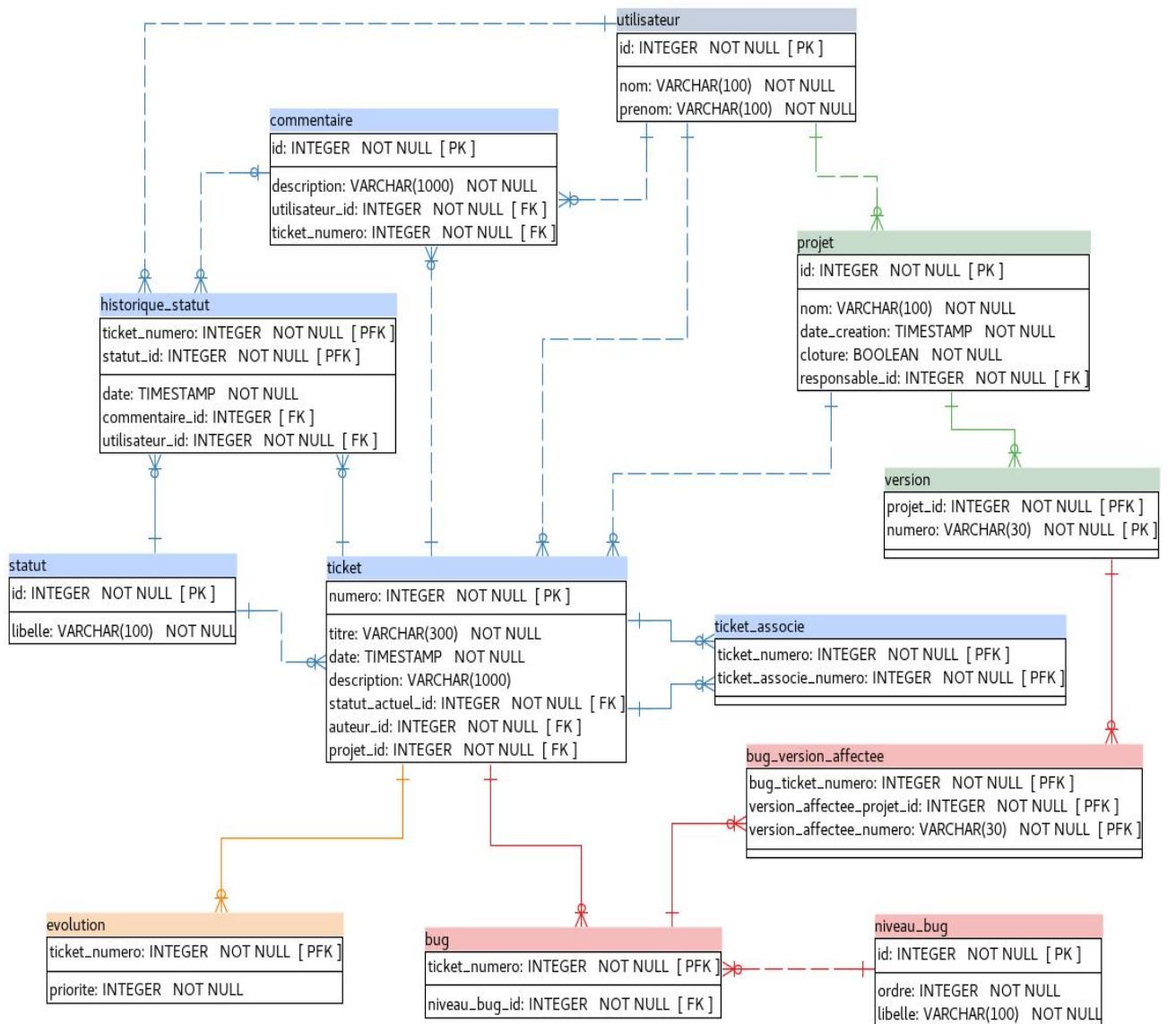
Afin de gagner du temps et de ne pas me répéter, je vous propose de partir du projet déjà initialisé disponible sur [GitHub](https://github.com/marwensaid/spring-core-training) (<https://github.com/marwensaid/spring-core-training>). Il vous suffit de le télécharger et de l'importer dans votre IDE.

Ce projet contient déjà certains éléments, comme vous pourrez le voir.

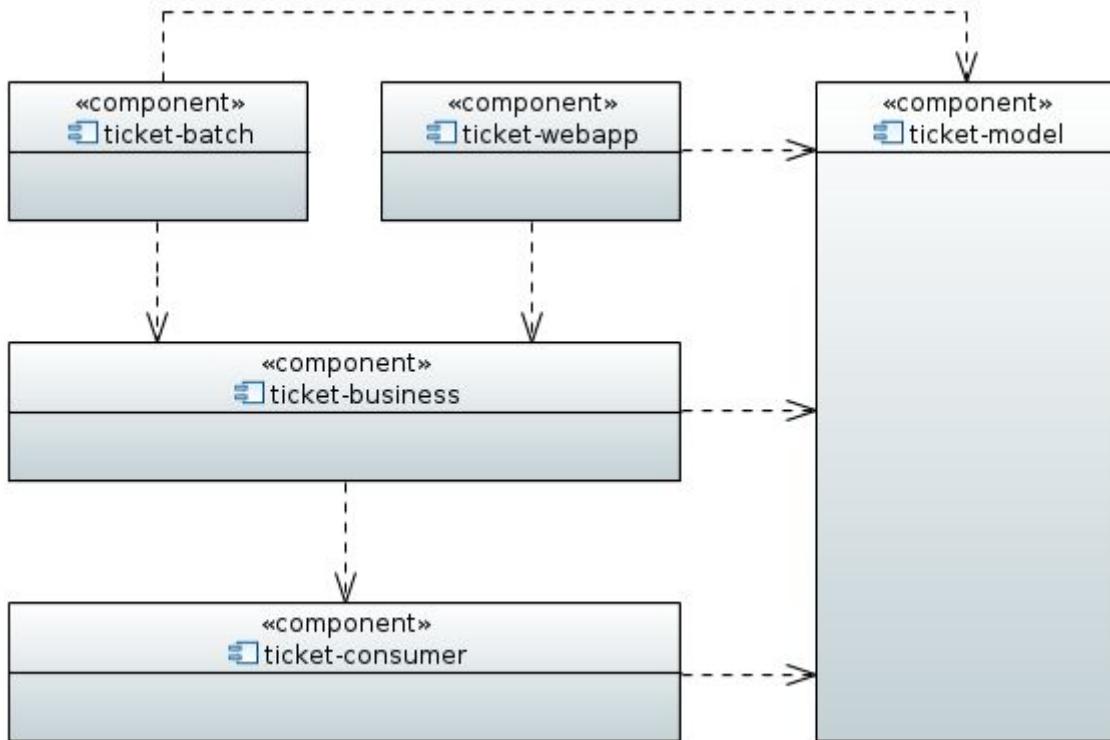
Comment est organisé le projet ?

Pour vous repérer, voici le diagramme de classes du domaine fonctionnel du projet ainsi que le modèle physique de données (MPD). Tous les éléments du domaine fonctionnel ne sont pas codés.





Le projet est un projet Maven multi-modules, organisé suivant une architecture multi-tiers où chaque couche fait l'objet d'un module Maven.



Voici le contenu du projet :

- Le module *ticket-webapp* contient l'application web. Pour plus de simplicité, l'application web ne fournit pas d'interface graphique mais uniquement des données, via une simple API proposant des webservices (backend). Ces données sont destinées à être consommées par une interface graphique web (frontend) qui ne sera pas développée dans ce cours.
- Le module *ticket-batch* contient les batches de l'application.
- Le module *ticket-business* contient la logique métier de l'application avec notamment les classes *Manager*.
- Le module *ticket-consumer* contient les éléments d'interaction avec les services extérieurs (base de données, webservices...). S'y trouvent notamment les DAO.
- Le module *ticket-model* contient les objets métiers (les *beans*).
- Le module *ticket-technical* contient les éléments et dépendances techniques de l'application (gestion des logs...). Tous les autres modules dépendent de ce module technique.

Module ticket-model

Contenu :

- package org.example.demo.ticket.model.bean : les objets du domaine fonctionnel (les *beans*).
 - les beans sont répartis dans des sous-packages (*projet*, *ticket*...)
- package org.example.demo.ticket.model.recherche : les objets de définition des recherches (critères de recherche, de chargement...)
 - même découpage en sous-package que pour les beans

- package org.example.demo.ticket.model.exception : les exceptions propres à l'application

Module ticket-consumer

Non implémenté dans la version fournie

Module ticket-business

Contenu :

- package org.example.demo.ticket.business.manager : *Business Managers*, c'est-à-dire les gestionnaires métier liés aux beans.
 - un manager par sous-packages de beans
 - nommé du nom du sous-package suffixé par Manager (ex: *ProjetManager*, *TicketManager*...)

Module ticket-webapp

Pour plus de simplicité, je ne développerai pas l'interface graphique web de l'application, mais juste une implémentation de quelques cas d'utilisation sous forme de webservices REST. Le but n'est pas de développer toute l'application, mais le strict nécessaire pour interagir avec elle.

Les webservices REST fournis par l'application web utilisent l'API JAX-RS et [Jersey](#) comme implémentation. Je ne rentrerai pas dans le détail de l'implémentation de webservices REST en Java, cela n'a pas d'importance pour ce cours. Ils nous permettront un peu d'interactivité à titre d'illustration.

L'implémentation des webservices REST se trouve dans le package org.example.demo.ticket.webapp.rest :

- sous-package resource : les *Ressources REST*
 - un sous-package par sous-packages de bean (*projet*, *ticket*...)
 - une classe de ressource par beans
 - nommée du nom du bean suffixé par Resource (ex: *ProjetResource*, *TicketResource*...)

Dessine-moi une inversion de contrôle (IoC)

Maintenant que l'environnement de développement est mis en place et que l'organisation du projet est posée, entrons dans le vif du sujet. Explorons un peu l'application pour voir comment elle est implémentée... Je vais commencer par le module *ticket-webapp*. J'y trouve tout d'abord la classe *RestApplication*.

```
package org.example.demo.ticket.webapp.rest;

import javax.ws.rs.ApplicationPath;
import org.glassfish.jersey.server.ResourceConfig;

@ApplicationPath("/")
public class RestApplication extends ResourceConfig {

    public RestApplication() {
        packages("org.example.demo.ticket.webapp.rest");
    }
}
```

Cette classe est spécifique au framework *Jersey* pour l'implémentation des webservices REST avec JAX-RS. Elle fournit la configuration et l'initialisation du framework. Je déclare ici quels sont les packages que *Jersey* doit scanner pour trouver l'implémentation des webservices REST. Je ne vais pas m'attarder dessus, ce n'est pas le sujet de ce cours. Passons à la suite, qui est beaucoup plus intéressante...

Est-ce que c'est SOLID !

Prenez la classe *ProjetResource* dans le module *ticket-webapp* :

```
package org.example.demo.ticket.webapp.rest.resource.projet;
// import ...

/**
 * Ressource REST pour les {@link Projet}.
 */
@Path("/projets")
@Produces(MediaType.APPLICATION_JSON)
public class ProjetResource {

    /**
     * Renvoie le {@link Projet} d'identifiant {@code pId}
     *
     * @param pId identifiant du {@link Projet}
     * @return Le {@link Projet}
     * @throws NotFoundException Si le {@link Projet} n'a pas été trouvé
     */
    @GET
    @Path("{id}")
    public Projet get(@PathParam("id") Integer pId) throws NotFoundException {
        ProjetManager vProjetManager = new ProjetManager();
        Projet vProjet = vProjetManager.getProjet(pId);
        return vProjet;
    }

    //...
}
```

Vous voyez que celle-ci fait appel à la couche *business* et en particulier à la classe *ProjetManager*. Jusque-là, c'est assez logique.

Ce qui pose un problème en revanche, c'est que la classe *ProjetResource* instancie la classe *ProjetManager*. Il n'est pas très logique et propre que ce soit les classes de la couche *webapp* qui soient responsables du cycle de vie des classes de la couche *business* !

Il serait mieux de pouvoir appeler les méthodes d'une instance de *ProjetManager*, sans se soucier de son instanciation.

Pourquoi ce serait mieux ?

Car sinon, cela viole — principalement — le **S** et le **D**, du célèbre acronyme **SOLID**, donnant 5 principes de conception en programmation orientée objet afin d'obtenir une application plus fiable et plus robuste :

- Single responsibility principle (responsabilité unique)
- Open/closed principle (ouvert/fermé)
- Liskov substitution principle (substitution de Liskov)
- Interface segregation principle (ségrégation des interfaces)
- Dependency inversion principle (inversion des dépendances)



Ces principes sont décrits dans le livre de [Robert Cecil Martin](#) : *Agile Software Development. Principles, Patterns, and Practices* (2003, ISBN-13: 978-0135974445) :

- Le principe de **responsabilité unique**(*Single responsibility*) veut qu'une classe ne doit avoir qu'une seule responsabilité. Selon Robert C. Martin « *A class should have only one reason to change* ».
- Avec le principe d'**inversion des dépendances**(*Dependency inversion principle*), Robert C. Martin énonce que :
 - Les modules de haut niveau ne doivent pas dépendre des modules de plus bas niveau. Les deux doivent dépendre d'abstractions.
 - Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

Je ne vais pas — enfin, pas tout de suite — entrer dans le détail, mais en gros, afin de respecter ces principes, il serait préférable, ici, d'appeler des méthodes sur une instance de *ProjetManager*, sans avoir à l'instancier dans la classe *ProjetResource*.

Comment faire ? Eh bien, c'est ici qu'intervient le principe d'**inversion de contrôle** (*inversion of control* en anglais) communément abrégé en **IoC**.

Il y a plusieurs moyens de mettre en place cette inversion de contrôle dans notre cas :

- en utilisant le patron de conception (*design pattern*)**Factory**;
- en faisant de l'**injection de dépendances**...

Pour ceux qui manipulent régulièrement les différents design patterns, j'écarte volontairement l'utilisation du design pattern [Singleton](#) pour la classe *ProjetManager*. Celui-ci ne permet pas, à lui seul, de faire de l'inversion de contrôle.

Certes, l'instance et l'instanciation de la classe *ProjetManager* ne serait plus gérée par la classe *ProjetResource*, mais par la classe *ProjetManager* elle-même. Cependant, l'objectif du *Singleton* n'est pas de simplement éviter le new (instanciation) mais bien de limiter le nombre d'instances possibles d'une classe (généralement à 1) !

De plus, l'emploi du *Singleton* pour la classe *ProjetManager* ajouterait un couplage fort de la classe *ProjetResource* à l'implémentation du *ProjetManager*. Ce qui pose aussi problème : cela viole le *D* et le *L* de *SOLID*, mais j'y reviendrai plus tard.

Maintenant que le problème est identifié, voyons comment nous pouvons le résoudre grâce à l'**inversion de contrôle** (**IoC**).

Je vous propose d'y aller progressivement.

Utiliser le design pattern Factory

Une première solution envisageable est d'utiliser le design pattern *Factory*. Grâce à lui, la classe *ProjetResource* n'a plus à se soucier de l'instanciation de la classe *ProjetManager*, elle ne fait que demander à la *Factory* de lui donner une instance de *ProjetManager*.

Je vais créer une classe *ManagerFactory* à laquelle on pourra demander les instances des managers (*ProjetManager*, *TicketManager*...)



```
package org.example.demo.ticket.business;

public class ManagerFactory {

    public ProjetManager getProjetManager() {
        return new ProjetManager();
    }

    public TicketManager getTicketManager() {
        return new TicketManager();
    }
    //...
}
```

```
package org.example.demo.ticket.webapp.rest.resource.projet;
//...
public class ProjetResource {

    private ManagerFactory managerFactory;

    @GET
    @Path("{id}")
    public Projet get(@PathParam("id") Integer pId) throws NotFoundException {
        Projet vProjet = managerFactory.getProjetManager().getProjet(pId);
        return vProjet;
    }

    //...
}
```

OK, mais il reste encore une interrogation :

Comment obtenir l'instance de *ManagerFactory* dans la classe *ProjetResource* ?

On instancie la Factory ?

```
public class ProjetResource {
    private ManagerFactory managerFactory = new ManagerFactory();
    //...
}
```

⇒ Surtout pas !! On retomberait sur le même problème qu'au début !

On fait de la Factory un singleton ?

```

package org.example.demo.ticket.business;
//...
public final class ManagerFactory {
    /** Instance unique de la classe (design pattern Singleton) */
    private static final ManagerFactory INSTANCE = new ManagerFactory();

    /**
     * Constructeur.
     */
    private ManagerFactory() {
        super();
    }

    /**
     * Renvoie l'instance unique de la classe (design pattern Singleton).
     *
     * @return {@link ManagerFactory}
     */
    public static ManagerFactory getInstance() {
        return ManagerFactory.INSTANCE;
    }

    public ProjetManager getProjetManager() {
        return new ProjetManager();
    }

    public TicketManager getTicketManager() {
        return new TicketManager();
    }
    //...
}

```

Et dans la classe *ProjetResource* on obtient :

```

public class ProjetResource {
    private ManagerFactory managerFactory = ManagerFactory.getInstance();
    //...
}

```

→ C'est mieux, mais on ne peut plus faire d'abstraction sur la *Factory* (vous vous rappelez, le *D* de *SOLID*). C'est-à-dire que la classe *ProjetResource* dépend de l'implémentation de la *Factory*, pas seulement d'un contrat du genre "renvoie-moi un *ProjetManager*, renvoie-moi un *TicketManager*" (ceci vous paraîtra plus clair dans quelques minutes...).

Ce qui serait bien, ce serait qu'un élément extérieur à la classe *ProjetResource* lui donne, lui *injecte*, l'instance de *ManagerFactory*, via son constructeur ou un *setter* par exemple... Eh bien, c'est là qu'intervient **l'injection de dépendances** (abrégé **DI** pour *dependency injection* en anglais).

Procéder à une injection de dépendances

Afin que vous compreniez le principe de l'**injection de dépendances** et puissiez bien suivre le déroulement des opérations, je vais tout faire « à la main ». Je ne vais pas utiliser la « magie des annotations JEE ».

En effet, JEE adresse ce genre de problématique et le mécanisme d'injection de dépendances, notamment grâce à l'utilisation d'annotations ([JSR-250](#) et [JSR-330](#)).

Je commence par faire un peu de refactoring pour simplifier les choses :

1. Je crée une classe abstraite `AbstractResource` de laquelle vont hériter mes classes `Resource` (`ProjetResource`, `TicketResource`...).
2. Dans cette classe `AbstractResource`, j'ajoute un attribut `static` contenant l'instance de `ManagerFactory` à utiliser par toutes les classes `Resource`.
3. J'ajoute un `getter` pour cet attribut afin qu'il soit accessible dans les sous-classes (visibilité en `protected` suffisante).
4. Je crée également un `setter` pour cet attribut, avec une visibilité `public`, afin qu'il soit appellable par une classe tierce.
5. Je supprime les attributs d'instance `managerFactory` des classes `Resource`.

```
public abstract class AbstractResource {  
  
    private static ManagerFactory managerFactory;  
  
    protected static ManagerFactory getManagerFactory() {  
        return managerFactory;  
    }  
    public static void setManagerFactory(ManagerFactory pManagerFactory) {  
        managerFactory = pManagerFactory;  
    }  
}  
  
  
@Path("/projets")  
@Produces(MediaType.APPLICATION_JSON)  
public class ProjetResource extends AbstractResource {  
    @GET  
    @Path("{id}")  
    public Projet get(@PathParam("id") Integer pId) throws NotFoundException {  
        Projet vProjet = getManagerFactory().getProjetManager().getProjet(pId);  
        return vProjet;  
    }  
}
```

pour injecter l'instance de `ManagerFactory`, afin que toutes les instances de `Resource` y aient accès, il ne reste plus qu'à faire appel au `setter static AbstractResource.setManagerFactory(ManagerFactory)`.

Pour cela, je crée une classe `DependencyInjectionListener` implémentant l'interface `ServletContextListener`:

```
package org.example.demo.ticket.webapp.listener;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

import org.example.demo.ticket.business.ManagerFactory;
import org.example.demo.ticket.webapp.rest.resource.AbstractResource;

public class DependencyInjectionListener implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent pServletContextEvent) {
        // Création de l'instance de ManagerFactory
        ManagerFactory vManagerFactory = new ManagerFactory();

        // Injection de l'instance de ManagerFactory dans la classe AbstractResource
        AbstractResource.setManagerFactory(vManagerFactory);
    }

    @Override
    public void contextDestroyed(ServletContextEvent pServletContextEvent) {
        // NOP
    }
}
```

Je déclare ce *Listener* dans le fichier `src/main/webapp/WEB-INF/web.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee
/web-app_3_1.xsd"
          version="3.1">

    <display-name>Ticket</display-name>

    <listener>
        <listener-
class>org.example.demo.ticket.webapp.listener.DependencyInjectionListener</listener-class>
    </listener>

    <!-- ... -->
</web-app>
```

C'est fini. La classe `DependencyInjectionListener` sera instanciée au démarrage de l'application web par le serveur d'application puis celui-ci appellera sa méthode `contextInitialized(ServletContextEvent)`. Cela aura pour conséquence d'injecter l'instance de `ManagerFactory` dans la classe `AbstractResource`.

Et voilà, je viens de mettre en place une **injection de dépendances** :



1. Au déploiement de l'application, la classe `DependencyInjectionListener` est instanciée par le serveur d'application.
2. Le serveur appelle ensuite la méthode `contextInitialized(ServletContextEvent)` sur l'instance qu'il vient de créer.
3. Cette méthode crée l'instance de `ManagerFactory` et l'injecte dans l'attribut static dédié (`managerFactory`) de la classe `AbstractResource`.

Ça y est, les classes `Resource` disposent enfin d'une instance de `ManagerFactory` (via la méthode `staticgetManagerFactory()`) sans avoir à se soucier de comment créer/récupérer cette instance !

Grâce au mécanisme que je viens de mettre en place, il y a bien **inversion de contrôle** au niveau des classes `Resources`. Celles-ci ne gèrent plus la totalité du flot de contrôle de la tâche, mais seulement la partie sous leur responsabilité : prendre les paramètres passés en entrée et effectuer le traitement demandé.

Exemple avec `ProjetResource.get(Integer)` :

1. à partir de l'identifiant de projet reçu en paramètre,
2. rechercher le projet
3. et renvoyer une réponse contenant ce projet.

Vous voyez que maintenant, la classe `ProjetResource` n'a plus à gérer quoi que ce soit de plus (instanciation du `ProjetManager`, récupération du singleton `ManagerFactory`...) que le traitement demandé.

Améliorons la classe ManagerFactory

Revenons un peu sur la classe `ManagerFactory`.

```
public class ManagerFactory {
    // ...
    public ProjetManager getProjetManager() {
        return new ProjetManager();
    }
}
```

Sérieusement, on va instancier un `ProjetManager` à chaque fois que l'on va appeler la méthode `ManagerFactory.getProjetManager()`?

On ne pourrait pas en faire un Singleton ou un attribut « interne » ?

Alors je dis, *non, non et non !!*

D'abord, *non*, on ne va pas créer une instance à chaque appel. Il faut changer cela. Sinon, une instance va être créée à chaque requête et cela peut vite faire augmenter la mémoire utilisée car les instances inutilisées ne sont détruites que de temps en temps par le *garbage-collector*. En cas de sollicitation un peu forte du serveur, les performances de l'application vont s'écrouler.

Deuxièmement, *non*, on ne va pas créer de singleton. Certes, c'est son objectif de limiter le nombre d'instances, mais un vrai singleton est une classe *final*. Or cela empêche de « bouchonner » (*mock* en anglais) simplement la classe par héritage et polymorphisme lors des tests. C'est donc une solution à proscrire ici (valable pour les *Managers* et d'autant plus pour les *DAO*).



Et enfin, *non*, on ne va pas faire un attribut « interne » :

```
public class ManagerFactory {  
    // ...  
    private final ProjetManager projetManager = new ProjetManager();  
  
    public ProjetManager getProjetManager() {  
        return projetManager;  
    }  
}
```

Cela va également compliquer le bouchonnage lors des tests (comment influer sur le new ProjetManager() ou modifier l'instance dans cet attribut ?)

La solution qui je préconise : encore une fois l'***injection de dépendances*** !

```
public class ManagerFactory {  
    // ...  
  
    // Ajout d'un attribut projetManager  
    private ProjetManager projetManager;  
  
    // On renvoie désormais simplement l'attribut projetManager  
    public ProjetManager getProjetManager() {  
        return projetManager;  
    }  
  
    // Ajout d'un setter pour l'attribut projetManager  
    public void setProjetManager(ProjetManager pProjetManager) {  
        projetManager = pProjetManager;  
    }  
}  
  
  
public class DependencyInjectionListener implements ServletContextListener {  
  
    @Override  
    public void contextInitialized(ServletContextEvent pServletContextEvent) {  
        ManagerFactory vManagerFactory = new ManagerFactory();  
        // On ajoute l'injection des Managers dans la ManagerFactory.  
        vManagerFactory.setProjetManager(new ProjetManager());  
        vManagerFactory.setTicketManager(new TicketManager());  
        //...  
        AbstractResource.setManagerFactory(vManagerFactory);  
    }  
  
    // ...  
}
```

Ça commence à faire pas mal de choses à injecter ! Et en plus, quand je vais écrire les tests, il faudra que j'écrive l'injection aussi ! Il n'y a pas moyen d'alléger tout ça ? Eh bien si justement, et c'est là que *Spring* vole à notre secours ! On y arrive...

Ajouter de l'abstraction

Patience jeune padawan, il y a encore une chose importante à voir avant d'attaquer *Spring*. Vous vous rappelez du *D* de *SOLID* et de cette histoire d'abstraction ? ... Non ?

Petit rappel.

SOLID est un acronyme permettant de se rappeler 5 principes de conception en programmation orientée objet afin d'obtenir une application plus fiable et plus robuste :

- Single responsibility principle (responsabilité unique)
- Open/closed principle (ouvert/fermé)
- Liskov substitution principle (substitution de Liskov)
- Interface segregation principle (ségrégation des interfaces)
- Dependency inversion principle (inversion des dépendances)

L'inversion des dépendances c'est quoi ?

Avec le principe d'*inversion des dépendances*(*Dependency inversion principle*), Robert C. Martin énonce que :

- Les modules de haut niveau ne doivent pas dépendre des modules de plus bas niveau. Les deux doivent dépendre d'abstractions.
- Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

Euh, j'ai rien compris, c'est quoi cette histoire d'**abstraction** ?

Ce n'est pas très compliqué, vous allez voir. Dans le chapitre précédent, j'ai découpé le flot de contrôle de l'application (l'exécution du programme), afin que la responsabilité de la gestion des instances des *Managers* ne se trouve plus dans les classes *Resources* mais soit gérée de manière extérieure. Eh bien, avec l'**abstraction**, il s'agit de faire de même avec le code et non plus avec l'exécution.

Il s'agit de séparer les « contrats » de leur implémentation.

C'est clair non ?

Un exemple concret... Dans la classe *ProjetResource*, ce qui est important pour elle, c'est qu'elle puisse demander à un *Manager* de lui « *donner le projet qu'elle veut* », peu importe comment le *Manager* fait et comment il est implémenté.

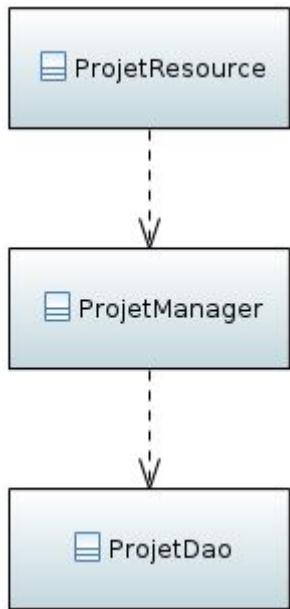
Dans ce cas :

- le contrat serait : « un *manager de projet* doit pouvoir me *envoyer le projet* correspondant à l'*identifiant que je lui donne* »
- l'implémentation serait :
 1. prendre l'*identifiant* passé en paramètre,
 2. appeler le *DAO* en lui donnant ce paramètre
 3. ...

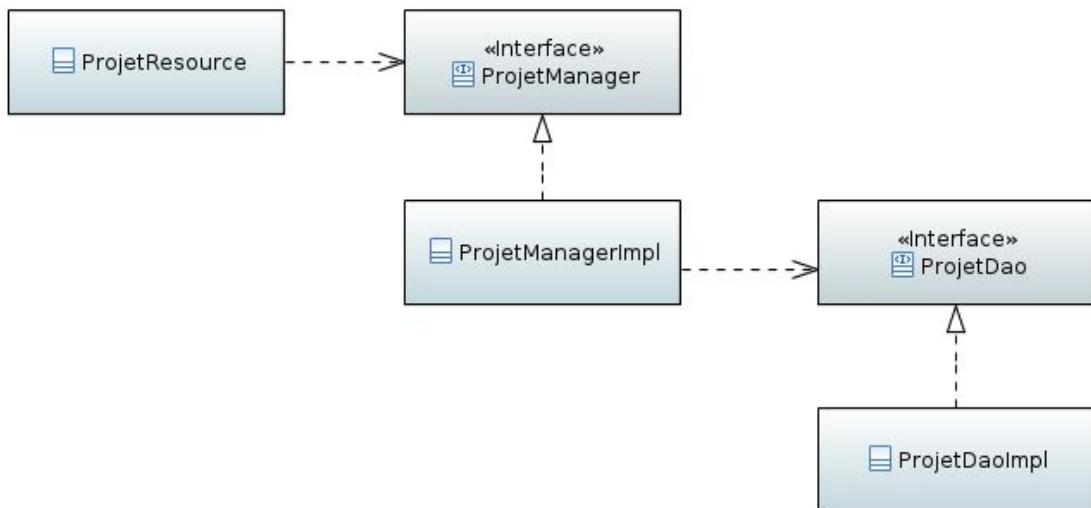
En pratique, pour écrire un contrat, il suffit d'écrire une *Interface*.

Voici le diagramme de dépendances avant l'abstraction :





Voici le diagramme de dépendances qu'il faut mettre en place :



Grâce à l'**abstraction**, les modules de haut niveau ne dépendent plus de l'implémentation faites dans les modules de plus bas niveau, mais seulement de l'abstraction. De même, cette abstraction permet de définir quels contrats doivent implémenter les modules de plus bas niveau. Ces derniers dépendent donc également de cette abstraction.

Donc si je reprends le principe d'*inversion des dépendances*(Dependency inversion principle), je pense que vous comprenez maintenant le premier point :

- Les modules de haut niveau ne doivent pas dépendre des modules de plus bas niveau. Les deux doivent dépendre d'abstractions.
- Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

Les abstractions ne doivent pas dépendre des détails... ?

En ce qui concerne le deuxième point, cela dit qu'il ne suffit pas de faire une interface simplement à partir des méthodes qui se trouveraient dans l'implémentation. Ces interfaces se doivent d'être « génériques » et non dirigées par l'implémentation. En d'autres termes, **si l'implémentation change, cela ne doit pas remettre en question l'interface.**

Par exemple, dans notre application, les utilisateurs sont enregistrés dans une table de la base de données. Si demain, il faut plutôt aller les récupérer dans un [annuaire LDAP](#), il ne faut pas que cela change l'interface *UtilisateurDao*.

Donc il ne doit pas y avoir de paramètres dans les méthodes qui dépendent de l'implémentation (*DataSource*, groupe dans l'annuaire LDAP...)

Mais en vrai, je fais quoi ?

OK, j'ai à peu près compris, mais concrètement qu'est-ce que ça donne ?

Voilà comment mettre tout cela en place avec le *ProjetManager*:

1. *ProjetManager* est désormais une interface.
2. L'ancienne classe *ProjetManager* devient la classe *ProjetManagerImpl* et implémente l'interface *ProjetManager*.
3. La *ManagerFactory* dépend de l'interface *ProjetManager* et non pas de l'implémentation *ProjetManagerImpl*. Il faut ajouter un attribut *projetManager* de type *ProjetManager* (l'interface) avec un *getter* et un *setter*.
4. Dans le *DependencyInjectionListener* il faut désormais injecter l'implémentation (classe *ProjetManagerImpl*) dans la *ManagerFactory*.



```

// ProjetManager est désormais une interface.
public interface ProjetManager {
    Projet getProjet(Integer pId);
    // ...
}

// La classe ProjetManager devient ProjetManagerImpl
// et implémente l'interface ProjetManager
public class ProjetManagerImpl implements ProjetManager {

    @Override
    public Projet getProjet(Integer pId) throws NotFoundException {
        // ...
    }

    // ...
}

public class ManagerFactory {
    // ...

    // ProjetManager est désormais une interface.
    // La Factory dépend de cette interface et non pas de l'implémentation

    // Ajout d'un attribut projetManager
    private ProjetManager projetManager;

    // On renvoie désormais simplement l'attribut projetManager
    public ProjetManager getProjetManager() {
        return projetManager;
    }
}

```

Vous pouvez appliquer le même principe à la *ManagerFactory*:

1. *ManagerFactory* devient une interface.
2. L'ancienne classe *ManagerFactory* devient la classe *ManagerFactoryImpl* et implémente l'interface *ManagerFactory*.
3. La classe *AbstractResource* est dépendante de l'interface *ManagerFactory* et pas de la classe *ManagerFactoryImpl*.

Pour mieux respecter le principe de dependency inversion et les avantages qu'il procure, il faudrait dissocier les interfaces des implementations, en les mettant dans des packages différents et des modules Maven séparés.

Exemple :

- *module ticket-business-*
 - *package de base org.example.ticket.business.contract*
- *module ticket-business-impl*
 - *package de base org.example.ticket.business.impl*



Cela dit, pour de petits projets, où les modules ne sont pas partagés entre plusieurs applications, il est acceptable de simplement séparer les interfaces et les implémentations dans des packages différents, tout en les laissant dans le même module Maven.

Grâce à l'abstraction, l'inversion de contrôle et l'injection de dépendances, vous pouvez voir que j'ai pu réduire significativement le couplage entre les classes de couches différentes.

Avec l'injection de dépendances, l'emploi des singletons détournés de leur but premier devient inutile et c'est tant mieux. En effet, les singletons :

- induisent des couplages forts aux implémentations,
- empêchent la substitution de type (violation du *L* de *SOLID*),
- et induisent ainsi une rigidité nuisant à la testabilité et la modularité du code.

En effet, il devient plus difficile de créer des bouchons (*mock*) se substituant facilement aux singletons lors de l'écriture des tests par exemple.

Questions, problèmes, angoisses... ?

Mais du coup, avec l'injection de dépendances, à quoi servent les Factories ? Elles sont toujours utiles ?

Il est possible d'aller plus loin en supprimant les *Factories* et en injectant directement :

- les *Managers* dans les classes *Resource*,
- les *DAO* dans les classes *Manager*...

Cela va apporter :

- plus de souplesse dans l'injection de dépendances (comme pour le bouchonnage des tests),
- affiner et clarifier le graphe de dépendances : on a une vision exacte des dépendances nécessaires dans une classe simplement en lisant les imports (la *Factory* masquait cela).

Mais en contrepartie :

- Cela demande beaucoup de travail au niveau de l'injection de dépendances et un temps de démarrage plus long (enfin tout est relatif).
- Il y a moins de cohésion dans les dépendances. En passant par une *Factory*, si on modifie une dépendance, ce changement se retrouve « propagé », de fait, à toutes les classes utilisant cette *Factory*. Ce n'est plus — forcément — le cas sans *Factory*.

À vous donc, de peser le pour et le contre en fonction de votre projet. Personnellement, je préfère garder les *Factories*, si je n'ai pas besoin de plus de souplesse dans les tests. Cela me permet de conserver une cohésion d'ensemble mais aussi d'avoir des points d'accès uniques sur mes éléments (*Managers*, *DAO*...).

Euh, il va falloir gérer toutes les dépendances dans le *DependencyInjectionListener*, dans la classe *Main* du module *ticket-batch*, dans les tests... ?

Ben oui, pourquoi ?

... Non, je vous fais marcher. La classe *DependencyInjectionListener* que j'ai créée ici est mon « moteur » d'injection de dépendances pour les besoins didactiques de ce cours. Vous vous rappelez, je vous ai dit que j'allais faire ça « à la main » pour avancer progressivement et pour que vous compreniez bien le mécanisme.



Maintenant, je vais passer à la vitesse supérieure : il existe des frameworks permettant de faire de l'inversion de contrôle et de l'injection de dépendances. C'est justement le cas de *Spring IoC*...

Conclusion

Petit résumé de ce que nous avons vu, avant de passer à la suite :

- *Les classes doivent avoir une seule raison de changer* : ne gérez pas le cycle de vie des classes des couches inférieures dans les classes des couches supérieures. Faites de l'**inversion de contrôle (IoC)** grâce à des **Factories** et de l'**injection de dépendances (DI)**.
- Les classes des couches supérieures ne dépendent pas des classes d'implémentations des couches inférieures, mais d'**abstractions**.
- Les classes des couches inférieures implémentent ces abstractions. Elles en dépendent donc.
- Les abstractions ne doivent pas dépendre des détails : n'orientez pas les abstractions en fonction de l'implémentation prévue.
- Les abstractions sont réalisées avec des **interfaces**.
- Celles-ci permettent d'écrire le contrat d'échange entre les couches supérieures et l'implémentation faites dans les couches inférieures.

Si vous voulez approfondir le sujet de l'injection de dépendances et la réflexion qui y a mené, vous pouvez consulter le célèbre article de [Martin Fowler](#) : [*Inversion of Control Containers and the Dependency Injection pattern*](#).

Dans les chapitres suivants, nous allons mettre en pratique tout cela en mettant en œuvre une **inversion de contrôle** avec *Spring IoC*.

Une injection de dépendances plus sophistiquée

Vous avez compris le principe de l'**inversion de contrôle (IoC)** et l'avantage de faire de l'**injection de dépendances (DI)**.

Faire cela à la main peut vite devenir fastidieux et source d'erreur. Un oubli est vite arrivé !

L'injection de dépendances avec une approche séquentielle

Actuellement, dans l'application web, toute l'injection de dépendances est « pilotée » par la classe *DependencyInjectionListener*. L'approche adoptée est du genre :

1. Je crée toutes les instances des *Managers*.
2. Je crée l'instance de la *ManagerFactory*.
3. J'injecte toutes les instances des *Managers* dans la *ManagerFactory* via des *setters*.
4. J'injecte l'instance de la *ManagerFactory* dans les classes qui en ont besoin (ici *AbstractResource*).
5. Et on fait de même avec les *DAO* et la *DaoFactory*, etc.

Avec cette approche, je peux rencontrer quelques soucis. Par exemple, imaginez que *ProjetDao* ait besoin de *UtilisateurDao* pour pouvoir charger l'utilisateur qui a créé le projet quand il va récupérer ce dernier en base de données. Il faudrait alors :

- soit injecter la *DaoFactory* dans *ProjetDao* ;



- soit lui injecter directement l'*UtilisateurDao*.

Cela impose donc de bien ordonner les instanciations et les injections. Et ça, ça commence à complexifier la tâche de mon « injection de dépendances maison » !

Mais, rassurez-vous, je ne vais pas vous laisser comme ça, une autre manière de faire est possible...

Utiliser un « IoC container »

Spring apporte une solution visant à simplifier l'injection de dépendances en changeant un petit peu d'approche. Il utilise un « **conteneur d'inversion de contrôle** », **IoC container** en anglais.

L'idée est que :

1. chaque classe entrant en jeu dans l'injection de dépendances se déclare dans l'IoC container (*AbstractResource*, *ManagerFactoryImpl*, *ProjetManagerImpl*, *ProjetDaoImpl*...) ;
2. pour chaque objet, on indique les dépendances à injecter (ex.: *ManagerFactoryImpl* a besoin d'un *ProjetManager*, d'un *TicketManager*...) ;
3. l'IoC container se charge d'instancier et d'injecter les dépendances afin de produire la grappe d'objets pleinement configurée.

C'est plutôt cool, non ?

Dans le jargon de Java EE et de *Spring*, les objets manipulés dans l'injection de dépendances s'appellent des **managed beans** (JEE) ou simplement **beans** (*Spring*).

Voici à quoi peut ressembler la configuration de l'IoC container de *Spring IoC*:

```

<beans>
    <!-- ===== Déclaration des DAO ===== -->
    <bean id="projetDao" class="org.example.demo.ticket.consumer.impl.dao.ProjetDaoImpl">
        <property name="utilisateurDao" ref="utilisateurDao" />
    </bean>

    <bean id="utilisateurDao" class="org.example.demo.ticket.consumer.impl.dao.UtilisateurDaoImpl" />

    <!-- ===== DaoFactory ===== -->
    <bean id="daoFactory" class="org.example.demo.ticket.consumer.impl.dao.DaoFactoryImpl">
        <property name="projetDao" ref="projetDao" />
        <property name="utilisateurDao" ref="utilisateurDao" />
    </bean>

    <!-- ===== Déclaration des Managers ===== -->
    <bean id="projetManager" class="org.example.demo.ticket.business.impl.manager.ProjetManagerImpl">
        <property name="daoFactory" ref="daoFactory" />
    </bean>
    <bean id="ticketManager" class="org.example.demo.ticket.business.impl.manager.TicketManagerImpl">
        <property name="daoFactory" ref="daoFactory" />
    </bean>

    <!-- ===== ManagerFactory ===== -->
    <bean id="managerFactory" class="org.example.demo.ticket.business.impl.ManagerFactoryImpl">
        <property name="projetManager" ref="projetManager" />
        <property name="ticketManager" ref="ticketManager" />
    </bean>
</beans>

```

Même si c'est assez parlant, voici quelques explications :

- On déclare les *beans* (élément *bean*). Chaque bean a :
 - un identifiant (attribut *id*),
 - et on précise quelle est la classe d'implémentation (attribut *class*).
- Pour chaque bean on indique quels sont les attributs de l'objet à injecter (élément *property*) :
 - en précisant le nom de l'attribut (attribut *name*),
 - l'identifiant du bean à injecter dedans (attribut *ref*).

Je ne vois pas bien la différence et l'utilité de l'IoC container ?

La différence réside dans le fait que l'injection de dépendances adopte une **approche déclarative** au lieu d'une approche séquentielle d'instanciations et d'injections. On n'est alors plus soumis à un ordre précis de création d'objets pour pouvoir gérer l'injection, et toute la configuration de l'injection pour une classe se trouve à un seul endroit.

Ceci permet également d'adresser de manière transparente les dépendances cycliques — même si elles devraient rester rares.

Imaginez que :

- *ProjetDaoImpl* ait besoin d'un *TicketDao* et d'un *UtilisateurDao* ;
- *TicketDaoImpl* ait besoin d'un *ProjetDaoImpl* et d'un *UtilisateurDao* ;



Avec une approche séquentielle (sans IoC container)

Avec mon système d'injection « à la main », cela donnerait :

```
utilisateurDaoImpl vUtilisateurDaoImpl = new UtilisateurDaoImpl();

ProjetDaoImpl vProjetDaoImpl = new ProjetDaoImpl();
vProjetDaoImpl.setUtilisateurDao(vUtilisateurDaoImpl);
// La deuxième injection n'est pas encore réalisable
// (vTicketDaoImpl n'existe pas encore)
// vProjetDaoImpl.setTicketDao(vTicketDaoImpl);

TicketDaoImpl vTicketDaoImpl = new TicketDaoImpl();
vTicketDaoImpl.setProjetDao(vProjetDaoImpl);
vTicketDaoImpl.setUtilisateurDao(vUtilisateurDaoImpl);

// Maintenant je peux injecter le TicketDao dans le ProjetDaoImpl
vProjetDaoImpl.setTicketDao(vTicketDaoImpl);

//
```

On pourrait ré-ordonner les choses en faisant d'abord toutes les instantiations, puis toutes les injections :

Mais alors le risque d'oublier une injection est plus grand car l'initialisation d'un objet est répartie sur 2 blocs distincts.

Avec une approche déclarative (avec IoC container)

Avec une approche déclarative, je ne me soucie plus de l'ordre, je ne fais que déclarer les éléments :

```
<beans>
    <bean id="projetDao" class="org.example.demo.ticket.consumer.impl.dao.ProjetDaoImpl">
        <property name="ticketDao" ref="ticketDao" />
        <property name="utilisateurDao" ref="utilisateurDao" />
    </bean>

    <bean id="ticketDao" class="org.example.demo.ticket.consumer.impl.dao.TicketDaoImpl">
        <property name="projetDao" ref="projetDao" />
        <property name="utilisateurDao" ref="utilisateurDao" />
    </bean>

    <bean id="utilisateurDao" class="org.example.demo.ticket.consumer.impl.dao.UtilisateurDaoImpl" />
</beans>
```

Et cette approche déclarative prend tout son sens si on utilise directement les annotations de la [JSR-250](#) et [JSR-330](#) dans le code source à la place d'un fichier de configuration :

```
@ManagedBean  
public class ProjetDaoImpl implements ProjetDao {  
    @Inject  
    private TicketDao ticketDao;  
  
    @Inject  
    private UtilisateurDao utilisateurDao;  
  
    // ...  
}  
  
  
@ManagedBean  
public class TicketDaoImpl implements TicketDao {  
    @Inject  
    private ProjetDao projetDao;  
  
    @Inject  
    private UtilisateurDao utilisateurDao;  
  
    // ...  
}  
  
  
@ManagedBean  
public class UtilisateurDaoImpl implements UtilisateurDao {  
    // ...  
}
```

OK, ça fait peut-être beaucoup de choses d'un coup. Ce que vous devez retenir et bien comprendre, c'est le principe de l'**IoC container** et de l'**approche déclarative**.

Ne vous souciez pas de retenir la syntaxe de configuration, c'est justement ce que je vais détailler pas à pas dans le prochain chapitre. Vous allez enfin utiliser *Spring* !

L'inversion de contrôle avec Spring

La première chose à faire pour utiliser *Spring* c'est d'ajouter les dépendances requises dans le projet Maven.

1. Je commence par ajouter la définition des dépendances dans la section *dependencyManagement* du projet racine (fichier ticket/pom.xml) :

```
<project>
  ...
  <properties>
    ...
    <spring.version>4.3.11.RELEASE</spring.version>
  </properties>
  <dependencyManagement>
    <dependencies>
      ...
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-framework-bom</artifactId>
        <version>${spring.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

Comme vous pouvez le remarquer, je ne déclare pas tous les modules de Spring un à un, mais j'utilise ce qu'on appelle une « *Bill Of Materials* » (BOM). C'est un fichier POM généralement fourni par les frameworks et qui liste l'ensemble des composants de ce framework avec leur version. Il suffit d'importer celui-ci dans la section *dependencyManagement* de notre POM pour ajouter la liste des dépendances fournies par ce framework

2. Dans le module *ticket-technical*, j'ajoute la dépendance vers *Spring Context*:

```
<project>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
    </dependency>
  </dependencies>
  ...
</project>
```

Créer un premier fichier de configuration

Avec Spring, il y a plusieurs moyens de configurer l'IoC Container :

- avec un fichier XML ;
- en utilisant les annotations des JSR-250 et JSR-330 ;
- en utilisant les annotations spécifiques à Spring ;
- via le code de l'application ;
- et même en mixant plusieurs de ces moyens !

Pour l'instant, je vais me concentrer uniquement sur l'utilisation d'un fichier XML.

Je crée un fichier applicationContext.xml dans le répertoire src/main/resources du module *ticket-webapp* et y ajoute la configuration de l'injection de dépendances :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- ===== Déclaration des Managers ===== -->
    <bean id="projetManager"
        class="org.example.demo.ticket.business.impl.manager.ProjetManagerImpl"/>
    <bean id="ticketManager"
        class="org.example.demo.ticket.business.impl.manager.TicketManagerImpl"/>

    <!-- ===== ManagerFactory ===== -->
    <bean id="managerFactory" class="org.example.demo.ticket.business.ManagerFactory">
        <property name="projetManager" ref="projetManager"/>
        <property name="ticketManager" ref="ticketManager"/>
    </bean>

    <!-- ===== AbstractResource ===== -->
    <bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
        <property name="targetClass"
            value="org.example.demo.ticket.webapp.rest.resource.AbstractResource"/>
        <property name="targetMethod" value="setManagerFactory"/>
        <property name="arguments" ref="managerFactory"/>
    </bean>
</beans>
```

Par défaut, dans l'IoC container de Spring, les beans sont des « singltons » : Spring ne les instancie qu'une seule fois. Ce comportement peut être modifié en utilisant l'attribut *scope*.

En ce qui concerne la classe *AbstractResource*, il faut appeler la méthode **static***setManagerFactory(ManagerFactory)*. Cela se fait de manière particulière dans la configuration Spring, via la définition d'un bean sans identifiant et avec la classe *org.springframework.beans.factory.config.MethodInvokingFactoryBean*

Initialiser Spring dans l'application web

En informatique, l'initialisation d'une application est souvent appelée *bootstrapping* (j'y reviendrai).

Il reste une dernière étape : initialiser l'IoC container de Spring dans l'application web en remplaçant l'actuel listener *DependencyInjectionListener* par celui fourni par *Spring* (module *spring-web*).

1. Ajoutez la dépendance vers *spring-web* dans le POM du module *ticket-webapp* :

```
<project>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
    </dependency>
  </dependencies>
</project>
```

2. Ajoutez le *listener* de Spring et sa configuration dans le fichier *src/main/webapp/WEB-INF/web.xml* :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
          version="3.1">
  <!-- ... -->
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:/applicationContext.xml</param-value>
  </context-param>
</web-app>
```

3. Supprimez l'ancien *listener* dans le fichier *src/main/webapp/WEB-INF/web.xml* (*org.example.demo.ticket.webapp.listener.DependencyInjectionListener*).

4. Supprimez la classe *org.example.demo.ticket.webapp.listener.DependencyInjectionListener*. Et voilà, il n'y a rien d'autre à faire, *Spring* fait tout le reste.

Initialiser Spring dans les batches

Pour le module *ticket-batch*, il faut créer également un fichier *applicationContext.xml* dans le répertoire *src/main/resources* du module :



```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- ===== Déclaration des Managers ===== -->
    <bean id="projetManager"
          class="org.example.demo.ticket.business.impl.manager.ProjetManagerImpl"/>
    <bean id="ticketManager"
          class="org.example.demo.ticket.business.impl.manager.TicketManagerImpl"/>

    <!-- ===== ManagerFactory ===== -->
    <bean id="managerFactory" class="org.example.demo.ticket.business.ManagerFactory">
        <property name="projetManager" ref="projetManager"/>
        <property name="ticketManager" ref="ticketManager"/>
    </bean>

    ...
</beans>

```

Et dans la classe la méthode *main*, il faut charger l'*application context* Spring à partir du fichier XML :

```

package org.example.demo.ticket.batch;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] pArgs) throws TechnicalException {
        ApplicationContext vApplicationContext
            = new ClassPathXmlApplicationContext("classpath:/applicationContext.xml");

        // Il est possible de récupérer un bean dans ce contexte :
        ManagerFactory vManagerFactory
            = vApplicationContext.getBean("managerFactory", ManagerFactory.class);

        // suite de l'implémentation des batches...
    }
}

```

Configurer l'IoC avec des annotations

Comme je le disais plus tôt, il y a plusieurs moyens de configurer l'IoC container de Spring. À la place ou en complément de fichiers XML, vous pouvez aussi utiliser des annotations. Avec les annotations, la définition des besoins en dépendances est au plus près du code.

Au niveau des annotations vous avez le choix :

- soit utiliser celles des JSR-250 et JSR-330 ;



- soit utiliser les annotations spécifiques à Spring ;
- soit mixer les deux.

Les annotations spécifiques à Spring proposent [plus de fonctionnalités](#) que celles des JSR-250 et JSR-330, mais vous créez alors une dépendance forte entre le code de votre application et l'API du framework Spring.

Pour pouvoir utiliser les annotations des JSR-250 et/ou JSR-330 vous devez ajouter les dépendances vers ces API dans Maven :

```
<!-- JSR-250 -->
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.2</version>
</dependency>
<!-- JSR-330 -->
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

Je vais maintenant vous montrer comment utiliser les annotations. Je ne vais pas passer sur toutes les annotations, mais seulement les plus importantes, vous pourrez toujours consulter la [documentation](#) pour en savoir plus.

Il faut distinguer 3 parties dans la configuration :

- un bean qui déclare ses dépendances, c'est-à-dire les éléments qu'il faut lui injecter
- un bean qui se déclare auprès de l'IoC container pour indiquer qu'il peut être injecté dans d'autres beans
- l'initialisation de l'IoC container (le *bootstrapping*)

Définir ses dépendances

Quand un élément doit être la cible d'une injection de dépendances, il suffit de l'annoter avec @Inject (JSR-330) ou @Autowired (Spring).

```
import javax.inject.Inject;
// ...

public class ManagerFactoryImpl implements ManagerFactory {

    @Inject
    private ProjetManager projetManager;

    @Inject
    private TicketManager ticketManager;

    // ...
}
```

Les annotations @Inject et @Autowired peuvent être positionnées :

- sur un attribut, comme ci-dessus
- sur le *setter* d'un attribut
- sur une méthode ayant un ou plusieurs paramètres (ce sont ces paramètres qui seront injectés)
- sur un constructeur ayant un ou plusieurs paramètres (ce sont ces paramètres qui seront injectés)

Se déclarer en tant que bean

Afin que les classes se signalent en tant que bean injectable, il suffit de les annoter avec @ManagedBean (JSR-250), @Named (JSR-330) ou @Component (Spring).

```
import javax.inject.Named;
// ...

@Named
public class ProjetManagerImpl implements ProjetManager {
    //...
}
```

Il est possible de définir l'identifiant du bean dans l'annotation. Par exemple :

```
import javax.inject.Named;
// ...

@Named("projetManager")
public class ProjetManagerImpl implements ProjetManager {
    //...
}
```

Si l'identifiant du bean n'est pas spécifié, son identifiant sera le nom de la classe avec la première lettre en minuscule. Dans cet exemple, l'identifiant du bean sera *projetManagerImpl* :

```
@Named
public class ProjetManagerImpl implements ProjetManager {
    //...
}
```

Initialiser l'IoC container

Pour initialiser le contexte vous avez plusieurs solutions. Je vais vous en proposer 2 qui permettent de combiner les configurations à base de fichiers XML et d'annotations.

Dans tous les cas, dans le fichier XML *applicationContext.xml*, il faut supprimer tout ce qui est maintenant géré par les annotations (déclaration du bean *projetManager*, injection du bean *projetManager* dans le bean *managerFactory*...).

Ajout de contexte dans la configuration XML



Avec cette première solution, vous conservez l'initialisation actuelle qui charge le fichier XML applicationContext.xml.

Vous ajoutez simplement dans ce fichier les éléments suivants :

- <context:annotation-config/> : prend en compte la configuration des injections (@Inject, @Autowired...)
- <context:component-scan base-package="..."/> : scanne les packages pour trouver les beans qui se déclarent (@Named, @Component...)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- Prend en compte la configuration des injections (@Inject...) -->
    <context:annotation-config/>

    <!-- Scanne les packages pour trouver les beans qui se déclarent (@Named...) -->
    <context:component-scan base-package="org.example.demo.ticket"/>

    <!-- ... -->
</beans>
```

Remarquez l'ajout indispensable du *namespace XML* context.

Création d'une classe Java de configuration

La deuxième solution consiste à créer une classe Java de configuration utilisant des annotations spécifiques à Spring :

```
package org.example.demo.ticket.webapp.bootstrap;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;

@Configuration
@ComponentScan("org.example.demo.ticket")
@ImportResource("classpath:/applicationContext.xml")
public class SpringConfiguration {

}
```

- @Configuration : indique que cette classe propose des éléments de configuration Spring
- @ComponentScan("org.example.demo.ticket") : indique qu'il faut scanner le package org.example.demo.ticket et ses sous-packages à la recherche de beans se déclarant

- `@ImportResource("classpath:/applicationContext.xml")` : il est toujours possible d'ajouter des éléments de configuration dans des fichiers XML. Cette annotation permet d'importer ces fichiers XML.

Au niveau du *bootstrapping*, il faut maintenant utiliser cette nouvelle classe comme point d'entrée de la configuration et non plus le fichier `applicationContext.xml`.

Pour l'application web, c'est dans le fichier `src/main/webapp/WEB-INF/web.xml` que cela se passe :

- le `context-param-contextConfigLocation` indique maintenant la classe de configuration au lieu du fichier XML ;
- ajout du `context-param-contextClass` indiquant à Spring d'utiliser la classe `AnnotationConfigWebApplicationContext` pour charger l'*application context* Spring au lieu de la classe par défaut (`XmlWebApplicationContext`).

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee
/web-app_3_1.xsd"
          version="3.1">

    <display-name>Ticket</display-name>

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>org.example.demo.ticket.webapp.bootstrap.SpringConfiguration</param-value>
    </context-param>
    <context-param>
        <param-name>contextClass</param-name>
        <param-
value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>
    </context-param>
</web-app>
```

Pour le module *ticket-batch*, il faut utiliser le même principe :

- créer une classe de configuration Spring : `SpringConfiguration` ;
- utiliser la classe `AnnotationConfigApplicationContext` au lieu de la classe `ClassPathXmlApplicationContext` pour charger l'*application context* Spring.

```
package org.example.demo.ticket.batch;

import org.example.demo.ticket.batch.bootstrap.SpringConfiguration;
import org.example.demo.ticket.business.ManagerFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {

    public static void main(String[] pArgs) throws TechnicalException {
        ApplicationContext vApplicationContext = new
AnnotationConfigApplicationContext(SpringConfiguration.class);

        ManagerFactory vManagerFactory = vApplicationContext.getBean("managerFactoryImpl",
ManagerFactory.class);
        // ...
    }
}
```

Annotation vs XML

Pour résumer, la configuration uniquement en XML est assez verbeuse, mais elle n'introduit aucun couplage des objets manipulés, tant à Spring, qu'au mécanisme d'injection de dépendances (les objets ne savent d'ailleurs même pas qu'ils font l'objet d'une injection de dépendances automatisée). La configuration est centralisée mais sa séparation du code peut aussi être source d'erreur en cas d'oubli.

D'un autre côté, avec les annotations, la configuration est au plus près du code, les risques d'oubli sont moins importants, mais la configuration est disséminée dans toutes l'application. De plus, le code de votre application se retrouve couplé à l'API Java EE (JSR-250 et JSR-330) ou celle du framework Spring. Les annotations sont plus invasives.

Enfin, il est possible de tirer les avantages des deux types de configuration en les mixant. Cependant, si vous faites cela, je ne saurais trop vous conseiller de le faire de manière cohérente et consistante dans toute l'application, mais également de documenter votre manière de faire. Mettez au moins un commentaire dans votre fichier XML indiquant qu'une partie de la configuration se fait via des annotations et si possible vos règles disant quand vous utilisez une annotation et quand vous utilisez le XML.

Chaque approche a ses avantages et inconvénients. À vous de choisir en fonction de vos priorités et de vos préférences.

Dans le chapitre suivant, je vous donnerai des conseils pour mieux utiliser Spring et vous montrerai qu'il est possible d'aller un peu plus loin en faisant de l'injection de configuration par exemple.

Mieux utiliser Spring

Les concepts d'inversion de contrôle et d'injection de dépendances vous sont désormais familiers. Vous savez ce qu'est l'abstraction et comment mettre tout cela en oeuvre avec Spring. Dans ce chapitre, je vais aborder quelques points afin d'améliorer et étendre votre utilisation de Spring.

Organiser la configuration de Spring IOC

Compartimenter la configuration

Une des premières choses à faire maintenant que vous avez acquis les bases de *Spring IoC*, c'est de mieux organiser sa configuration. En effet, vous avez pu vous rendre compte que le fichier XML de configuration du contexte va vite devenir énorme car il va falloir y lister tous les *beans* (*managers*, *DAO*, *factories*...) et leurs injections de dépendances. De plus, dans notre exemple, une grande partie du fichier XML sera identique entre celui du module *ticket-webapp* et celui du module *ticket-batch*.

Sachez qu'il est possible d'importer dans le fichier XML de configuration, d'autres fichiers XML de configuration. Ainsi, ce que je vous conseillerai, c'est de répartir votre configuration du contexte sur plusieurs fichiers XML, mais pas n'importe comment !

Séparer le bootstrapping

Pour commencer, je sépare le *bootstrapping* du reste de la configuration. Pour le moment ça ne change pas grand-chose, mais ça ne va pas durer.

Je crée un fichier bootstrapContext.xml et dans ce fichier j'importe le fichier applicationContext.xml actuel :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- Inclusion d'autres fichiers de contexte Spring -->
    <import resource="classpath:/applicationContext.xml" />
</beans>
```

Bien sûr, il ne faut pas oublier de modifier le chargement de la configuration dans les modules *ticket-batch* et *ticket-webapp*.

```

package org.example.demo.ticket.batch;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] pArgs) throws TechnicalException {
        ApplicationContext vApplicationContext
            = new ClassPathXmlApplicationContext("classpath:/bootstrapContext.xml");

        // ...
    }
}

```

```

<web-app>
    <!-- ... -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:/bootstrapContext.xml</param-value>
    </context-param>
</web-app>

```

Séparer la configuration de chaque module

L'idée est de :

1. Faire un fichier de contexte par module/couche applicative et de le mettre dans le module Maven concerné (businessContext.xml dans le module *ticket-business*, consumerContext.xml dans le module *ticket-consumer*...).
2. D'importer ces fichiers dans le fichier de bootstrapping (bootstrapContext.xml). *ticket-batch* et *ticket-webapp*.

Afin d'éviter les éventuels problèmes de conflit dans le nom des fichiers dans le classpath, je vous conseille de mettre chaque fichier dans le package qui correspond au module.

Par exemple, le fichier businessContext.xml irait dans le package org.example.demo.ticket.business, ce qui correspond au répertoire src/main/resources/org/example/demo/ticket/business du module *ticket-business*.

Le fichier src/main/resources/applicationContext.xml du module *ticket-webapp* devient src/main/resources/org/example/demo/ticket/webapp/webappContext.xml...

Exemple pour le fichier consumerContext.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- ===== Déclaration des DAO ===== -->
    <bean id="projetDao" class="org.example.demo.ticket.consumer.impl.dao.ProjetDaoImpl"/>
    <bean id="ticketDao" class="org.example.demo.ticket.consumer.impl.dao.TicketDaoImpl"/>

    <!-- ===== DaoFactory ===== -->
    <bean id="daoFactory" class="org.example.demo.ticket.business.impl.DaoFactory">
        <property name="projetDao" ref="projetDao"/>
        <property name="ticketDao" ref="ticketDao"/>
    </bean>
</beans>
```

Exemple pour le fichier businessContext.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- ===== Déclaration des Managers ===== -->
    <bean id="projetManager" class="org.example.demo.ticket.business.impl.manager.ProjetManagerImpl">
        <property name="daoFactory" ref="daoFactory" />
    </bean>
    <bean id="ticketManager" class="org.example.demo.ticket.business.impl.manager.TicketManagerImpl">
        <property name="daoFactory" ref="daoFactory" />
    </bean>

    <!-- ===== ManagerFactory ===== -->
    <bean id="managerFactory" class="org.example.demo.ticket.business.impl.ManagerFactoryImpl">
        <property name="projetManager" ref="projetManager"/>
        <property name="ticketManager" ref="ticketManager"/>
    </bean>
</beans>
```

Vous pouvez remarquer que dans ce fichier businessContext.xml, je fais rérférence à un bean déclaré dans le fichier consumerContext.xml : le bean *daoFactory*. Cela ne pose pas de problème car j'importe les deux fichiers dans le fichier de bootstrapping et Spring se charge de fusionner toute la configuration apportée par ces fichiers.

Le fichier bootstrapContext.xml :



```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- Inclusion d'autres fichiers de contexte Spring -->
    <import resource="classpath:/org/example/demo/ticket/business/consumerContext.xml" />
    <import resource="classpath:/org/example/demo/ticket/business/businessContext.xml" />
    <import resource="classpath:/org/example/demo/ticket/webapp/webappContext.xml" />
</beans>
```

Opter pour d'autres types d'injection de dépendances

Il existe plusieurs types d'injection de dépendances. Vous les avez déjà aperçus dans le chapitre précédent mais sans forcément y faire attention.

Ces types d'injections sont :

- l'injection par *constructeur*
- l'injection par *setter*
- l'injection par *champs* ou *attribut*

L'injection par constructeur

Comme son nom l'indique, l'injection se fait au niveau du constructeur. Voici ce que cela donne avec la classe ManagerFactoryImpl et les annotations :

```

public class ManagerFactoryImpl implements ManagerFactory {

    @Inject
    public ManagerFactoryImpl(ProjetManager pProjetManager,
                             TicketManager pTicketManager) {
        this.projetManager = pProjetManager;
        this.ticketManager = pTicketManager;
    }

    // ...
}
```

Sans les annotations :

```
public class ManagerFactoryImpl implements ManagerFactory {  
  
    public ManagerFactoryImpl(ProjetManager pProjetManager,  
                             TicketManager pTicketManager) {  
        this.projetManager = pProjetManager;  
        this.ticketManager = pTicketManager;  
    }  
  
    // ...  
}
```

```
<beans>  
    <bean id="managerFactory" class="org.example.demo.ticket.business.impl.ManagerFactoryImpl">  
        <constructor-arg ref="projetManager"/>  
        <constructor-arg ref="ticketManager"/>  
    </bean>  
</beans>
```

L'ordre dans lequel vous écrivez les éléments `<constructor-arg>` doit correspondre à l'ordre des paramètres du constructeur.

Avec l'injection par constructeur, le bean est pleinement configuré dès le retour du constructeur. Cela évite d'avoir des dépendances à null. Vous pouvez aussi créer des beans immuables, sans *setter*, car ils ne sont pas nécessaires. Cependant, il est impossible d'avoir un cycle de dépendances entre les beans.

L'injection par setter

Cette fois-ci, l'injection se fait grâce aux setters. Voici ce que cela donne avec la classe ManagerFactoryImpl et les annotations :

```
public class ManagerFactoryImpl implements ManagerFactory {  
  
    private ProjetManager projetManager;  
    private TicketManager ticketManager;  
  
    public ManagerFactoryImpl() {  
    }  
  
    public ProjetManager getProjetManager() {  
        return projetManager;  
    }  
  
    @Inject  
    public void setProjetManager(ProjetManager pProjetManager) {  
        projetManager = pProjetManager;  
    }  
  
    public TicketManager getTicketManager() {  
        return ticketManager;  
    }  
  
    @Inject  
    public void setTicketManager(TicketManager pTicketManager) {  
        ticketManager = pTicketManager;  
    }  
  
    //...  
}
```

Sans les annotations :

```
public class ManagerFactoryImpl implements ManagerFactory {  
  
    private ProjetManager projetManager;  
    private TicketManager ticketManager;  
  
    public ManagerFactoryImpl() {  
    }  
  
    public ProjetManager getProjetManager() {  
        return projetManager;  
    }  
    public void setProjetManager(ProjetManager pProjetManager) {  
        projetManager = pProjetManager;  
    }  
    public TicketManager getTicketManager() {  
        return ticketManager;  
    }  
    public void setTicketManager(TicketManager pTicketManager) {  
        ticketManager = pTicketManager;  
    }  
  
    //...  
}
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xmlns="http://www.springframework.org/schema/beans"  
       xsi:schemaLocation="  
           http://www.springframework.org/schema/beans  
           http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
    <!-- ===== ManagerFactory ===== -->  
    <bean id="managerFactory" class="org.example.demo.ticket.business.impl.ManagerFactoryImpl">  
        <property name="projetManager" ref="projetManager"/>  
        <property name="ticketManager" ref="ticketManager"/>  
    </bean>  
</beans>
```

Avec l'injection par *setter*, vous pouvez gérer des beans avec un cycle de dépendances. Vous pouvez aussi modifier leurs dépendances au cours de l'exécution de l'application. Attention toutefois, lors de la construction du bean par Spring, les dépendances ne sont pas injectées tout de suite. Cela se passe en deux temps : d'abord le bean est instancié, puis les dépendances sont injectées dans l'ordre que Spring a déterminé suite à l'analyse de la hiérarchie de dépendances issue de la configuration. Ce qui est à prendre en considération si vous faites des traitements dans le constructeur.

L'injection par champs

Ici, l'injection se fait directement dans les champs (attributs) de la classe par introspection, sans passer par les *setters*. Voici ce que cela donne avec la classe ManagerFactoryImpl et les annotations :

```

public class ManagerFactoryImpl implements ManagerFactory {
    @Inject
    private ProjetManager projetManager;
    @Inject
    private TicketManager ticketManager;

    //...
}

```

Vous n'êtes plus obligé de créer les *setters* et donc avoir des beans immuables. Vous pouvez gérer des beans avec un cycle de dépendances.

Attention toutefois, lors de la construction du bean par Spring, les dépendances ne sont pas injectées tout de suite. C'est le même principe que pour l'injection par *setters*.

Bien entendu, vous pouvez mixer tous ces types d'injection mais comme pour l'utilisation du XML et des annotations, faites-le de manière cohérente et consistante tout au long du projet. Vous pouvez par exemple, injecter toutes les dépendances obligatoires via les constructeurs et les dépendances facultatives via des setters. Veuillez noter toutefois qu'un nombre important de paramètres dans un constructeur est un indicateur de code potentiellement mal conçu...

Factoriser la configuration avec l'héritage de définition

Quand vous déclarez un bean et précisez sa configuration, cela s'appelle la *définition d'un bean* dans le jargon de Spring.

Si je reprends l'exemple des *Managers*, vous remarquez que j'injecte à chaque fois le bean *daoFactory*:

```

<beans>
    <bean id="projetManager" class="org.example.demo.ticket.business.impl.manager.ProjetManagerImpl">
        <property name="daoFactory" ref="daoFactory" />
    </bean>
    <bean id="ticketManager" class="org.example.demo.ticket.business.impl.manager.TicketManagerImpl">
        <property name="daoFactory" ref="daoFactory" />
    </bean>

    <!-- ... -->
</beans>

```

Eh bien, il est possible de factoriser cette configuration en créant un bean abstrait :

```

<beans>
    <bean id="abstractManager" abstract="true">
        <property name="daoFactory" ref="daoFactory" />
    </bean>

    <bean id="projetManager" class="org.example.demo.ticket.business.impl.manager.ProjetManagerImpl"
          parent="abstractManager"/>
    <bean id="ticketManager" class="org.example.demo.ticket.business.impl.manager.TicketManagerImpl"
          parent="abstractManager"
    <!-- ... -->
</beans>

```



Ici, j'ai créé le bean *abstractManager* en spécifiant l'attribut `abstract="true"`. Ce bean ne sera pas instancié par Spring — d'ailleurs aucune classe n'est précisée, il sert simplement de "modèle". Et, dans les beans *projetManager* et *ticketManager*, j'indique utiliser le bean *abstractManager* comme parent grâce à l'attribut `parent`.

Vous pouvez également :

- préciser une classe pour le bean parent ;
- surcharger une propriété de la définition du bean parent dans le bean enfant en l'ajoutant à sa définition...

Vous trouverez plus de détail dans la [documentation officielle](#) de Spring.

Utiliser d'autres scopes pour les beans

Je vous disais plus tôt dans le cours que, par défaut, les beans créés par Spring sont des « *singletons* ». En fait, ils n'implémentent pas le design pattern *Singleton* mais cela veut dire que Spring ne créera qu'un seule instance de ces beans et injectera cette instance partout où vous y faites référence. Plus précisément, ces beans ont un ***scopesingleton***.

Il existe d'autres scopes, que vous pouvez utiliser en spécifiant l'attribut `scope` dans la définition du bean.

Vous avez notamment le scope *prototype*. Avec ce scope, une instance du bean est créée par Spring à chaque fois que celui-ci est utilisé, c'est-à-dire :

- à chaque que vous y faites référence dans une injection (ex: `<property name="x" ref="beanId" />`)
- à chaque fois que vous le demandez au contexte de Spring via `ApplicationContext.getBean(...)`

Les principaux scopes sont *singleton* et *prototype*. Vous trouverez plus de détails, et les autres scopes dans la [documentation officielle](#) de Spring.

Les logs

Pour produire ses logs, Spring utilise [Apache Commons Logging](#). Vous pouvez facilement récupérer ces logs dans [Apache Log4J 2](#).

Pour cela, il vous faut ajouter les dépendances Maven suivantes :

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.6.2</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-jcl</artifactId>
    <version>2.6.2</version>
    <scope>runtime</scope>
</dependency>
```

Et vous pouvez obtenir les logs de Spring via le logger `org.springframework` :

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="info">
    <ThresholdFilter level="trace"/>

    <Appenders>
        <Console name="STDOUT">
            <PatternLayout pattern="%highlight{%-5level} [%t] %c : %m%n"/>
        </Console>
    </Appenders>

    <Loggers>
        <Logger name="org.springframework" level="debug" additivity="false">
            <AppenderRef ref="STDOUT"/>
        </Logger>

        <Logger name="org.example.demo.ticket" level="debug" additivity="false">
            <AppenderRef ref="STDOUT"/>
        </Logger>

        <Root level="warn">
            <AppenderRef ref="STDOUT"/>
        </Root>
    </Loggers>
</Configuration>
```



Utiliser Spring pour injecter d'autres éléments

Vous pouvez utiliser Spring pour injecter d'autres éléments que des beans/dépendances — du moins des éléments différents de ce que nous avons vu jusqu'à maintenant. Je vais vous montrer quelques exemples.

Utiliser des valeurs de fichiers properties

```
<beans>
    <!-- chargement du fichier conf/db-ticket.properties -->
    <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>file:${TICKET_HOME}/conf/db-ticket.properties</value>
            </list>
        </property>
        <property name="ignoreUnresolvablePlaceholders" value="false"/>
    </bean>

    <!-- création d'un bean javax.sql.DataSource en utilisant
        des propriétés chargées depuis le fichier conf/db-ticket.properties -->
    <bean id="dataSourceTicket" destroy-method="close"
        class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName" value="${database.ticket.driverClassName}"/>
        <property name="url" value="${database.ticket.url}"/>
        <property name="username" value="${database.ticket.username}"/>
        <property name="password" value="${database.ticket.password}"/>
    </bean>

    <!-- injection de la javax.sql.DataSource "dataSourceTicket"
        dans le bean parent des DAO -->
    <bean id="abstractDao" abstract="true">
        <property name="dataSourceTicket" ref="dataSourceTicket" />
    </bean>

    <bean id="projetDao" class="org.example.demo.ticket.consumer.impl.dao.ProjetDaoImpl"
        parent="abstractDao" />
    <bean id="ticketDao" class="org.example.demo.ticket.consumer.impl.dao.TicketDaoImpl"
        parent="abstractDao" />
</beans>
```



Injecter une ressource JNDI

```
<beans>
    <!-- Récupération de la ressource JNDI :
        javax.sql.DataSource pour la base de données DB_TICKET -->
    <bean id="dataSourceTicket" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="java:comp/env/jdbc/DB_TICKET" />
    </bean>

    <!-- Injection de la "dataSourceTicket" dans le bean parent des DAO -->
    <bean id="abstractDao" abstract="true">
        <property name="dataSourceTicket" ref="dataSourceTicket" />
    </bean>

    <bean id="projetDao" class="org.example.demo.ticket.consumer.impl.dao.ProjetDaoImpl"
          parent="abstractDao" />
    <bean id="ticketDao" class="org.example.demo.ticket.consumer.impl.dao.TicketDaoImpl"
          parent="abstractDao" />
</beans>
```

Bien évidemment, les exemples que je vous ai montrés ci-dessus sont loin d'être exhaustifs. Je vous laisse le soin de parcourir la [documentation officielle](#) pour découvrir encore plus de possibilités.

Conclusion

Voilà, vous arrivez à la fin de cette première partie. Ouf, ça fait plaisir ! En plus, votre ventre gargouille et vous avez une folle envie de tartiflette...

Vous avez vu, ou revu, les principes **SOLID** :

- Single responsibility principle (responsabilité unique)
- Open/closed principle (ouvert/fermé)
- Liskov substitution principle (substitution de Liskov)
- Interface segregation principle (ségrégation des interfaces)
- Dependency inversion principle (inversion des dépendances)

De ces principes découlent des patrons de conception et des approches tels que :

- l'**inversion de contrôle** : le flot d'exécution n'est plus totalement géré par le développeur ou les traitements de l'application, mais il est pris en charge par des éléments annexes.
- le design pattern **Factory** : une classe est responsable de la fabrication/fourniture d'instance d'autres classes.
- l'**injection de dépendances** : les objets dépendant d'autres objets ne se soucient plus de comment les obtenir, ils leur sont directement injectés.
- les **abstractions** : très souvent réalisées à l'aide d'interfaces, ces abstractions sont les « contrats » des interactions entre les différentes couches de l'application. Les couches basses implémentent ces abstractions. Les couches hautes dépendent de ces abstractions et non des implémentations faites dans les couches basses.

Après avoir créé un système d'injection de dépendances rustique, nous sommes passés à un niveau supérieur avec Spring. Vous avez pu constater qu'en termes de configuration,



Spring vous laisse le choix (fichiers XML, annotations Spring/JSR-250/JSR-330...). Je vous ai donné quelques conseils pour mieux utiliser Spring et vous organiser.

Dans le chapitre suivant, je vous montrerai comment Spring peut vous faciliter la vie lors de vos interactions avec une base de données. Vous exécuterez toujours des requêtes SQL, mais sans avoir à tout faire à la main en JDBC !!

Implémenter des DAO

Lorsque vous interagissez avec une base de données, vous utilisez souvent le design pattern DAO (*Data Access Objet*) et l'API JDBC (*Java Database Connectivity*).

Dans ce chapitre, je vais vous montrer comment créer vos DAO en respectant les principes *SOLID* et comment les configurer avec *Spring IoC*.

Création et configuration des DAO

Les DAO sont à créer dans le module *ticket-consumer* qui, je vous le rappelle, est responsable des relations avec les services extérieurs à l'application, notamment la base de données.

Pour les classes DAO, je vais suivre le même principe que pour les *Managers* : je crée une classe DAO par sous-packages de beans métier.

Bien entendu, il ne faut pas oublier l'*inversion de dépendances* et les *abstractions* donc, chaque classe DAO (suffixée par *DaoImpl*) implémente une interface DAO (suffixée par *Dao*).

```
package org.example.demo.ticket.consumer.contract.dao;

/**
 * Interface DAO du package
 * {@link org.example.demo.ticket.model.bean.ticket}
 */
public interface TicketDao {
    // ...
}
```

```
package org.example.demo.ticket.consumer.impl.dao;

import org.example.demo.ticket.consumer.contract.dao.TicketDao;

/**
 * Classe d'implémentation de {@link TicketDao}.
 */
@Named
public class TicketDaoImpl extends AbstractDaoImpl implements TicketDao {
    // ...
}
```

Afin de généraliser certains éléments, toutes les classes DAO vont hériter de la classe *AbstractDaolmpl* qui aura, entre autres, un attribut *dataSource* permettant la connexion à la base de données de l'application.

```
package org.example.demo.ticket.consumer.impl.dao;

import javax.sql.DataSource;

public abstract class AbstractDaoImpl {

    private DataSource dataSource;

    protected DataSource getDataSource() {
        return dataSource;
    }

    // ...
}
```

Configuration des DAO

Je veux que la DataSource soit injectée dans l'attribut dataSource des classes d'implémentation des DAO.

Pour faire cela facilement, je dis à Spring d'injecter le bean dataSourceTicket via les annotations `@Inject` et `@Named("dataSourceTicket")` directement sur l'attribut dataSource dans la classe `AbstractDaoImpl`:

```
package org.example.demo.ticket.consumer.impl.dao;

import javax.inject.Inject;
import javax.inject.Named;
import javax.sql.DataSource;

public abstract class AbstractDaoImpl {

    @Inject
    @Named("dataSourceTicket")
    private DataSource dataSource;

    // ...
}
```

Configuration de la DataSource

Il ne reste plus qu'à créer la DataSource d'accès à la base de données DB_TICKET que Spring doit injecter (bean `dataSourceTicket`).

Drivers JDBC

Afin de se connecter à une base de données avec JDBC, les drivers JDBC correspondant au SGBD (*Système de Gestion de Base de Données*) doivent être accessibles par le *classloader* lors de la création de la *DataSource*.

Dans le fichier pom.xml racine, j'ajoute :



```
<project>
  ...
  <dependencyManagement>
    <dependencies>
      <!-- Drivers JDBC PostgreSQL -->
      <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>9.4.1212</version>
        <scope>runtime</scope>
      </dependency>
      ...
    </dependencies>
  </dependencyManagement>
</project>
```

Remarquez ici l'utilisation du scope runtime : les drivers ne sont pas utiles en phase de compilation mais seulement au runtime.

Module ticket-batch

Pour les batches, je vais gérer moi-même la création de la DataSource. Je dois donc ajouter les drivers JDBC de *PostgreSQL* dans les dépendances Maven du module *ticket-batch*.

Dans le fichier ticket-batch/pom.xml, j'ajoute :

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
    </dependency>
    ...
  </dependencies>
</project>
```

Module ticket-webapp

Pour l'application web, en revanche, il est possible :

- soit d'ajouter le JAR des drivers JDBC au *classpath* de *Tomcat* (typiquement dans \$CATALINA_HOME/lib) ;
- soit de l'embarquer directement dans le WAR (dans WEB-INF/lib).

C'est ce que je vais faire ici : embarquer directement les drivers dans le WAR. J'ajoute donc la dépendance Maven dans le fichier ticket-webapp/pom.xml :

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
    </dependency>
    ...
  </dependencies>
</project>
```

Configuration de l'application web

Le serveur [Apache Tomcat](#), comme la majorité des serveurs d'application, permet de créer des ressources JNDI de DataSource s'appuyant sur un pool de connexion.

Ces ressources peuvent être configurées :

- soit au niveau du serveur Tomcat,
- soit directement dans le WAR de déploiement de la webapp (cf [documentation](#)).

Je vais utiliser la dernière solution ici, qui est plus facile à mettre en oeuvre dans cette démonstration (je n'ai pas besoin de rentrer le détail de la configuration de Tomcat).

Généralement, en tant que développeur, vous ne connaîtrez pas les éléments de configuration de la production (URL du serveur, identifiants, mots de passe...). Vous ne pourrez donc pas configurer les ressources directement dans le WAR destiné à la mise en production. Les ressources devront alors être configurées directement dans Tomcat par les administrateurs.

Pour configurer les ressources directement dans le WAR, il faut ajouter un fichier /META-INF/context.xml dans celui-ci.

Afin de l'embarquer facilement lors du build Maven, je crée le fichier src/main/webapp/META-INF/context.xml dans le module *ticket-webapp* :



```

<?xml version="1.0" encoding="UTF-8"?>
<Context>
    <Resource name="jdbc/DB_TICKET"
              auth="Container"
              type="javax.sql.DataSource"

              url="jdbc:postgresql://localhost:5432/db_ticket"
              driverClassName="org.postgresql.Driver"
              username="ticket"
              password="ticket"
              defaultAutoCommit="false"
              defaultTransactionIsolation="READ_COMMITTED"

              initialSize="1"
              maxTotal="30"
              maxIdle="10"
              maxWaitMillis="60000"
              minIdle="1"
              removeAbandonedTimeout="60"
              removeAbandonedOnBorrow="true"
              logAbandoned="true"
              minEvictableIdleTimeMillis="10000"
              timeBetweenEvictionRunsMillis="30000"
              validationQuery="SELECT 1"
              testWhileIdle="true"
              testOnBorrow="true"
    />
</Context>

```

Chaque ressource JNDI à créer fait l'objet d'un élément `<Resource>` dans l'élément racine `<Context>`. J'ai formaté ici l'élément `<Resource>` en 3 blocs d'attributs :

- le premier définit la nature et le nom de la ressource JNDI ;
- le deuxième définit les paramètres de connexion à la base de données ;
- le troisième définit les [paramètres de configuration](#) du pool de connexion ([Apache Commons DBCP](#)).

La ressource JNDI est maintenant configurée dans Tomcat, il ne reste plus qu'à utiliser cette ressource JNDI dans le module *ticket-webapp* :

1. Je commence par indiquer dans le fichier `src/main/webapp/WEB-INF/web.xml` que j'ai besoin de la ressource JNDI `jdbc/DB_TICKET` :

```

<web-app>
    <resource-ref>
        <res-ref-name>jdbc/DB_TICKET</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
    ...
</web-app>

```

2. Puis, j'ajoute cette ressource en tant que bean dataSourceTicket dans l'IoC container de Spring. J'ajoute dans le fichier src/main/resources/.../webappContext.xml :

```
<bean id="dataSourceTicket" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/DB_TICKET"/>
</bean>
```

Je vous fais remarquer ici deux points particuliers :

- le nom des ressources JNDI est préfixé par java:comp/env/ ;
- les beans correspondant à des ressources JNDI sont créés dans le *context* Spring via une *factory* spécifique : org.springframework.jndi.JndiObjectFactoryBean

Configuration des batches

En ce qui concerne les batches, il n'y a pas de serveur d'application pour gérer les ressources JNDI. Je vais embarquer et gérer un pool de connexion directement dans l'application.

Je vais utiliser le pool de connexion [Apache Commons DBCP](#). J'ajoute donc la dépendance Maven :

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.1.1</version>
</dependency>
```

Afin de configurer le pool de connexion, je vais utiliser un fichier *properties*.

Je veux que ce fichier soit modifiable par les administrateurs (cf section précédente). Il sera donc externalisé et non embarqué dans le JAR des batches.

1. Je crée un répertoire src/data/conf dans le module *ticket-batch* contenant un exemple des fichiers de configuration.
2. Lors du build Maven, le plugin maven-assembly-plugin crée une archive avec tous les JAR nécessaires et ces fichiers de configuration.
3. L'administrateur décomprime l'archive sur le serveur de production et modifie les valeurs dans les fichiers de configuration.

Je crée donc le fichier src/data/conf/db-ticket.properties contenant les propriétés de configuration du pool de connexion à la base DB_TICKET :

```

url=jdbc:postgresql://localhost:5432/db_ticket
driverClassName=org.postgresql.Driver
username=ticket
password=ticket
defaultAutoCommit=false
defaultTransactionIsolation=READ_COMMITTED

initialSize=1
maxTotal=30
maxIdle=10
maxWaitMillis=60000
minIdle=1
removeAbandonedTimeout=60
removeAbandonedOnBorrow=true
logAbandoned=true
minEvictableIdleTimeMillis=10000
timeBetweenEvictionRunsMillis=30000
validationQuery=SELECT 1
testWhileIdle=true
testOnBorrow=true

```

Afin de faciliter l'adressage des fichiers de l'application, mes scripts de lancement des batches assignent la propriété système `application.home` indiquant la racine de l'application (le répertoire où a été décompressée l'archive de déploiement).

Lors du lancement des batches via votre IDE, il faut donc aussi assigner cette propriété, généralement via une option de VM : `-Dapplication.home=..."` (le `-D` collé devant `application.home` n'est pas une erreur !)

Dans le fichier `src/main/resources/.../batchContext.xml` du module `ticket-batch`, j'importe le fichier `properties` de configuration de la base de données et je crée la datasource `dataSourceTicket` :

```

<beans>
    <!-- Chargement du fichier properties contenant
        la configuration de la datasource vers DB_TICKET -->
    <bean id="dataSourceTicketConfiguration"
        class="org.springframework.beans.factory.config.PropertiesFactoryBean">
        <property name="location" value="file:${application.home}/conf/db-ticket.properties"/>
    </bean>

    <!-- Crédation de la datasource "dataSourceTicket" -->
    <bean id="dataSourceTicket"
        class="org.apache.commons.dbcp2.BasicDataSourceFactory"
        factory-method="createDataSource"
        destroy-method="close">
        <constructor-arg ref="dataSourceTicketConfiguration"/>
    </bean>
</beans>

```

Dans cette configuration Spring, je crée :

- une instance de `java.util.Properties` (bean `dataSourceTicketConfiguration`) à partir du fichier `properties file:${application.home}/conf/db-ticket.properties` ;
- une instance de `org.apache.commons.dbcp2.BasicDataSource` (bean `dataSourceTicket`).

Le bean `dataSourceTicket` est créé :

- grâce à la méthode static `createDataSource` (attribut *factory-method*),
- de la classe `org.apache.commons.dbcp2.BasicDataSourceFactory` (attribut *class*),
- en lui passant en paramètre le bean `dataSourceTicketConfiguration` (élément *constructor-arg*).

L'attribut *destroy-method* indique que la méthode `close` devra être appelée sur ce bean lorsqu'il sera détruit (typiquement à l'arrêt de l'application).

Ça y est, tout est prêt ! Cela vous a peut-être semblé un peu long, mais je tenais à bien vous expliquer chaque étape. Ne vous inquiétez pas, en réalité (et d'autant plus avec l'habitude), ce travail préparatoire ne prend que quelques minutes.

Dans le chapitre suivant, je vais passer aux choses sérieuses : exécuter des requêtes SQL... mais en utilisant *Spring JDBC* !



Simplifier l'exécution de requêtes SQL avec Spring JDBC

Lorsqu'il s'agit d'interagir avec une base de données, l'exécution de requêtes SQL via JDBC n'est pas difficile. Mais il s'agit d'un travail assez fastidieux : il faut, entre autres, ouvrir une connexion, créer un *Statement*, exécuter la requête, boucler sur le *ResultSet* pour charger les résultats et enfin tout fermer proprement (même en cas d'erreur).

Spring peut alléger le travail du développeur en encapsulant tout le code de « tuyauterie », le développeur n'ayant plus qu'à implémenter les parties spécifiques :

Action	Spring	Développeur
Définir la <i>DataSource</i>		✓
Ouvrir la connexion	✓	
Spécifier la requête SQL		✓
Déclarer les paramètres et fournir leur valeur		✓
Préparer et exécuter la requête (<i>Statement</i>)	✓	
Boucler, si besoin, sur les résultats (<i>ResultSet</i>)	✓	
Faire ce qu'il y a à faire à chaque itération		✓
Fermer la connexion, le <i>Statement</i> et le <i>ResultSet</i>	✓	
Prendre en charge les transactions	✓	
Traiter les exceptions	✓	

Travail préparatoire

Ajout de la dépendance vers Spring JDBC

Le module Spring à utiliser pour l'exécution des requêtes SQL est *Spring JDBC*. Il faut donc ajouter cette dépendance dans le projet Maven (module *ticket-consumer*) :



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

Et c'est tout !

Requête SQL de sélection

Définir et exécuter une requête SQL de sélection

Commençons par un exemple très simple : récupérer le nombre de tickets.

```
package org.example.demo.ticket.consumer.impl.dao;

import javax.inject.Named;
import org.example.demo.ticket.consumer.contract.dao.TicketDao;
import org.example.demo.ticket.model.recherche.ticket.RechercheTicket;
import org.springframework.jdbc.core.JdbcTemplate;

@Named
public class TicketDaoImpl extends AbstractDaoImpl implements TicketDao {

    @Override
    public int getCountTicket(RechercheTicket pRechercheTicket) {
        JdbcTemplate vJdbcTemplate = new JdbcTemplate(getDataSource());
        int vNbrTicket = vJdbcTemplate.queryForObject(
            "SELECT COUNT(*) FROM ticket",
            Integer.class);

        return vNbrTicket;
    }
}
```

Comparer à ce que vous devriez coder avec JDBC, avouez qu'avec Spring JDBC, c'est bien plus simple et rapide !

Je viens de vous montrer un premier élément de Spring JDBC : la classe JdbcTemplate. C'est grâce à cette classe, créée à partir d'une DataSource, que je définis et exécute une requête SQL.

Spécifier des paramètres

Ajoutons maintenant quelques paramètres : récupérer le nombre de tickets correspondant aux critères de recherche.

Les paramètres sont désignés par des ?, comme en JDBC classique.



```

public class TicketDaoImpl extends AbstractDaoImpl implements TicketDao {

    @Override
    public int getCountTicket(RechercheTicket pRechercheTicket) {
        String vSQL
            = "SELECT COUNT(*) FROM ticket"
            + " WHERE auteur_id = ?"
            + " AND projet_id = ?";

        JdbcTemplate vJdbcTemplate = new JdbcTemplate(getDataSource());
        int vNbrTickets = vJdbcTemplate.queryForObject(
            vSQL, Integer.class,
            pRechercheTicket.getAuteurId(),
            pRechercheTicket.getProjetId());

        return vNbrTicket;
    }
}

```

Les plus attentifs auront remarqué quelques problèmes : pas de test de nullité de pRechercheTicket, tous les critères de recherches sont appliqués même s'ils ne sont pas renseignés...

Je l'ai fait exprès, pour vous montrer l'essentiel (passer des paramètres), je reviendrai dessus plus tard.

Spring permet également d'utiliser des paramètres nommés (de la forme :param) grâce à la classe NamedParameterJdbcTemplate (utilisée à la place de JdbcTemplate) :

```

public class TicketDaoImpl extends AbstractDaoImpl implements TicketDao {

    @Override
    public int getCountTicket(RechercheTicket pRechercheTicket) {
        String vSQL
            = "SELECT COUNT(*) FROM ticket"
            + " WHERE auteur_id = :auteur_id"
            + " AND projet_id = :projet_id";

        NamedParameterJdbcTemplate vJdbcTemplate = new NamedParameterJdbcTemplate(getDataSource());

        MapSqlParameterSource vParams = new MapSqlParameterSource();
        vParams.addValue("auteur_id", pRechercheTicket.getAuteurId());
        vParams.addValue("projet_id", pRechercheTicket.getProjetId());

        int vNbrTicket = vJdbcTemplate.queryForObject(vSQL, vParams, Integer.class);

        return vNbrTicket;
    }
}

```

Avant de passer à la suite, à titre d'exercice, vous pouvez améliorer le code de cette méthode pour prendre en compte la nullité de pRechercheTicket, les critères non renseignés...

Je vous montre cela juste en dessous...



```

public class TicketDaoImpl extends AbstractDaoImpl implements TicketDao {

    @Override
    public int getCountTicket(RechercheTicket pRechercheTicket) {
        MapSqlParameterSource vParams = new MapSqlParameterSource();

        StringBuilder vSQL = new StringBuilder("SELECT COUNT(*) FROM ticket WHERE 1=1");

        if (pRechercheTicket != null) {
            if (pRechercheTicket.getAuteurId() != null) {
                vSQL.append(" AND auteur_id = :auteur_id");
                vParams.addValue("auteur_id", pRechercheTicket.getAuteurId());
            }
            if (pRechercheTicket.getProjetId() != null) {
                vSQL.append(" AND projet_id = :projet_id");
                vParams.addValue("projet_id", pRechercheTicket.getProjetId());
            }
        }

        NamedParameterJdbcTemplate vJdbcTemplate = new NamedParameterJdbcTemplate(getDataSource());
        int vNbrTicket = vJdbcTemplate.queryForObject(vSQL.toString(), vParams, Integer.class);

        return vNbrTicket;
    }
}

```

Je reviendrai sur la spécification des paramètres dans la section traitant des requêtes de mise à jour.

Récupérer le résultat d'une requête

Jusqu'à présent j'ai abordé des exemples très simples où la requête ne retourne qu'une seule ligne avec une seule colonne. Spring permet de gérer très simplement ces cas en revoyant directement un objet Java correspondant à la valeur renournée par la requête SQL.

Abordons maintenant un cas plus intéressant : la requête renvoie plusieurs lignes et colonnes et je veux obtenir une liste d'objets.

Si vous reprenez les tableaux de répartition des responsabilités entre le développeur et Spring, vous voyez que le développeur n'a à coder que le travail à faire sur chaque ligne de résultat. C'est-à-dire qu'il doit simplement coder le mapping entre une ligne de résultat et l'objet Java correspondant.

Pour ce faire, le développeur doit implémenter une interface fournie par Spring : RowMapper<T>. Le nom est parlant : il s'agit de mapper une ligne de résultat en objet de type T.

Commençons pas un cas simple : charger tous les statuts de ticket depuis la base de données.

```

package org.example.demo.ticket.consumer.impl.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import javax.inject.Named;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import org.example.demo.ticket.consumer.contract.dao.TicketDao;
import org.example.demo.ticket.model.bean.ticket.TicketStatut;

@Named
public class TicketDaoImpl extends AbstractDaoImpl implements TicketDao {
    // ...

    @Override
    public List<TicketStatut> getListStatut() {
        String vSQL = "SELECT * FROM public.statut";

        JdbcTemplate vJdbcTemplate = new JdbcTemplate(getDataSource());

        RowMapper<TicketStatut> vRowMapper = new RowMapper<TicketStatut>() {
            public TicketStatut mapRow(ResultSet pRS, int pRowNum) throws SQLException {
                TicketStatut vTicketStatut = new TicketStatut(pRS.getInt("id"));
                vTicketStatut.setLibelle(pRS.getString("libelle"));
                return vTicketstatut;
            }
        };

        List<TicketStatut> vListStatut = vJdbcTemplate.query(vSQL, vRowMapper);

        return vListStatut;
    }
}

```

Afin de clarifier le code et mutualiser les *RowMappers*, je vous conseille de créer des classes de *RowMapper* dédiées pour chaque objet métier (en respectant le découpage en package) :

- org.example.demo.ticket.consumer.impl.rowmapper
 - projet
 - ProjetRM
 - ...
 - ticket
 - TicketRM
 - TicketStatutRM
 - ...
 - ...
 - ...

Requête SQL de mise à jour

Pour les requêtes de mise à jour des données (INSERT, UPDATE...), cela se passe comme pour les requêtes de sélection sauf qu'il n'y a plus de données à charger (donc pas de *RowMapper*) et qu'il faut utiliser les méthodes update au lieu des méthodes query*.

```
public void updateTicketStatut(TicketStatut pTicketStatut) {  
    String vSQL = "UPDATE statut SET libelle = :libelle WHERE id = :id";  
  
    MapSqlParameterSource vParams = new MapSqlParameterSource();  
    vParams.addValue("id", pTicketStatut.getId());  
    vParams.addValue("libelle", pTicketStatut.getLibelle());  
  
    NamedParameterJdbcTemplate vJdbcTemplate = new NamedParameterJdbcTemplate(getDataSource());  
  
    int vNbrLigneMaj = vJdbcTemplate.update(vSQL, vParams);  
}
```

Pour information, les méthodes `JdbcTemplate.update(...)` renvoient le nombre de lignes affectées par la requête.

Passer des paramètres

Utiliser les attributs d'un JavaBean

Vous pouvez encore alléger votre code en passant directement un JavaBean en paramètre, et non plus chaque paramètre individuellement !

Pour cela, vous devez utiliser :

- des paramètres nommés (:param) dans votre requête SQL : le nom des paramètres correspond aux attributs du JavaBean ;
- la classe `NamedParameterJdbcTemplate` pour exécuter la requête
- la classe `BeanPropertySqlParameterSource` comme source des paramètres en lui passant le JavaBean

Voici ce que cela donne si je reprends la requête précédente :



```

public void updateTicketStatut(TicketStatut pTicketStatut) {
    String vSQL = "UPDATE statut SET libelle = :libelle WHERE id = :id";
    SqlParameterSource vParams = new BeanPropertySqlParameterSource(pTicketStatut);
    NamedParameterJdbcTemplate vJdbcTemplate = new NamedParameterJdbcTemplate(getDataSource());
    int vNbrLigneMaj = vJdbcTemplate.update(vSQL, vParams);
}

```

```

package org.example.demo.ticket.model.bean.ticket;

public class TicketStatut {
    private Integer id;
    private String libelle;

    public Integer getId() {
        return id;
    }
    public String getLibelle() {
        return libelle;
    }
    //...
}

```

Spécifier les types de données

Spring définit les types de données à injecter dans la requête SQL à partir des classes Java utilisées comme valeur de paramètre.

Dans certains cas, cette auto-détermination n'est pas correcte (paramètre à null par exemple) et nécessite alors de préciser le type attendu par la base de données.

Vous pouvez préciser ce type :

- soit en l'indiquant lors de l'ajout du paramètre dans la liste des paramètres ;
- soit en spécifiant le type des paramètres pour un BeanPropertySqlParameterSource ;
- soit en utilisant la classe SqlParameterValue comme wrapper de votre paramètre ;
- soit lors de l'exécution de la requête.

Je vous conseille d'utiliser l'une des 3 premières solutions, qui sont bien plus lisibles que la dernière.

```

package org.example.demo.ticket.consumer.impl.dao;

import java.sql.Types;
// import ...

public class TicketDaoImpl extends AbstractDaoImpl implements TicketDao {
    /**
     * ...
     */
    public void updateTicketStatut1(TicketStatut pTicketStatut) {
        String vSQL = "UPDATE statut SET libelle = :libelle WHERE id = :id";
        MapSqlParameterSource vParams = new MapSqlParameterSource();
        vParams.addValue("id", pTicketStatut.getId(), Types.INTEGER);
        vParams.addValue("libelle", pTicketStatut.getLibelle(), Types.VARCHAR);

        NamedParameterJdbcTemplate vJdbcTemplate = new NamedParameterJdbcTemplate(getDataSource());
        int nbrLigneMod = vJdbcTemplate.update(vSQL, vParams);
    }

    public void updateTicketStatut2(TicketStatut pTicketStatut) {
        String vSQL = "UPDATE statut SET libelle = :libelle WHERE id = :id";
        BeanPropertySqlParameterSource vParams = new BeanPropertySqlParameterSource(pTicketStatut);
        vParams.registerSqlType("id", Types.INTEGER);
        vParams.registerSqlType("libelle", Types.VARCHAR);

        NamedParameterJdbcTemplate
            vJdbcTemplate = new NamedParameterJdbcTemplate(getDataSource());
        int nbrLigneMod = vJdbcTemplate.update(vSQL, vParams);
    }

    public void updateTicketStatut3(TicketStatut pTicketStatut) {
        String vSQL = "UPDATE statut SET libelle = ? WHERE id = ?";
        Object[] vParams = {
            new SqlParameterValue(Types.INTEGER, pTicketStatut.getId()),
            new SqlParameterValue(Types.VARCHAR, pTicketStatut.getLibelle())
        };

        JdbcTemplate vJdbcTemplate = new JdbcTemplate(getDataSource());
        vJdbcTemplate.update(vSQL, vParams);
    }

    public void updateTicketStatut4(TicketStatut pTicketStatut) {
        String vSQL = "UPDATE statut SET libelle = ? WHERE id = ?";
        Object[] vParams = {
            pTicketStatut.getId(),
            pTicketStatut.getLibelle()
        };

        JdbcTemplate vJdbcTemplate = new JdbcTemplate(getDataSource());
        vJdbcTemplate.update(vSQL,
            vParams,
            new int[] {
                Types.INTEGER,
                Types.VARCHAR
            });
    }
}

```



Gérer les exceptions

Vous avez peut-être remarqué, mais jusqu'à présent, je ne me suis pas soucié des Exceptions pouvant survenir lors de l'exécution des requêtes SQL.

Et pour cause, Spring gère directement les SQLException et les traduits en RuntimeException. J'en ai vu qui ont bondi de leur chaise en lisant cela !

Spring fait mieux que ça, il traduit les SQLException en s'appuyant sur les codes de retour SQL pour lever des *Exceptions* distinctes (DataIntegrityViolationException, DuplicateKeyException, EmptyResultDataAccessException...). Toutes ces Exceptions héritent de la classe DataAccessException.

Et comme ce sont des RuntimeException, leur déclaration et leur gestion n'est pas obligatoire lors de la compilation. Ceci réduit considérablement les besoins en try...catch ! Cependant, vous pouvez toujours faire un *catch* de certaines de ces exceptions au cas par cas.

```
public void insertTicketStatut(Ticketstatut pTicketstatut) {
    String vSQL = "INSERT INTO statut (id, libelle) VALUES (:id, :libelle)";
    NamedParameterJdbcTemplate vJdbcTemplate = new NamedParameterJdbcTemplate(getDataSource());

    BeanPropertySqlParameterSource vParams = new BeanPropertySqlParameterSource(pTicketstatut);
    vParams.registerSqlType("id", Types.INTEGER);
    vParams.registerSqlType("libelle", Types.VARCHAR);

    try {
        vJdbcTemplate.update(vSQL, vParams);
    } catch (DuplicateKeyException vEx) {
        LOGGER.error("Le TicketStatut existe déjà ! id=" + pTicketstatut.getId(), vEx);
        // ...
    }
}
```

Conclusion

Comme vous avez pu le constater, *Spring JDBC* simplifie grandement le travail du développeur lorsqu'il doit interagir avec une base de données relationnelle à l'aide de requêtes SQL.

Dans ce chapitre, je n'ai pas abordé toutes les fonctionnalités et les détails techniques de *Spring JDBC*. Je vous ai présenté ce que je considère comme la base et le principal pour l'utiliser.

Si vous souhaitez aller plus loin, n'hésitez pas à consulter la documentation officielle, notamment les parties :

- [DAO Support](#)
- [Data access with JDBC](#)

Vous y trouverez également des conseils ou bonnes pratiques préconisées par les développeurs de Spring. Par exemple, sachez que la création des *JdbcTemplate* peut être faite à un niveau global et non par requête car cette classe est *threadsafe* une fois configurée (cf [JdbcTemplate best practices](#)).



Je vous conseille de vous entraîner en créant les DAO manquants du projet *ticket* et en y implémentant quelques méthodes.

Dans le chapitre suivant, je vous montrerai comment gérer les transactions dans un projet grâce à *Spring TX*.



Gérer les transactions avec Spring TX

Je vous disais dans l'introduction que Spring était un framework modulaire. Et ce qui est génial c'est que les modules fonctionnent très bien ensemble.

Dans le chapitre précédent nous avons vu comment exécuter des requêtes SQL avec *Spring JDBC*.

Eh bien, sachez que *Spring JDBC* peut fonctionner dans un contexte transactionnel, lui aussi géré avec Spring ! C'est le rôle du module *Spring TX*.

Spring TX permet de gérer le contexte transactionnel de différentes API comme JDBC, Java Transaction API (JTA), Java Persistence API (JPA)...

Je vais me concentrer uniquement sur le contexte JDBC ici et faire le lien avec ce que nous avons vu dans le chapitre précédent. Si vous voulez en savoir plus sur les autres types de contexte, je vous renvoie vers la [documentation officielle](#).

Gérer le contexte transactionnel

Vous commencez à avoir l'habitude que Spring vous laisse le choix... Avec Spring TX, vous pouvez gérer plusieurs types de contexte transactionnel (JDBC, JTA...), mais en plus vous avez plusieurs manières de les gérer !

On distingue deux principales approches :

- la manière [déclarative](#) ;
- par [programmation](#).

Dans ce cours nous verrons **uniquement l'approche par programmation**.

En effet, la manière déclarative impose de faire de l'AOP ([Programmation Orientée Aspect](#)). Spring fournit un module pour cela : *Spring AOP* (cf [documentation](#)). Mais...

Je vous le dis en toute franchise : **personnellement**, je ne suis pas fan de ce module et de la mise en œuvre de l'AOP avec Spring ! Je ne vais pas épiloguer pendant des heures, mais en voici rapidement quelques raisons :

- Avec Spring, tout l'AOP se fait au runtime (création des aspects, tissage...). C'est souple, mais pas très efficace en termes de chargement de l'application (bootstrap), voire tout au long de l'exécution !
- Le module Spring AOP utilise par défaut les *dynamic proxies* du JDK. Là encore, cela peut vous poser des problèmes car le niveau d'imbrication des classes au runtime peut vite grossir et devient une horreur lors du debug en pas à pas de votre application ! (Spring peut aussi utiliser CGLIB pour gérer les aspects : cela modifie le byte code Java au runtime...)

Encore une fois, **personnellement**, je préfère me tourner vers d'autres solutions pour l'AOP, comme [AspectJ](#). Mais cela dépasse largement le cadre de ce cours...

Bref, ça existe, je ne vous empêche pas de tester pour vous faire votre propre avis, bien au contraire. Mais ce sera sans moi !



Bon, après ce petit aparté, passons aux choses sérieuses...
direction la couche... **business**.

Euh qui a dit *consumer* ?

Si vous vous demandez pourquoi nous allons dans le module *ticket-business* et non *ticket-consumer*, en voici l'explication.

Le rôle de la couche *consumer* est d'interagir avec les services externes. Cela peut être l'exécution d'une requête SQL ou l'appel à un webservice. Cependant, ce n'est pas à cette couche de prendre des décisions sur l'atomicité et la cohérence d'une action métier dans l'application. Ça, c'est le rôle de la couche *business* !

Un petit exemple : le cas d'utilisation « clôturer un ticket » implique plusieurs choses :

1. Passer le statut du ticket à « clôturer ».
2. Ajouter un ligne d'historique de statut du ticket.
3. Associer à cette ligne d'historique un éventuel commentaire de l'utilisateur.

Or il s'agit d'une action métier atomique : soit toutes les actions sont réalisées avec succès, soit une erreur se produit et aucune ne doit l'être.

Si vous n'êtes pas tout à fait convaincu·e, je vous laisse réfléchir à l'atomicité d'une action métier comme : *annuler un billet de train...*

OK, maintenant que tout est clair, vous pouvez ajouter la dépendance vers *Spring-TX* dans le module *ticket-business* :

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

I imagine that right now you're feeling a bit like Alice tumbling down the rabbit hole...

Vous avez deux solutions : la pilule bleue ou la pilule rouge...

La pilule bleue

Vous avez goûté au confort et à la simplicité avec la classe *JdbcTemplate*. Eh bien ça continue avec la classe *TransactionTemplate* !



Utiliser un TransactionTemplate

```
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallbackWithoutResult;
import org.springframework.transaction.support.TransactionTemplate;
// import ...

public class TicketManagerImpl extends AbstractManagerImpl implements TicketManager {

    @Inject
    @Named("txManagerTicket")
    private PlatformTransactionManager platformTransactionManager;

    @Override
    public void changerStatut(Ticket pTicket, TicketStatut pNewStatut,
                              Utilisateur pUtilisateur, Commentaire pCommentaire) {
        TransactionTemplate vTransactionTemplate
            = new TransactionTemplate(platformTransactionManager);

        vTransactionTemplate.execute(new TransactionCallbackWithoutResult() {
            @Override
            protected void doInTransactionWithoutResult(TransactionStatus
                pTransactionStatus) {
                pTicket.setStatut(pNewStatut);
                getDaoFactory().getTicketDao().updateTicket(pTicket);
                // TODO Ajout de la ligne d'historique + commentaire ...
            }
        });
    }
}
```

Il ne reste plus qu'à définir le bean txManagerTicket de type PlatformTransactionManager à injecter via Spring IoC.

Définir le PlatformTransactionManager

PlatformTransactionManager est une interface (souvenez-vous, Spring TX permet de gérer différentes natures de transaction). Ici, j'ai besoin de gérer des transactions sur une *DataSource JDBC*. Je vais donc utiliser la classe DataSourceTransactionManager. J'ajoute la définition du bean dans le fichier businessContext.xml :

```
<bean id="txManagerTicket" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSourceTicket"/>
</bean>
```

Silence, moteur... action !

Et c'est fini. Spring fait le reste ! En effet, lors de l'appel de la méthode vTransactionTemplate.execute(...) Spring va :

1. ouvrir la transaction ;



2. encapsuler dans un try...catch l'appel de la méthode doInTransactionWithoutResult(...) définie dans la classe anonyme de type TransactionCallbackWithoutResult ;
3. valider la transaction après le bloc try...catch ;
4. si une erreur survient, Spring va annuler la transaction dans le catch et :
 - o propager l'exception s'il s'agit d'une RuntimeException ou d'une Error
 - o lever une UndeclaredThrowableException avec le Throwable en cause dans les autres cas.

Demander vous-même l'annulation

En fonction de votre implémentation et des règles métier, vous aurez peut-être besoin de demander vous-même l'annulation de la transaction. Cela se fait en appelant la méthode setRollbackOnly() sur le TransactionStatus :

```
vTransactionTemplate.execute(new TransactionCallbackWithoutResult() {
    @Override
    protected void doInTransactionWithoutResult(TransactionStatus
                                                 pTransactionStatus) {
        TicketStatut vOldStatut = pTicket.getStatut();
        pTicket.setStatut(pNewStatut);
        try {
            getDaoFactory().getTicketDao().updateTicket(pTicket);
            // TODO Ajout de la ligne d'historique + commentaire ...
        } catch (TechnicalException vEx) {
            pTransactionStatus.setRollbackOnly();
            pTicket.setStatut(vOldStatut);
        }
    }
});
```

Renvoyer un objet en sortie

Si votre traitement transactionnel doit renvoyer un objet, il faut alors utiliser la classe TransactionCallback :

```
HistoriqueStatut vHistorique =
    vTransactionTemplate.execute(new TransactionCallback<HistoriqueStatut>() {
        @Override
        public HistoriqueStatut doInTransaction(TransactionStatus
                                                pTransactionStatus) {
            pTicket.setStatut(pNewStatut);
            getDaoFactory().getTicketDao().updateTicket(pTicket);

            HistoriqueStatut vHistoriqueStatut = new HistoriqueStatut();
            // TODO Ajout de la ligne d'historique + commentaire ...
            return vHistoriqueStatut;
        }
    });
});
```

Partager le TransactionTemplate

Enfin, sachez que, comme la classe JdbcTemplate, la classe TransactionTemplate est *threadsafe* d'un point de vue de la gestion de transaction. Elle ne conserve pas d'état de la transaction en cours. Cependant, elle conserve la configuration des transactions (je vous



en parle un peu plus loin). Vous pouvez donc partager l'instance de TransactionTemplate entre plusieurs méthodes, voire classes, si elles utilisent la même configuration.

La pilule rouge

On oublie le « tout enrobé » par la classe TransactionTemplate et on traite directement avec le PlatformTransactionManager.

Ici, Spring n'automatise plus — enfin, presque plus — l'ouverture, la validation et l'annulation de la transaction. C'est à vous de le faire.

```
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
// import ...

public class TicketManagerImpl extends AbstractManagerImpl implements TicketManager {

    @Inject
    @Named("txManagerTicket")
    private PlatformTransactionManager platformTransactionManager;

    @Override
    public void changerStatut(Ticket pTicket, TicketStatut pNewStatut,
        Utilisateur pUtilisateur, Commentaire pCommentaire) {

        TransactionStatus vTransactionStatus
            = platformTransactionManager.getTransaction(new DefaultTransactionDefinition());
        try {
            pTicket.setStatut(pNewStatut);
            getDaoFactory().getTicketDao().updateTicket(pTicket);
            // TODO : Ajout de la ligne d'historique + commentaire ...
        } catch (Throwable vEx) {
            platformTransactionManager.rollback(vTransactionStatus);
            throw vEx;
        }
        platformTransactionManager.commit(vTransactionStatus);
    }
}
```

Quand je dis Spring n'automatise **presque** plus, ce qu'il faut comprendre c'est que c'est à vous de « demander » l'ouverture/validation/annulation de la transaction, mais cela reste Spring qui s'occupe de faire le nécessaire.

Comme vous pouvez le remarquer on a toujours besoin du PlatformTransactionManager. Ceci ne change pas par rapport à la solution précédente. Il faut toujours le définir et l'injecter dans les *Managers*:

```
<bean id="txManagerTicket"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSourceTicket"/>
</bean>
```

Je propose d'améliorer un peu le code précédent. En effet, le *catch* de Throwable est vivement déconseillé pour plusieurs — mais ce n'est pas l'objet de ce cours. Mieux vaut passer par le *finally*:



```

TransactionStatus vTransactionStatus
    = platformTransactionManager.getTransaction(new DefaultTransactionDefinition());
try {
    pTicket.setStatut(pNewStatut);
    getDaoFactory().getTicketDao().updateTicket(pTicket);
    // TODO : Ajout de la ligne d'historique + commentaire ...

    platformTransactionManager.commit(vTransactionStatus);
    vTransactionStatus = null;
} finally {
    if (vTransactionStatus != null) {
        platformTransactionManager.rollback(vTransactionStatus);
    }
}

```

Attention, le code précédent n'est pas correct car il y a un détail de Spring TX à prendre en compte : Si une erreur se produit lors du *commit*, il ne faut pas appeler le *rollback* car Spring s'en charge. Si vous le faites, une exception sera levée par Spring.

Voici donc une implémentation correcte possible :

```

TransactionStatus vTransactionStatus
    = platformTransactionManager.getTransaction(new DefaultTransactionDefinition());
try {
    // le traitement transactionnel ...

    TransactionStatus vTScommit = vTransactionStatus;
    vTransactionStatus = null;
    platformTransactionManager.commit(vTScommit);
} finally {
    if (vTransactionStatus != null) {
        platformTransactionManager.rollback(vTransactionStatus);
    }
}

```

Il y a bien sûr d'autres moyens pour implémenter ce mécanisme (avec un *boolean* par exemple). Vous pouvez même créer une classe dédiée :

```
public class TransactionHelper {  
  
    @Inject  
    @Named("txManagerTicket")  
    private PlatformTransactionManager platformTransactionManager;  
  
    private DefaultTransactionDefinition definition = new DefaultTransactionDefinition();  
  
    public MutableObject<TransactionStatus> beginTransaction() {  
        return beginTransaction(null);  
    }  
  
    public MutableObject<TransactionStatus> beginTransaction(DefaultTransactionDefinition pDefinition) {  
        DefaultTransactionDefinition vDefinition = pDefinition != null ? pDefinition : definition;  
        Transactionstatus vStatus = platformTransactionManager.getTransaction(vDefinition);  
        return new MutableObject<TransactionStatus>(vStatus);  
    }  
  
    public void commit(MutableObject<TransactionStatus> pStatus) {  
        if (pStatus != null && pStatus.getValue() != null) {  
            pStatus.setValue(null);  
            platformTransactionManager.commit(pStatus.getValue());  
        }  
    }  
  
    public void rollback(MutableObject<TransactionStatus> pStatus) {  
        if (pStatus != null && pStatus.getValue() != null) {  
            pStatus.setValue(null);  
            platformTransactionManager.rollback(pStatus.getValue());  
        }  
    }  
}  
  
// ...  
  
MutableObject<TransactionStatus> vStatus = transactionHelper.beginTransaction();  
try {  
    // le traitement transactionnel ...  
    transactionHelper.commit(vStatus);  
} finally {  
    transactionHelper.rollback(vStatus);  
}
```

Personnellement, je préfère la pilule rouge. Surtout avec l'implémentation que je viens de vous montrer. Et ceci, surtout pour deux raisons :

1. Je trouve le code plus clair et lisible, pas de classe anonyme à créer et le debug en pas à pas en sera d'autant plus facile.
2. Je peux laisser remonter mes propres exceptions même si ce ne sont pas des *RuntimeException/Error*. Le *finally* fera le rollback. Dans le cas du *TransactionTemplate*, aucune exception non *RuntimeException/Error* ne peut

remonter car la méthode TransactionCallback.doInTransaction() ne déclare aucune exception.

```
public class TicketManagerImpl extends AbstractManagerImpl implements TicketManager {  
    // ...  
    public void changerStatut(Ticket pTicket, TicketStatut pNewstatut,  
                             Utilisateur pUtilisateur, Commentaire pcommentaire)  
        throws FunctionalException {  
  
        MutableObject<TransactionStatus> vStatus = transactionHelper.beginTransaction();  
        try {  
            // le traitement transactionnel ...  
            throw new FunctionalException("...");  
  
            transactionHelper.commit(vStatus);  
        } finally {  
            transactionHelper.rollback(vStatus);  
        }  
    }  
}
```

La propagation du contexte transactionnel

Il y a un truc qui me chiffonne tout de même. Reprenons l'exemple de la clôture d'un ticket. Le changement de statut du ticket se fait dans la classe TicketManagerImpl et l'ajout du commentaire, devrait se faire via une méthode de CommentaireManagerImpl. Or, a priori, cette méthode gère aussi une transaction vu qu'elle est censée être appelée lorsqu'un utilisateur ajoute lui-même un commentaire dans un ticket.

Ça ne va pas poser un problème ?

Ah, ah, bien vu !

La réponse est : ça peut ne pas en poser ! Et c'est ici qu'intervient la notion de [propagation](#) du contexte transactionnel.

En fait, Spring fait une différence entre une transaction physique (celle du SGBD par exemple) et une transaction logique (celle que vous ouvrez avec PlatformTransactionManager.getTransaction(...)).

Quand je parle de *propagation du contexte transactionnel* je parle de comment Spring gère le lien entre les transactions logiques et les transactions physiques.

Spring propose plusieurs types de propagation, que vous pouvez retrouver dans l'interface TransactionDefinition :

- PROPAGATION_MANDATORY
- PROPAGATION_NESTED
- PROPAGATION_NEVER
- PROPAGATION_NOT_SUPPORTED
- PROPAGATION_REQUIRED
- PROPAGATION_REQUIRES_NEW
- PROPAGATION_SUPPORTS

Je ne vais pas tous les détailler ici, je vous renvoie à la [JavaDoc](#) pour connaître le détail de chacun. Je vais cependant vous détailler les deux principaux : PROPAGATION_REQUIRED et PROPAGATION_REQUIRES_NEW.

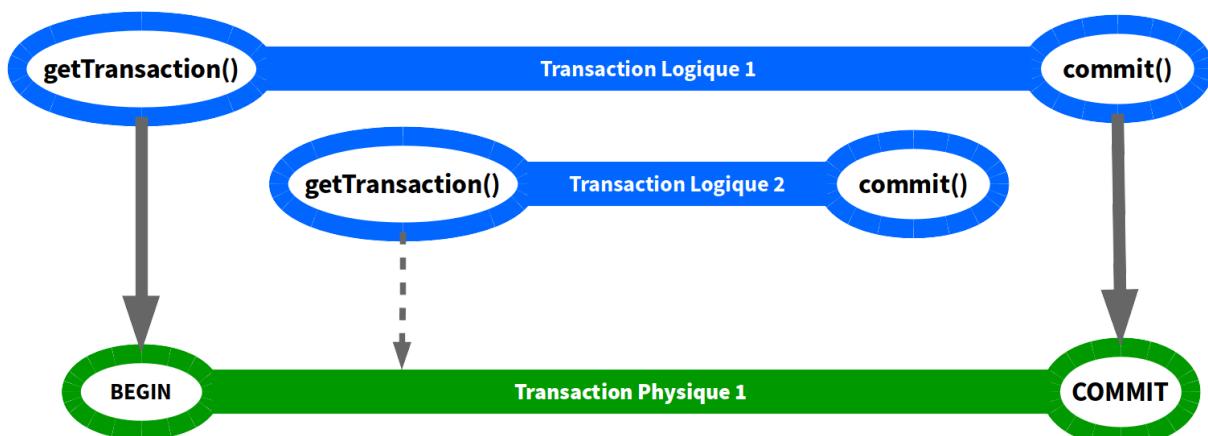


Il est important de noter que la gestion des transactions par Spring se fait par *Thread*. Si deux *Threads* différents ouvrent une transaction, ils obtiendront toujours des transactions différentes.

PROPAGATION_REQUIRED

Le type PROPAGATION_REQUIRED est le type de propagation par défaut.

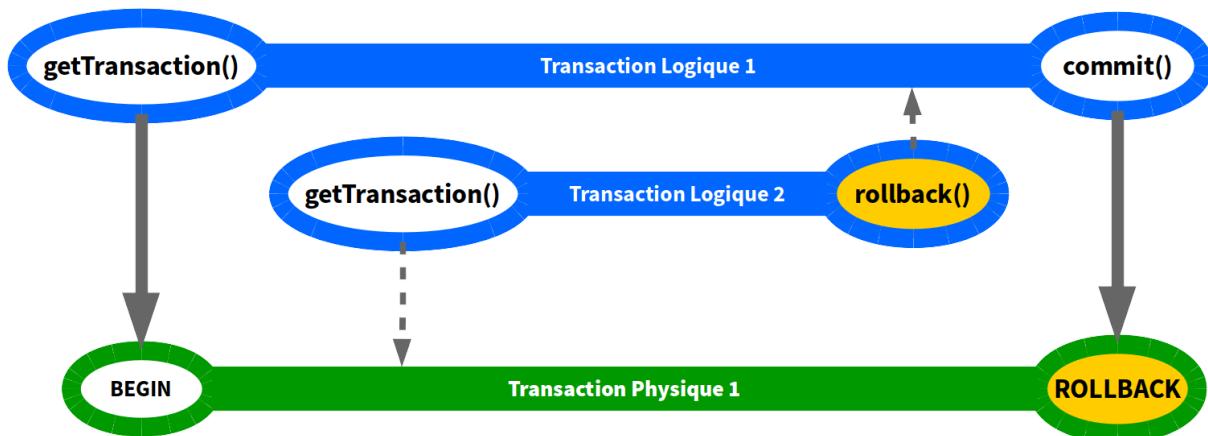
Quand vous demandez l'ouverture d'une transaction (`PlatformTransactionManager.getTransaction(...)`) dans ce mode, si une transaction physique est déjà en cours, alors elle est utilisée. Sinon, une nouvelle transaction est ouverte.



Si votre demande d'ouverture déclenche l'ouverture d'une transaction physique, alors votre transaction logique devient la transaction principale. Si en revanche vous réutilisez une transaction physique existante, alors votre transaction logique devient une sous-transaction.

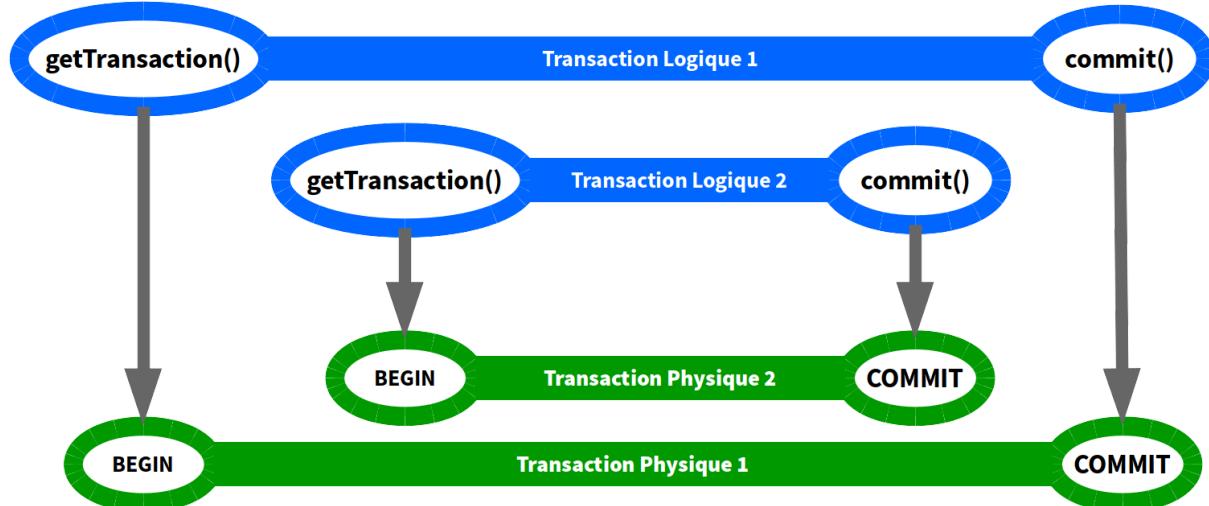
Voici les règles de validation/annulation qui s'appliquent dans un sous-transaction :

- Si vous appelez la méthode `commit()`, cela n'a aucun effet.
- Si vousappelez la méthode `rollback()`, cela marque la transaction principale en *rollback only*. Si vousappelez alors `commit()` dans la transaction principale, cela va déclencher un **rollback** et lever une `UnexpectedRollbackException`.

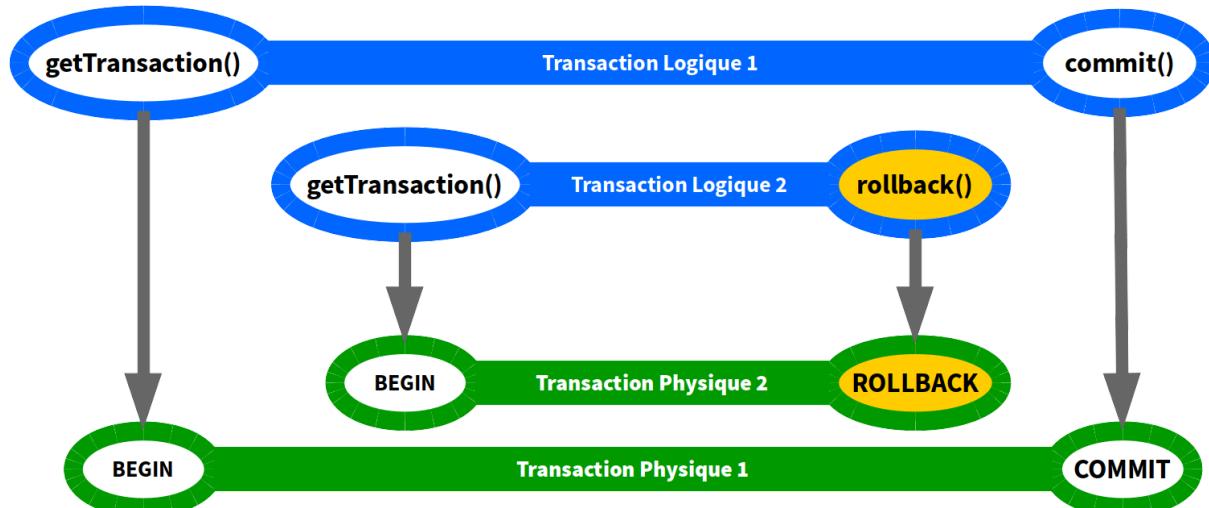


PROPAGATION_REQUIRES_NEW

Avec le type PROPAGATION_REQUIRES_NEW, quand vous demandez l'ouverture d'une transaction, une nouvelle transaction physique est toujours créée, peu importe si une transaction physique existe déjà ou pas.



Les transactions physiques (et logiques du coup), sont totalement indépendantes. L'annulation de l'une n'affecte aucunement l'autre.



Transactions en PROPAGATION_REQUIRES_NEW (rollback)

OK, je vous ai parlé des types de propagation des transactions. Mais comment le définir ? Eh bien, comme n'importe quel autres paramètres des transactions. C'est ce que je vous montre dans la section suivante.

Ajuster les paramètres des transactions

Le détail des paramètres définissables et leur valeurs par défaut se trouve dans la [documentation officielle](#) et la [JavaDoc](#).

La définition des paramètres des transactions se fait différemment en fonction de la couleur de la pilule...

Si vous utilisez le TransactionTemplate



Si vous utilisez le TransactionTemplate, la définition des paramètres se fait directement via les *setters* du TransactionTemplate avant l'appel à la méthode execute(...):

```
TransactionTemplate vTransactionTemplate = new TransactionTemplate();
vTransactionTemplate.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRES_NEW);
vTransactionTemplate.setTimeout(30); // 30 secondes

//vTransactionTemplate.execute(...);
```

Si vous utilisez le PlatformTransactionManager

Si vous utilisez le PlatformTransactionManager, la définition des paramètres se fait via l'instance de DefaultTransactionDefinition passée à l'ouverture de la transaction :

```
DefaultTransactionDefinition vDefintion = new DefaultTransactionDefinition();
vDefintion.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRES_NEW);
vDefintion.setTimeout(30); // 30 secondes

platformTransactionManager.getTransaction(vDefintion);
```

Synchroniser les transactions hors de Spring JDBC

Si vous utilisez *Spring JDBC* pour exécuter vos requêtes SQL, comme je le disais au début de ce chapitre, les modules fonctionnent parfaitement ensemble. La gestion des transactions au niveau de Spring JDBC est transparente pour le développeur.

En revanche, si des accès sont fait directement avec l'API JDBC (par un framework de test par exemple), alors ce dernier doit être intégré dans la mécanique transactionnelle.

Pour cela, Spring fournit deux classes utilitaires.

La classe DataSourceUtils permet d'obtenir une Connection. Spring crée un proxy au niveau de la connexion.

```
Connection vConnection = DataSourceUtils.getConnection(dataSourceTicket);
```

C'est la méthode recommandée. Cependant il arrive que n'ayez même pas besoin d'une Connexion mais directement d'une instance de DataSource. Dans ce cas, n'utilisez pas la DataSource de base (le bean *dataSourceTicket* dans notre application). Sans quoi, vous seriez totalement en dehors du mécanisme transactionnel de Spring. La solution est d'obtenir un proxy de la DataSource grâce à la classe TransactionAwareDataSourceProxy :

```
DataSource vDataSource = new TransactionAwareDataSourceProxy(dataSourceTicket);
```

Si besoin, vous trouverez plus de détail dans la [documentation officielle](#).

Conclusion

La gestion des transactions avec Spring peut vous paraître au premier abord un peu verbeuse et répétitive, surtout avec la définition des paramètres de ces transactions. Mais avec ce que vous avez appris tout au long de ce cours (notamment en première partie),



vous devriez être en mesure de simplifier tout cela avec l'injection de dépendances, les classes mères abstraites...

Ensuite, à vous de faire votre choix entre TransactionTemplate et PlatformTransactionManager.

La gestion des transactions est un sujet vaste et les possibilités offertes par Spring sont assez importantes. N'hésitez pas à parcourir la [documentation officielle](#) pour approfondir le sujet.



Récapitulatif général

Youhou, ça y est, vous arrivez au bout du tunnel ! Bravo !

Alors, je ne sais pas si tout est clair pour vous, mais ça fait quand même pas mal d'informations à ingurgiter non ?

Je vous propose de faire un petit récapitulatif général de ce que nous avons vu ensemble tout au long de ce cours.

Partie 1 : Mettre en œuvre une inversion de contrôle

Dans la première partie, je vous ai mis l'eau à la bouche avec ma tartiflette, l'**inversion de contrôle**, l'**injection de dépendances** et l'**abstraction** dont voici un petit rappel :

- l'**inversion de contrôle** : le flot d'exécution n'est plus totalement géré par le développeur ou les traitements de l'application, mais il est pris en charge par des éléments annexes.
- l'**injection de dépendances** : les objets dépendant d'autres objets ne se soucient plus de comment les obtenir, ils leurs sont directement injectés.
- les **abstractions** : très souvent réalisées à l'aide d'interfaces, ces abstractions sont les « contrats » des interactions entre les différentes couches de l'application. Les couches basses implémentent ces abstractions. Les couches hautes dépendent de ces abstractions et non des implémentations faites dans les couches basses.

Nous avons vu que cela était une réponse aux principes [SOLID](#):

- Single responsibility principle (responsabilité unique)
- Open/closed principle (ouvert/fermé)
- Liskov substitution principle (substitution de Liskov)
- Interface segregation principle (ségrégation des interfaces)
- Dependency inversion principle (inversion des dépendances)

Après avoir implémenté une injection de dépendance simple à partir de zéro, nous sommes passés à la vitesse supérieure avec Spring !

Ce qu'il faut retenir de l'approche de Spring, c'est qu'il fait de l'injection de dépendances en s'appuyant sur un *IoC container*.

Les *beans* sont déclarés (via la configuration XML) ou se déclarent (via les annotations) dans l'*IoC container*. Ces beans déclarent aussi les dépendances dont ils ont besoin. Tout cela forme le *context* Spring. À partir de ce *context*, Spring se charge d'instancier les beans et d'injecter les dépendances.

N'oubliez pas les divers conseils que j'ai pu vous donner dans cette première partie comme :

- Découpez la configuration du contexte par couche.
- Si vous mixer la Configuration XML et les annotations, restez consistant dans votre manière de faire sur tout le projet.
- Etc.

Partie 2 : Faciliter le développement avec Spring

Dans la deuxième partie, je vous ai montré quelques modules complémentaires du framework Spring :

- *Spring JDBC*: employé dans la couche **consumer** pour exécuter des requêtes SQL.
- *Spring TX*: employé dans la couche **business** pour gérer les transactions

Ces deux modules fonctionnent ensemble de manière transparente.

Pour *Spring JDBC*, les principales classes/interfaces à utiliser sont :

- `JdbcTemplate` : pour définir et exécuter une requête SQL
- `NamedParameterJdbcTemplate` : comme `JdbcTemplate` mais pour des requêtes avec paramètres nommés (:param)
- `RowMapper<T>` : interface à implémenter pour définir le mapping entre les lignes de résultats d'une requête et les objets java à créer.

Pour *Spring TX*, les principales classes/interfaces à utiliser sont :

- `PlatformTransactionManager` : interface pour faire le lien entre les transactions logique et physique (implémentée par la classe `DataSourceTransactionManager` dans le cas d'une `DataSource JDBC`)
- `TransactionTemplate`, `TransactionCallback`, `TransactionCallbackWithoutResult` : pour ouvrir et configurer une transaction et implémenter le traitement transactionnel
- Ou :
 - directement le `PlatformTransactionManager` : pour ouvrir/valider/annuler une transaction
 - et la classe `DefaultTransactionDefinition` pour configurer les paramètres de la transaction

Spring est vraiment un framework riche. Dans ce cours nous n'en avons vu qu'une partie. Même si cela est déjà suffisant pour commencer à développer des applications plus professionnelles, je vous invite à explorer le framework et sa documentation.

Vous pouvez regarder en premier lieu la [documentation de Spring Test](#). Ce module va vous faciliter la vie lors du développement des tests d'intégration en permettant de configurer rapidement un contexte Spring spécifique pour vos tests.

Maintenant, à vous de jouer. Lancez-vous, testez, expéimentez... Et d'ici peu de temps vous ne pourrez plus vous passer de Spring !



Spring Boot(Créez un Microservice grâce à Spring Boot)

Spring Initializr

Avez-vous déjà commandé une salade à composer au restaurant ? Et bien, *Spring Initializr* fait plus ou moins la même chose : il vous permet de **composer votre application selon vos besoins**.

Pour débuter, rendez-vous sur <https://start.spring.io>. Cliquez sur "Switch to the full version" :

The screenshot shows the Spring Initializr interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below that, there's a search bar with the placeholder "Generate a [Maven Project] with [Java] and Spring Boot 1.5.9". Under "Project Metadata", there are fields for "Group" (com.example) and "Artifact" (demo). Under "Dependencies", there's a section for "Selected Dependencies" with options like "Web, Security, JPA, Actuator, Devtools...". A green button at the bottom right says "Generate Project". A red circle highlights the "Switch to the full version" link at the bottom left of the interface.

Vous verrez apparaître, en bas, tous les **starters et ensembles de dépendances** pour les cas d'usage les plus courants. Cela va du starter web, que nous avons vu dans le chapitre précédent, aux bases de données, en passant par les moteurs de template, la sécurité, le messaging, ainsi que d'autres outils très importants dans une architecture Microservices, que nous verrons plus tard.

Nous allons, dans les prochains chapitres, développer un **mini-système d'e-commerce** fondé sur l'architecture Microservices avec *Spring Boot*. Nous allons commencer par un premier Microservice qui **gère les produits** que nous allons proposer à la vente. Il doit pouvoir ajouter, supprimer, mettre à jour et afficher les produits.

Dans une première étape, nous allons créer une version très simplifiée de ce Microservice. Nous enrichirons ce Microservice au fur et à mesure de notre découverte des différents concepts à assimiler.

Création et importation à partir de Spring Initializr

Retournez sur Spring Initializr et renseignez les Metadata du projet comme suit :

- Group : com.ecommerce
- Artifact : microcommerce
- Name : microcommerce
- Packaging : jar



- Java Version : 8

La figure suivante vous présente le formulaire renseigné avec ces informations :

Project Metadata

Artifact coordinates

Group

com.ecommerce

Artifact

microcommerce

Name

microcommerce

Description

Un ecommerce basé sur la MSA

Package Name

com.ecommerce.microcommerce

Packaging

Jar

Java Version

8

Too many options? [Switch back to the simple version.](#)

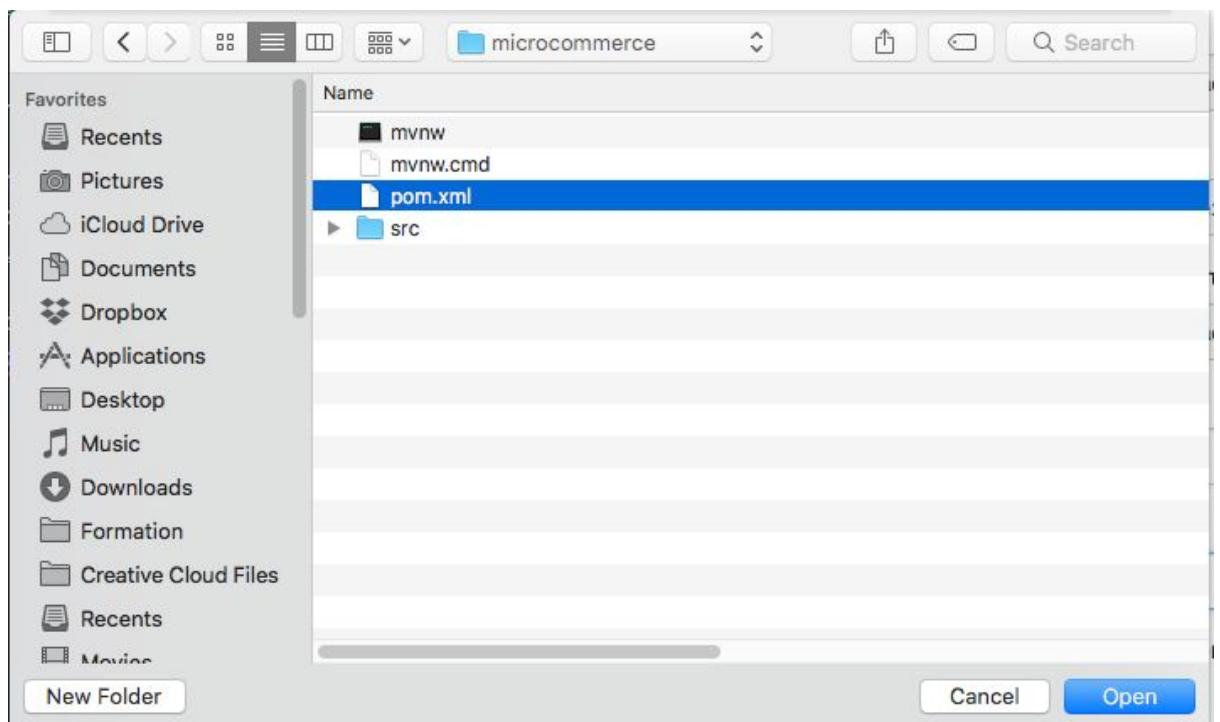
Interface de Spring Initializr, full version

Ensuite, suivez les étapes suivantes :

1. Sélectionnez en bas le starter *Web*, comme illustré dans la figure qui suit :

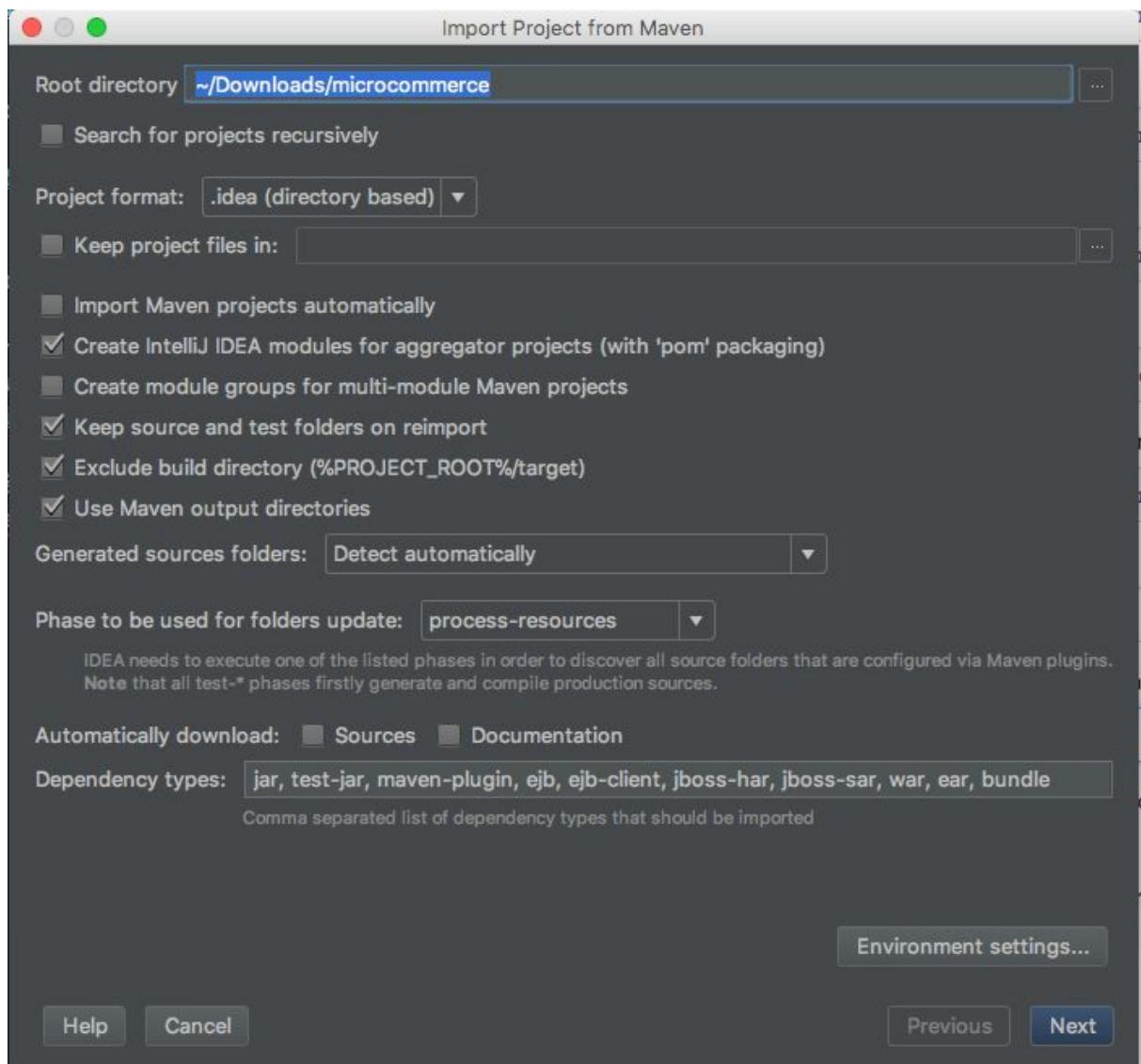


- Generate Project** * + ↗
- | | |
|---|---|
| Core <ul style="list-style-type: none"> <input type="checkbox"/> DevTools
Spring Boot Development Tools <input type="checkbox"/> Security
Secure your application via spring-security <input type="checkbox"/> Lombok | Web <ul style="list-style-type: none"> <input checked="" type="checkbox"/> Web
Full-stack web development with Tomcat and Spring MVC <input type="checkbox"/> Reactive Web
Reactive web development with Netty and Spring WebFlux
requires Spring Boot >=2.0.0.M1 |
|---|---|
2. Sélection du starter "Web"
 3. Cliquez sur "Generate Project" et téléchargez l'application générée.
 4. Dans ce cours, nous allons utiliser comme IDE : **IntelliJ**. Son installation est très simple.
 5. Procédez à l'extraction de l'application téléchargée. Si vous avez utilisé les mêmes noms que moi, elle devrait s'appeler *microcommerce.zip*.
 6. Une fois sur la page d'accueil d'IntelliJ, cliquez sur "Import Project" :
 - 7.
- 
- 8.
 9. Sélectionnez le dossier de l'application *microcommerce* puis sélectionnez le *pom.xml*:



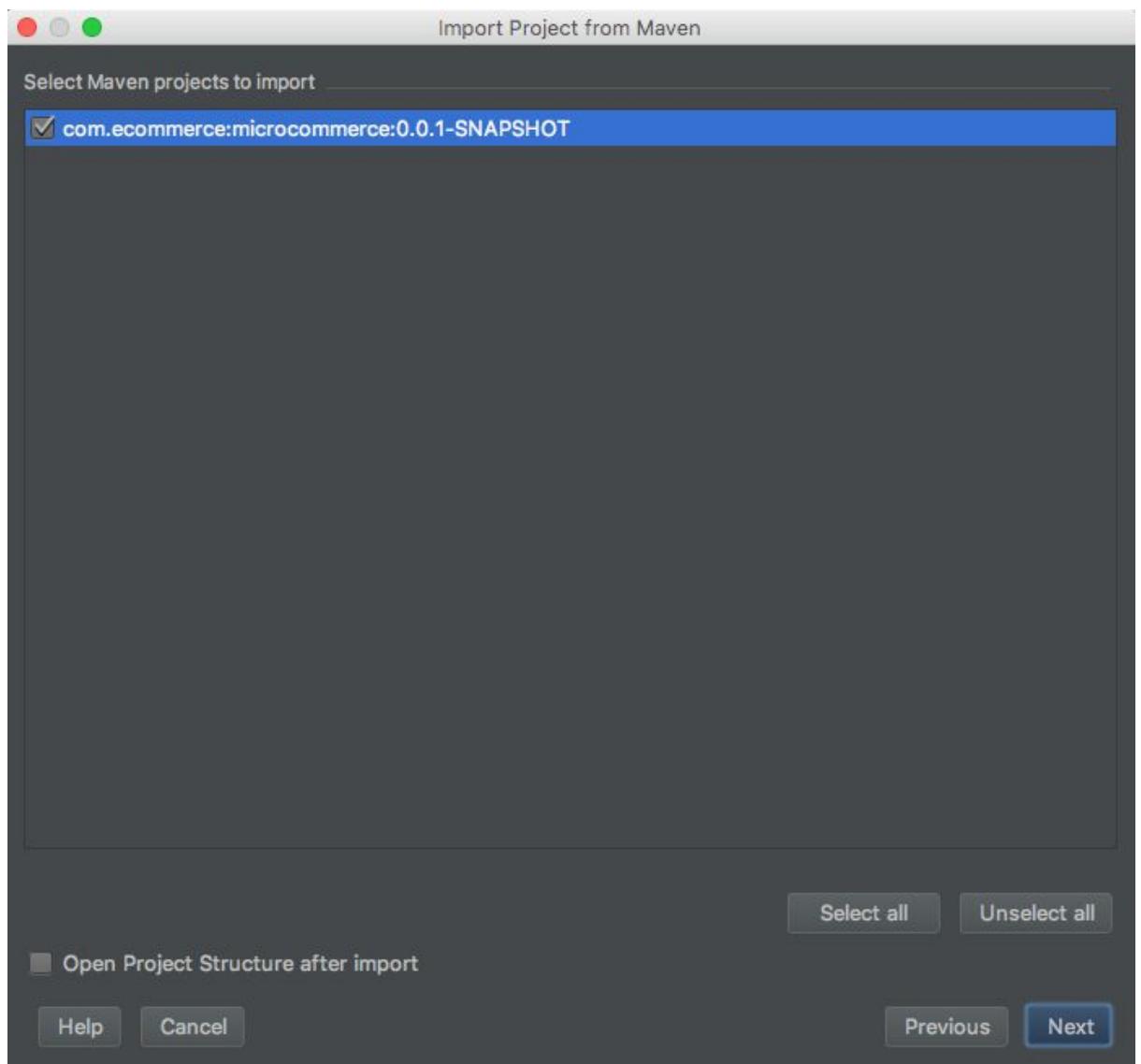
10.

11. Laissez les options d'import de projet *Maven* par défaut :



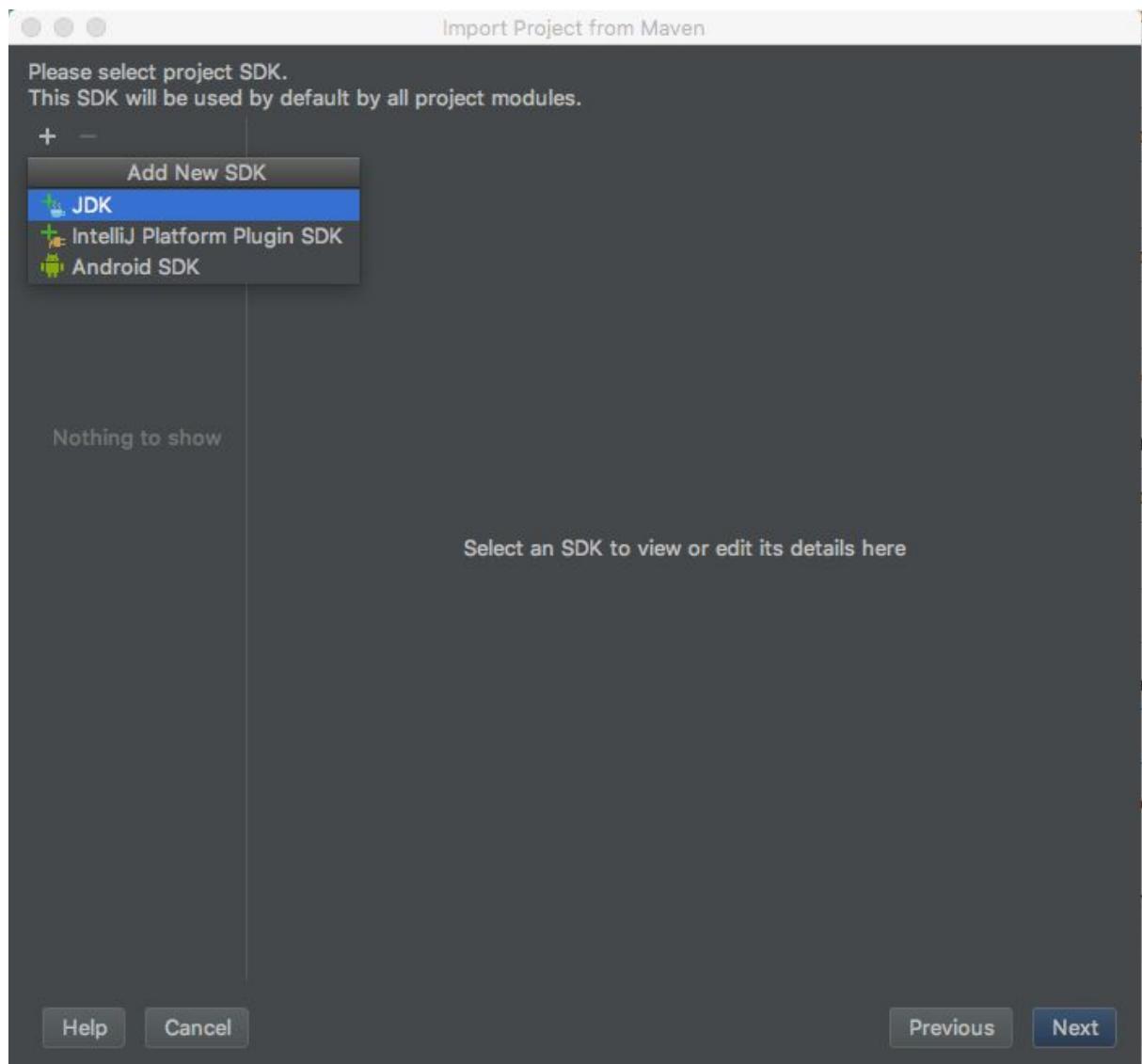
12.

13. Cliquez sur "Next" à la prochaine fenêtre :



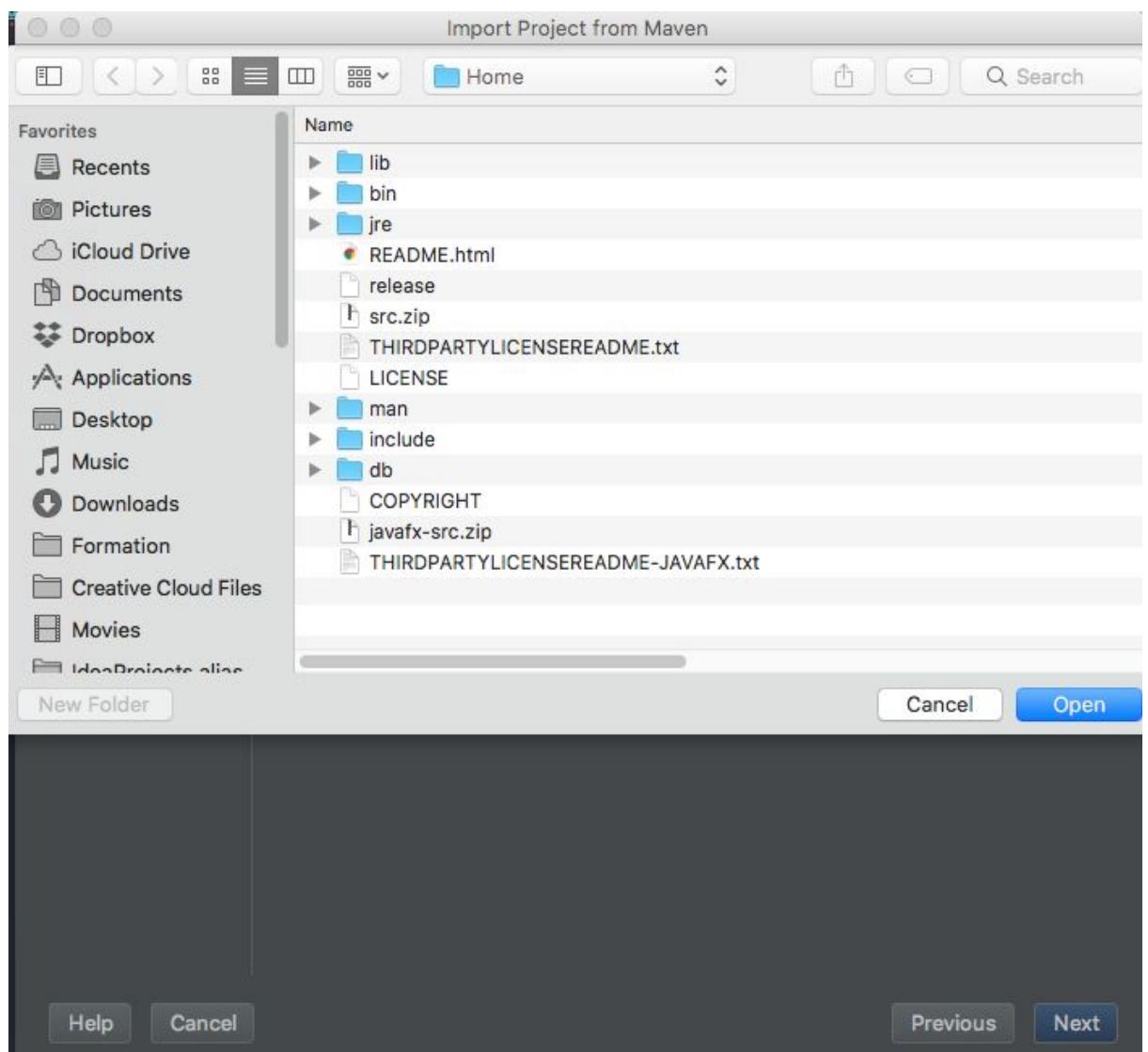
14.

15. Si vous venez d'installer *IntelliJ*, il vous demandera de choisir le JDK : cliquez sur "+" puis *JDK*:



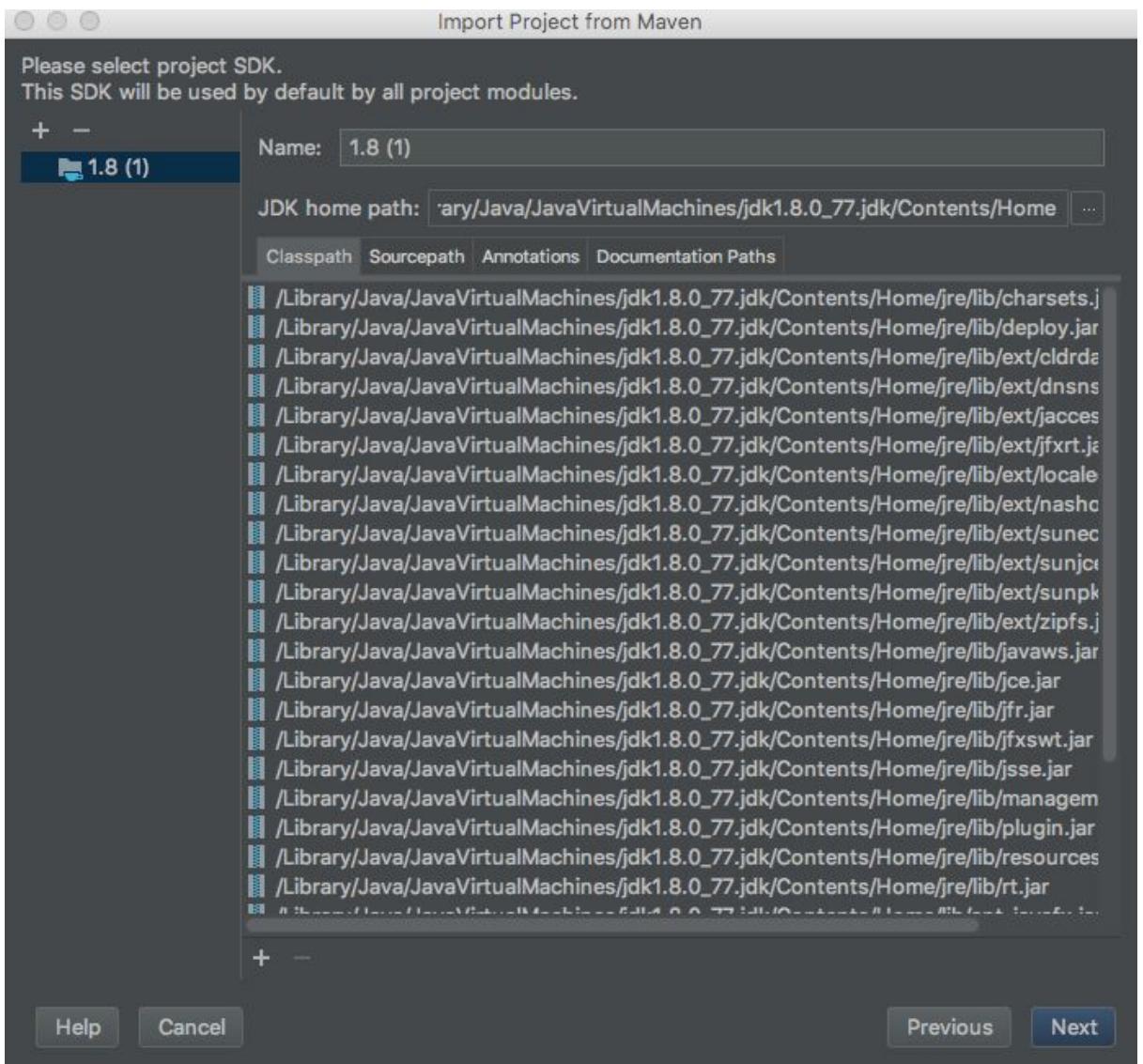
16.

17. Sélectionnez le répertoire "Home" de votre *JDK*:



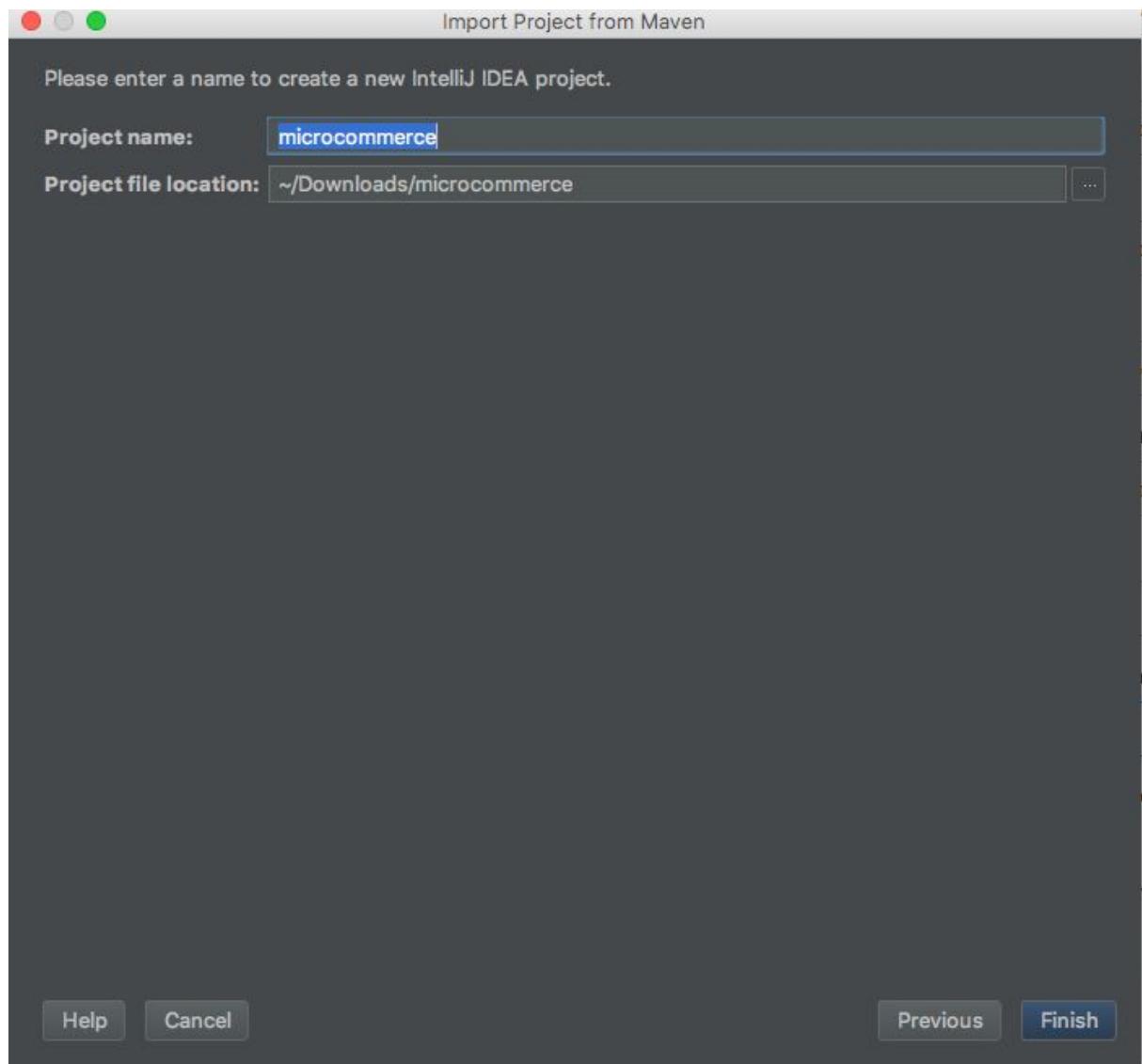
18.

19. Cliquez sur "Next" :



20.

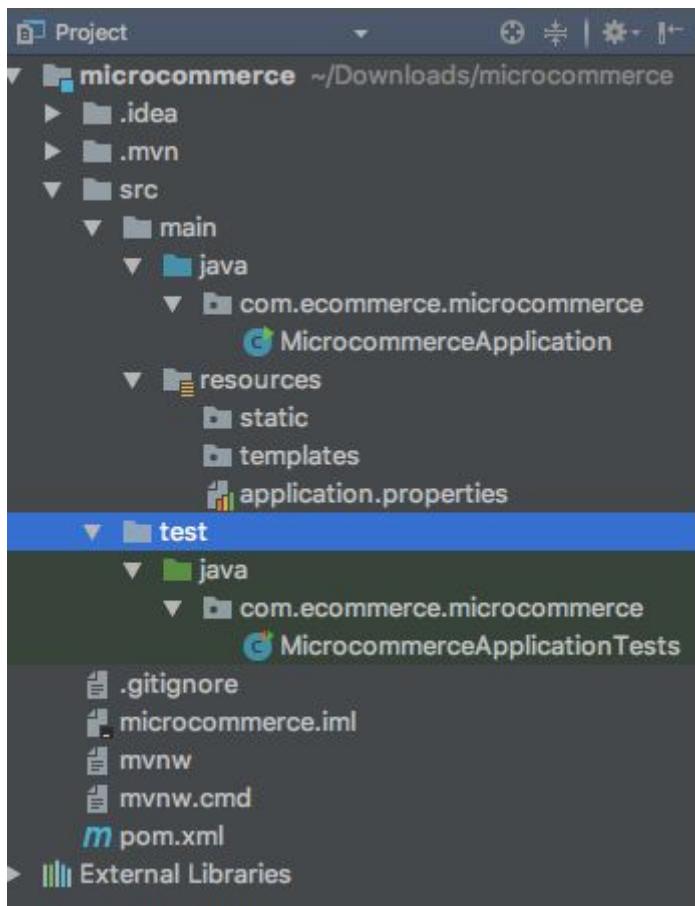
21. Entrez "microcommerce" pour le nom du projet et cliquez sur "Finish" :



22.

Analyse du code obtenu

Si tout s'est déroulé correctement, vous devriez avoir cette arborescence :



Étudions maintenant en détail les différents éléments de cette arborescence :

pom.xml

Comme nous l'avons vu dans le chapitre précédent, ce *pom.xml* hérite du parent *spring-boot-starter-parent* qui nous permet de ne plus nous soucier des versions des dépendances et de leur compatibilité :

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.9.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository --&gt;
&lt;/parent&gt;</pre>
```

Comme je l'ai indiqué dans le chapitre précédent, vous avez la possibilité de choisir la version de Java à utiliser. Comme nous avons sélectionné *Java 8* dans l'interface de Spring Initializr, une balise XML *<java.version>* a été ajoutée dans la partie *properties* du fichier *pom.xml*, comme nous pouvons le voir dans le code qui suit :

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
</properties>
```

Le *pom.xml* définit ensuite une dépendance vers le starter qui va ajouter à notre Microservice toutes les dépendances de base nécessaires pour démarrer rapidement :

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

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



Pour voir la liste des dépendances importées, rendez-vous à gauche dans "External



By Marwén Saidi
marwen.saidi@orange.com

- ▼  External Libraries
 -  < 1.8 (1) > /Library/Java/JavaVirtualMachines/jdk1.8.0_77.jdk/Contents/Home
 -  Maven: ch.qos.logback:logback-classic:1.1.11
 -  Maven: ch.qos.logback:logback-core:1.1.11
 -  Maven: com.fasterxml.jackson.core:jackson-annotations:2.8.0
 -  Maven: com.fasterxml.jackson.core:jackson-core:2.8.10
 -  Maven: com.fasterxml.jackson.core:jackson-databind:2.8.10
 -  Maven: com.fasterxml.xml:classmate:1.3.4
 -  Maven: com.jayway.jsonpath:json-path:2.2.0
 -  Maven: com.vaadin.external.google:android-json:0.0.20131108.vaadin1
 -  Maven: javax.validation:validation-api:1.1.0.Final
 -  Maven: junit:junit:4.12
 -  Maven: net.minidev:accessors-smart:1.1
 -  Maven: net.minidev:json-smart:2.2.1
 -  Maven: org.apache.tomcat.embed:tomcat-embed-core:8.5.23
 -  Maven: org.apache.tomcat.embed:tomcat-embed-el:8.5.23
 -  Maven: org.apache.tomcat.embed:tomcat-embed-websocket:8.5.23
 -  Maven: org.apache.tomcat:tomcat-annotations-api:8.5.23
 -  Maven: org.assertj:assertj-core:2.6.0
 -  Maven: org.hamcrest:hamcrest-core:1.3
 -  Maven: org.hamcrest:hamcrest-library:1.3
 -  Maven: org.hibernate:hibernate-validator:5.3.6.Final
 -  Maven: org.jboss.logging:jboss-logging:3.3.1.Final
 -  Maven: org.mockito:mockito-core:1.10.19
 -  Maven: org.objenesis:objenesis:2.1
 -  Maven: org.ow2.asm:asm:5.0.3
 -  Maven: org.skyscreamer:jsonassert:1.4.0
 -  Maven: org.slf4j:jcl-over-slf4j:1.7.25
 -  Maven: org.slf4j:jul-to-slf4j:1.7.25
 -  Maven: org.slf4j:log4j-over-slf4j:1.7.25
 -  Maven: org.slf4j:slf4j-api:1.7.25
 -  Maven: org.springframework.boot:spring-boot:1.5.9.RELEASE
 -  Maven: org.springframework.boot:spring-boot-autoconfigure:1.5.9.RELEASE
 -  Maven: org.springframework.boot:spring-boot-starter:1.5.9.RELEASE
 -  Maven: org.springframework.boot:spring-boot-starter-logging:1.5.9.RELEASE
 -  Maven: org.springframework.boot:spring-boot-starter-test:1.5.9.RELEASE
 -  Maven: org.springframework.boot:spring-boot-starter-tomcat:1.5.9.RELEASE
 -  Maven: org.springframework.boot:spring-boot-starter-web:1.5.9.RELEASE
 -  Maven: org.springframework.boot:spring-boot-test:1.5.9.RELEASE
 -  Maven: org.springframework.boot:spring-boot-test-autoconfigure:1.5.9.RELEASE
 -  Maven: org.springframework:spring-aop:4.3.13.RELEASE
 -  Maven: org.springframework:spring-beans:4.3.13.RELEASE
 -  Maven: org.springframework:spring-context:4.3.13.RELEASE
 -  Maven: org.springframework:spring-core:4.3.13.RELEASE
 -  Maven: org.springframework:spring-expression:4.3.13.RELEASE
 -  Maven: org.springframework:spring-test:4.3.13.RELEASE
 - Maven: org.springframework:spring-web:4.3.13.RELEASE
 - Maven: org.springframework:spring-webmvc:4.3.13.RELEASE
 - Maven: org.yaml:snakeyaml:1.17

Libraries" :

Vous avez dans cette liste principalement :

- **Jackson** : permet de parser JSON et faire le lien entre les classes Java et le contenu JSON.
- **Tomcat** : intégré, va nous permettre de lancer notre application en exécutant tout simplement le jar sans avoir à le déployer dans un serveur d'application.
- **Hibernate** : facilite la gestion des données.
- **Logging** grâce à logback et autres.

MicrocommerceApplication.java

Cette classe, générée automatiquement par Spring Boot, est le point de démarrage de l'application :

```
@SpringBootApplication
public class MicrocommerceApplication {

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

Elle lance, entre autres, la classe *SpringApplication*, responsable du démarrage de l'application Spring. Cette classe va créer le fameux *ApplicationContext* dans lequel iront toutes les configurations générées automatiquement ou ajoutées par vos soins.

Mais le plus important ici, c'est bien sûr l'annotation **@SpringBootApplication**, qui est une simple encapsulation de trois annotations :

1. **@Configuration** : donne à la classe actuelle la possibilité de définir des configurations qui iront remplacer les traditionnels fichiers XML. Ces configurations se font via des *Beans*.
2. **@EnableAutoConfiguration** : l'annotation vue précédemment qui permet, au démarrage de Spring, de générer automatiquement les configurations nécessaires en fonction des dépendances situées dans notre *classpath*.
3. **@ComponentScan** : Indique qu'il faut scanner les classes de ce package afin de trouver des Beans de configuration.

Nous reviendrons sur la configuration dans une prochaine section. Si vous souhaitez personnaliser finement le comportement de Spring Boot, il vous suffit de remplacer `@SpringBootApplication` par ces 3 annotations :

```
...
...
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class MicrocommerceApplication {
...
...
}
```

Cette modification permet notamment de paramétrer l'annotation @ComponentScan, par exemple pour cibler des fichiers à scanner.

application.properties

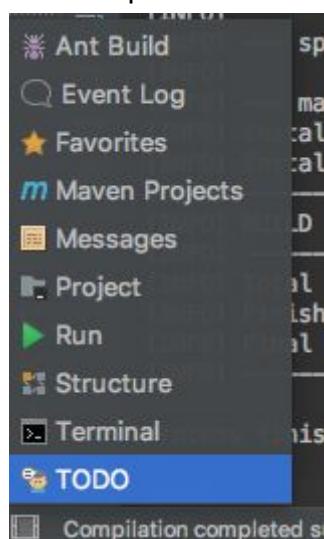
Ce fichier est votre ami (oui, comme Google). Ce fichier va vous permettre de modifier très simplement un nombre impressionnant de configurations liées à Spring Boot et ses dépendances. Par exemple : changer le port d'écoute de Tomcat, l'emplacement des fichiers de log, les paramètres d'envoi d'emails, etc... Je vous invite à jeter un oeil sur [la liste complète ici](#). Nous reviendrons sur ce fichier dans une prochaine section.

MicrocommerceApplicationTests.java

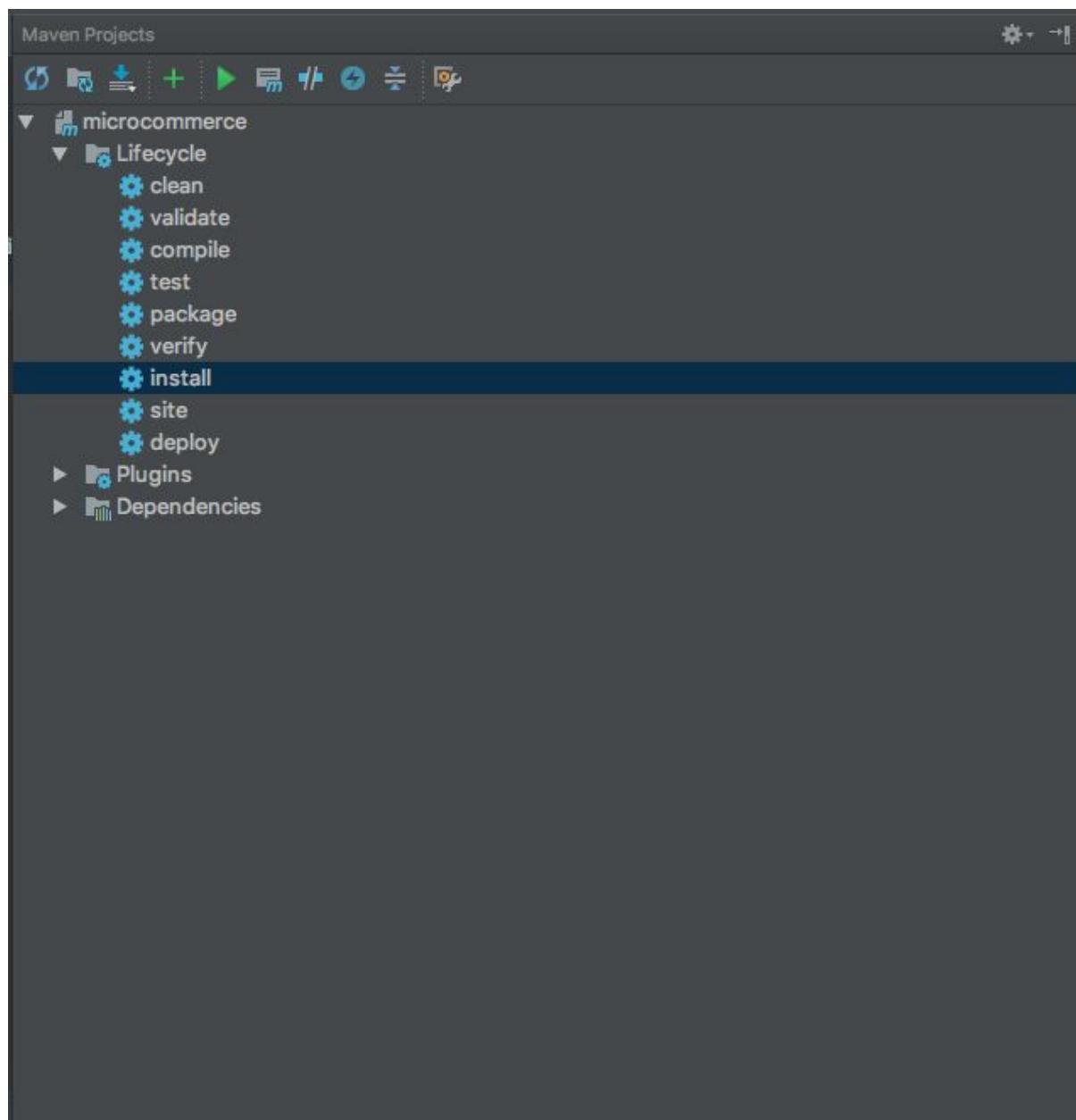
Ce fichier vous permet d'écrire vos tests.

Exécuter l'application

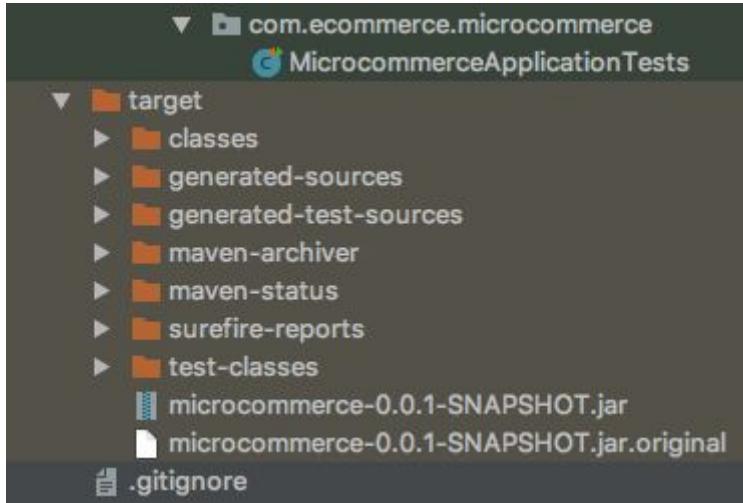
Nous n'avons rien ajouté dans notre application pour l'instant, mais nous pouvons déjà l'exécuter. Si vous n'avez pas le panneau Maven à droite, rendez-vous en bas à gauche d'IntelliJ pour l'activer :



Double-cliquez ensuite sur "Install" sous "Lifecycle" de Maven dans le panneau de droite :



L'application sera compilée et vous retrouverez le jar sous le nouveau dossier "*Target*" créé pour l'occasion par Maven :



Exécutez enfin l'application depuis un terminal comme n'importe quel jar grâce à la commande :

```
java -jar Chemin/vers/microcommerce/target/microcommerce-0.0.1-SNAPSHOT.jar
```

Vous devriez alors avoir un retour de ce type :

```
MacBook-Pro-de-amar:openclassrooms_moulinex amar$ java -jar /Users/amar/Downloads/microcommerce/target/microcommerce-0.0.1-SNAPSHOT.jar
.
.
.
:: Spring Boot ::      (v1.5.9.RELEASE)

2018-01-14 19:40:04.613  INFO 84419 --- [           main] c.e.m.MicrocommerceApplication        : Starting MicrocommerceApplication v0.0.1-SNAPSHOT on MacBoo
k-Pro-de-amar.local with PID 84419 (/Users/amar/Downloads/microcommerce/target/microcommerce-0.0.1-SNAPSHOT.jar started by amar in /Users/amar/Atom/openclassroo
ms_moulinex)
2018-01-14 19:40:04.616  INFO 84419 --- [           main] c.e.m.MicrocommerceApplication        : No active profile set, falling back to default profiles: de
fault
2018-01-14 19:40:04.678  INFO 84419 --- [           main] a.tionConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.Anno
tationConfigEmbeddedWebApplicationContext@5cb0d902: startup date [Sun Jan 14 19:40:04 CET 2018]; root of context hierarchy
2018-01-14 19:40:05.900  INFO 84419 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
2018-01-14 19:40:05.913  INFO 84419 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2018-01-14 19:40:05.914  INFO 84419 --- [           main] o.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.23
2018-01-14 19:40:06.020  INFO 84419 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2018-01-14 19:40:06.020  INFO 84419 --- [ost-startStop-1] o.s.web.context.ContextLoader      : Root WebApplicationContext: initialization completed in 134
5 ms
2018-01-14 19:40:06.151  INFO 84419 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean : Mapping servlet: 'dispatcherServlet' to []
2018-01-14 19:40:06.155  INFO 84419 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'characterEncodingFilter' to: [//*]
2018-01-14 19:40:06.155  INFO 84419 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'hiddenHttpMethodFilter' to: [//*]
2018-01-14 19:40:06.155  INFO 84419 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'httpPutFormContentFilter' to: [//*]
2018-01-14 19:40:06.155  INFO 84419 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'requestContextFilter' to: [//*]
2018-01-14 19:40:06.448  INFO 84419 --- [           main] s.w.s.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.con
text.embedded.AnnotationConfigEmbeddedWebApplicationContext@5cb0d902: startup date [Sun Jan 14 19:40:04 CET 2018]; root of context hierarchy
2018-01-14 19:40:06.543  INFO 84419 --- [           main] s.w.s.m.a.RequestMappingHandlerMapping : Mapped "[{error}]" onto public org.springframework.http.Re
sponseEntity<java.util.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.error.BasicErrorController.error(javax.servlet.http.HttpSer
vletRequest)
2018-01-14 19:40:06.544  INFO 84419 --- [           main] s.w.s.m.a.RequestMappingHandlerMapping : Mapped "[{error},produces=[text/html]]" onto public org.sp
ringframework.web.servlet.ModelAndView org.springframework.boot.autoconfigure.web.BasicErrorController.errorHtml(javax.servlet.http.HttpServletRequest,javax.ser
vlet.http.HttpServletResponse)
2018-01-14 19:40:06.580  INFO 84419 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class o
rg.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-01-14 19:40:06.580  INFO 84419 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.spring
framework.web.servlet.resource.ResourceHttpRequestHandler]
2018-01-14 19:40:06.624  INFO 84419 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [cla
ss org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-01-14 19:40:06.767  INFO 84419 --- [           main] o.s.j.e.a.AnnotationMBeanExporter       : Registering beans for JMX exposure on startup
2018-01-14 19:40:06.835  INFO 84419 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2018-01-14 19:40:06.841  INFO 84419 --- [           main] c.e.m.MicrocommerceApplication        : Started MicrocommerceApplication in 2.712 seconds (JVM runn
ing for 3.145)
```

Dans les dernières lignes vous remarquerez cette phrase : "Tomcat started on port(s): 8080 (http)" Ce qui vous indique que votre application tourne et qu'elle est en écoute grâce à **tomcat sur le port 8080**.

Et bien, rendez-vous dans votre navigateur à l'adresse <http://localhost:8080>. Vous obtenez alors cette magnifique erreur, car nous n'avons encore pas fourni d'éléments à afficher !

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Jan 14 19:43:17 CET 2018

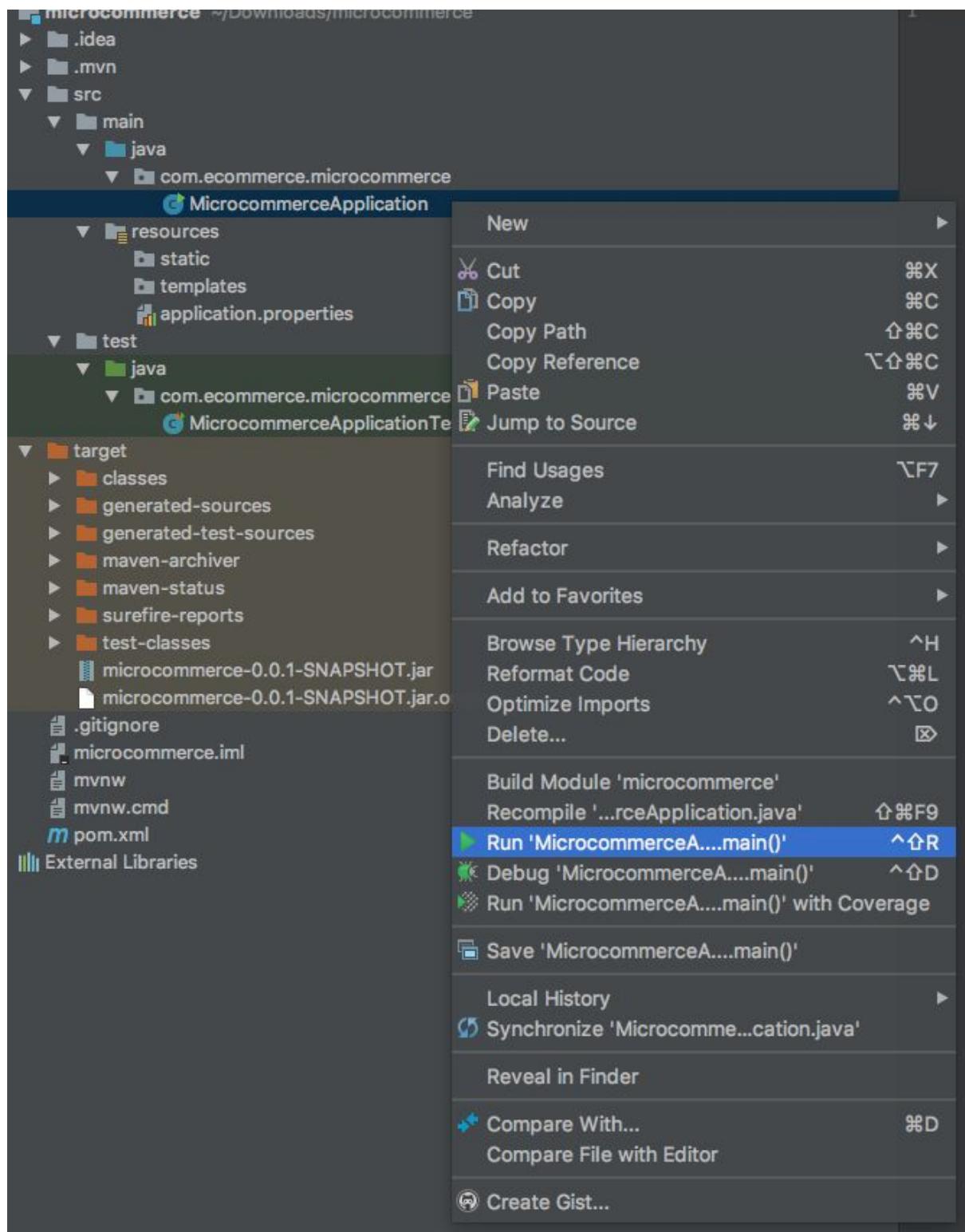
There was an unexpected error (type=Not Found, status=404).

No message available

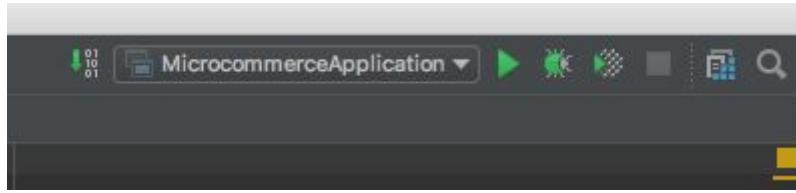
Message d'erreur au lancement de l'application

Pour faciliter l'exécution de notre service, vous pouvez profiter d'un raccourci d'IntelliJ. Pour cela, faites un clic droit sur le nom de la classe contenant la méthode main (*MicroserviceApplication.java*) puis cliquez sur "Run" dans le menu contextuel qui s'affiche. La figure suivante illustre cette opération :

qui s'affiche. La figure suivante illustre cette opération :



L'application se lance et vous pouvez avoir le retour dans la console intégrée en bas. Cette opération est à faire uniquement la première fois. Pour les fois suivantes, il suffira d'appuyer sur le bouton Play en haut à droite pour la démarrer et l'arrêter avec le bouton rouge :



N'oubliez pas, dans votre console, d'arrêter l'application afin de libérer le port pour une utilisation ultérieure.

Vous pouvez en profiter pour essayer la personnalisation de l'auto-configuration de Spring Boot via **application.properties**. Vous pouvez ainsi changer un [nombre impressionnant de paramètres](#) grâce à une simple ligne dans le fichier *application.properties*.

Changeons par exemple le port du serveur. Ajoutez tout simplement cette ligne :

```
server.port 9090
```

Exécutez maintenant l'application. Spring vous indique alors que l'application tourne désormais sur le port 9090 :

```
INFO : Mapped URL path //**/favicon.ico onto handler of type [org.springframework.web.servlet.handler.SimpleUrlHandlerMapping]
INFO : Registering beans for JMX exposure on startup
INFO : Tomcat started on port(s): 9090 (http)
INFO : Started MicrocommerceApplication in 5.595 seconds
```

Vous pouvez le vérifier en vous rendant également à l'url <http://localhost:9090>.

Créez l'API REST

Nous arrivons maintenant au cœur du Microservice que nous voulons développer. Ce Microservice va devoir être RESTful et donc pouvoir communiquer de cette manière.

Quels sont les besoins ?

Nous avons besoin d'un Microservice capable de gérer les produits. Pour cela, il doit pouvoir exposer une API REST qui propose toutes les **opérations CRUD** (Create, Read, Update, Delete).

Nous allons donc avoir :

- Une classe *Produit* qui représente les caractéristiques d'un produit (nom, prix, etc.) ;
- Un contrôleur qui s'occupera de répondre aux requêtes CRUD et de faire les opérations nécessaires.

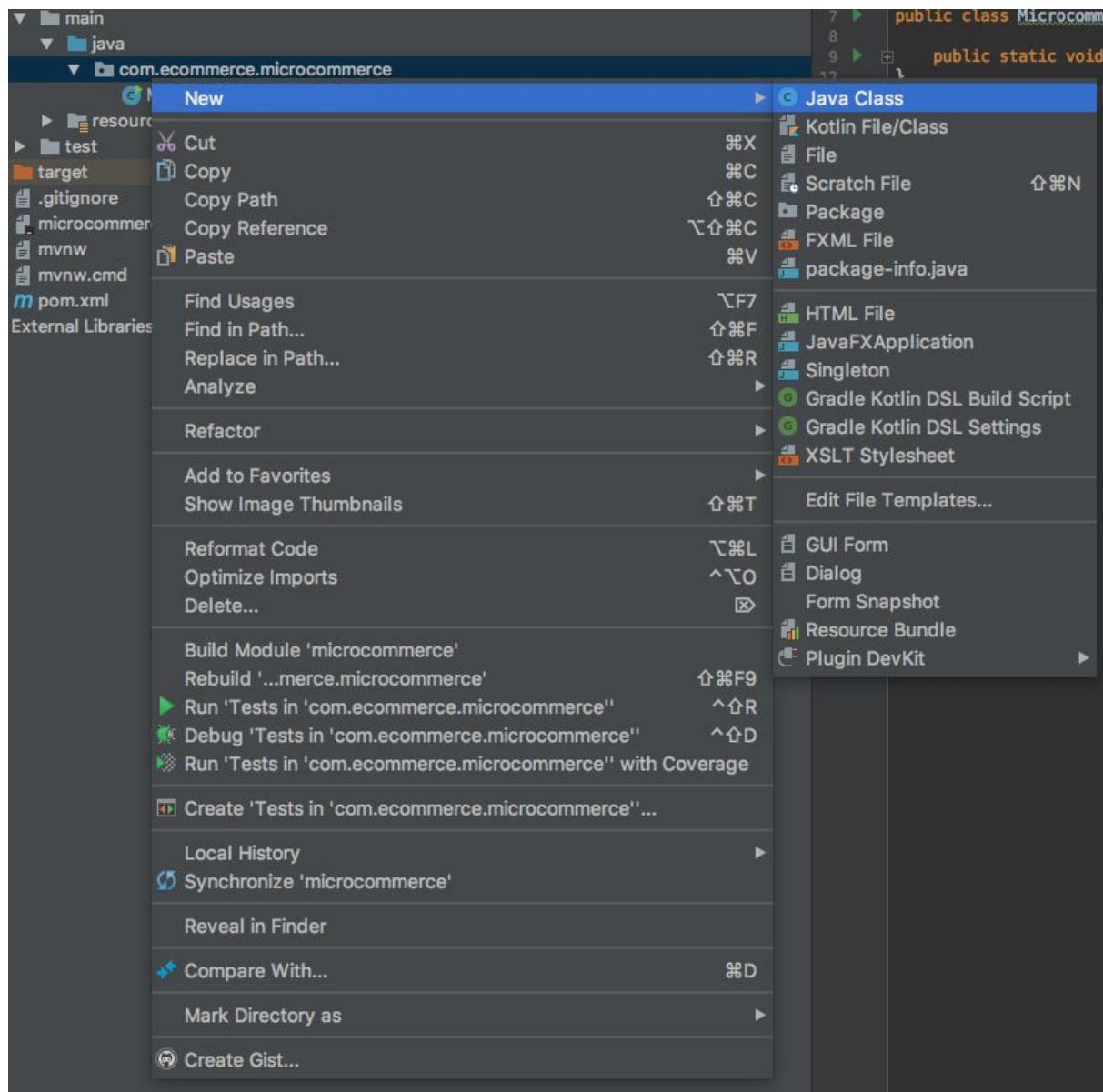
Nous voulons donc pouvoir appeler notre Microservice sur les URLs suivantes :

- Requête **GET** à **/Produits** : affiche la liste de tous les produits ;
- Requête **GET** à **/Produits/{id}** : affiche un produit par son Id ;
- Requête **PUT** à **/Produits/{id}** : met à jour un produit par son Id ;
- Requête **POST** à **/Produits** : ajoute un produit ;
- Requête **DELETE** à **/Produits/{id}** : supprime un produit par son Id.

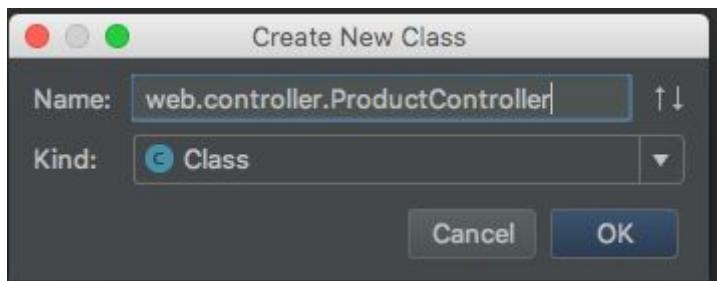
Passons au code !

Créez le contrôleur REST

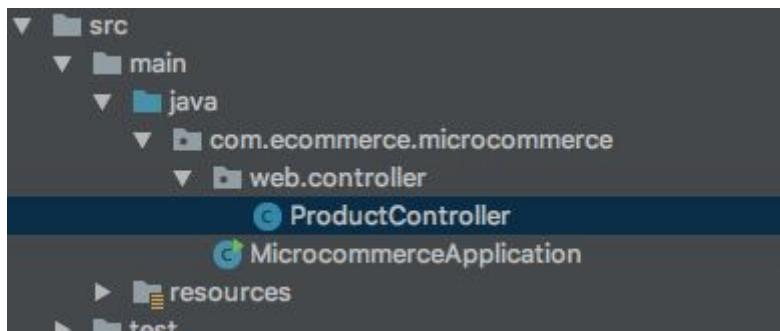
Nous allons créer un contrôleur et le placer dans un Package "controller", lui-même situé dans un package "web". Pour ce faire, faites un clic droit sur le package principal puis *New -> Java Class*:



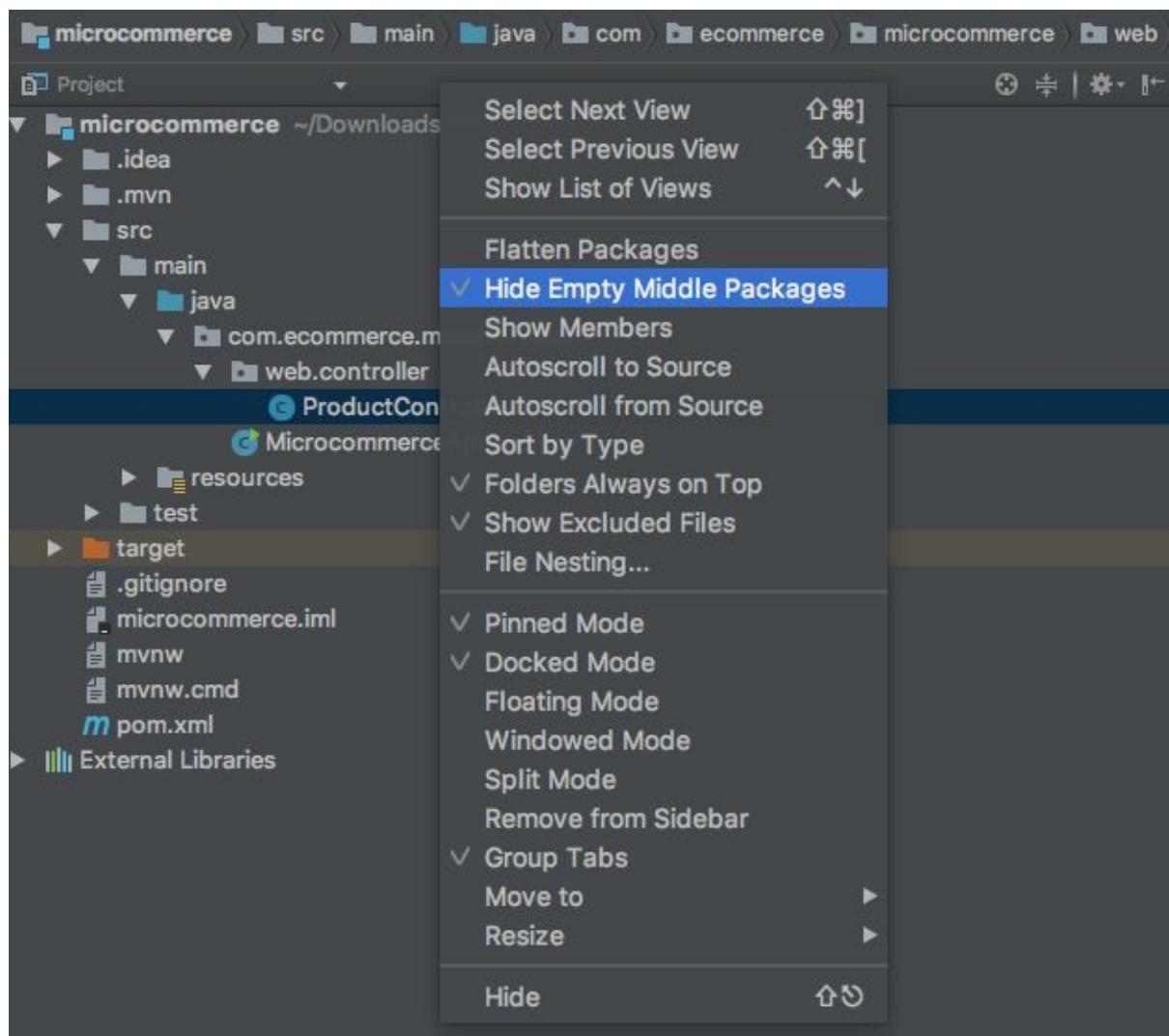
Pour créer les packages nécessaires et placer la classe directement à l'intérieur, nous allons utiliser une notation raccourcie fournie par *IntelliJ*. Dans la boîte de dialogue, écrivez **web.controller.ProductController**, comme illustré dans la figure suivante :



Lorsque vous cliquez sur OK, IntelliJ crée un package *web*, puis crée à l'intérieur de celui-ci un package *controller*. Enfin, la classe *ProductController* est créée à l'intérieur de ce dernier Package. Vous obtenez l'arborescence illustrée ci-après :



IntelliJ affiche les packages vides de façon compacte, comme vous pouvez le voir dans le cas du package *web* qui est fusionné avec *controller* : ***web.controller***. Si cette façon d'afficher les packages ne vous plaît pas, vous pouvez opter pour un affichage classique en faisant clic droit sur la barre du projet puis "Hide Empty Middle Packages" comme suit :



Nous allons commencer par indiquer à Spring que ce contrôleur est un contrôleur REST. Saisissez le code suivant dans la classe *ProductController*:

```
package com.ecommerce.microcommerce.web.controller;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ProductController {

}
```

Vous connaissez sans doute l'annotation `@Controller` de Spring qui permet de désigner une classe comme contrôleur, lui conférant la capacité de traiter les requêtes de type GET, POST, etc. Vous ajoutez ensuite `@ResponseBody` aux méthodes qui devront répondre directement sans passer par une vue.

`@RestController` est simplement la combinaison des deux annotations précédentes. Une fois ajouté, il indique que cette classe va pouvoir traiter les requêtes que nous allons définir. Il indique aussi que chaque méthode va renvoyer directement la réponse JSON à l'utilisateur, donc pas de vue dans le circuit.

Méthode pour GET /Produits

Commençons par créer une méthode `listeProduits`, très simple, qui retourne un `String`. Comme nous n'avons pas encore de produits, on retourne une simple phrase pour tester.

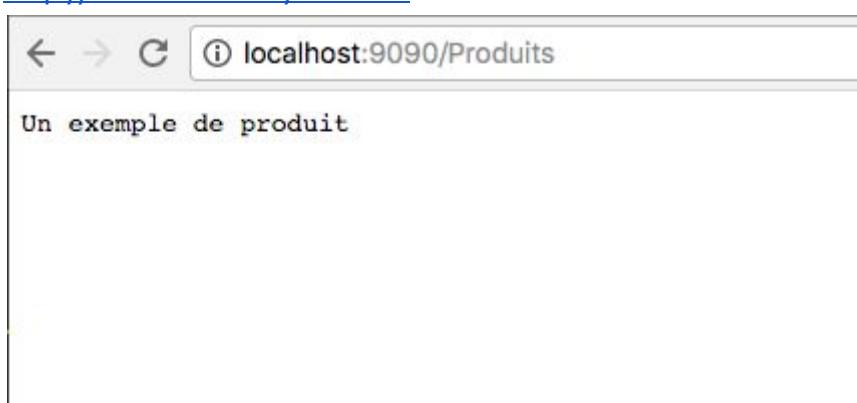
```
@RestController
public class ProductController {
    @RequestMapping(value="/Produits", method=RequestMethod.GET)
    public String listeProduits() {
        return "Un exemple de produit";
    }
}
```

Dans ce code, c'est l'annotation `@RequestMapping` qui permet de faire le lien entre l'URI `/Produits`, invoquée via GET, et la méthode `listeProduits`.

Cette annotation accepte [plusieurs paramètres](#), dont voici les principaux :

- **value** : C'est ici que vous indiquez l'URI à laquelle cette méthode doit répondre. Vous pouvez également indiquer des paramètres, nous y reviendrons.
- **method** : Vous indiquez ici à quel type de requêtes cette méthode doit répondre. Dans notre cas, notre méthode `listeProduits` ne sera déclenchée **que** si l'URI est exactement `/Produits` et que la requête est de type GET. Si vous appelez `/Produits` avec un POST, notre méthode ne sera tout simplement pas évoquée. `method` accepte toutes les requêtes CRUD et [plus encore](#).
- **produces** : Dans certains cas d'utilisations avancées, vous aurez besoin de préciser, par exemple, que votre méthode est capable de répondre en XML et en JSON. Cela entre aussi dans le choix de la méthode qui correspond le mieux à la requête. Si la requête contient du XML et que vous avez 2 méthodes identiques, dont une capable de produire du XML, c'est celle-ci qui sera appelée. Il en va de même pour **consumes** qui précise les formats acceptés. Dans la plupart des cas, vous n'avez pas besoin de renseigner ces paramètres.

Très bien ! Il ne reste plus qu'à lancer l'application et se rendre à <http://localhost:9090/Produits>.



Méthode pour GET /Produits/{id}

Créons maintenant une autre méthode capable d'accepter un Id de produit en paramètre :

```
@RequestMapping(value = "/Produits/{id}", method = RequestMethod.GET)
public String afficherUnProduit(@PathVariable int id) {
    return "Vous avez demandé un produit avec l'id " + id;
}
```

La première différence dans cette méthode est l'**ajout de {id} à l'URI**. Cette notation permet d'indiquer que cette méthode doit répondre uniquement aux requêtes avec une URI de type */Produits/25* par exemple. Comme nous avons indiqué que *id* doit être un *int* (dans `@PathVariable int id`), vous pouvez vous amuser à passer une chaîne de caractères à la place, vous verrez que Spring vous renverra une erreur.

Il existe des raccourcis pour éviter d'écrire l'annotation à rallonge `@RequestMapping`. Par exemple, pour une `@RequestMapping` qui accepte des requêtes de type GET, vous pouvez la remplacer par `@GetMapping`. Vous obtenez un code comme ceci :

```
@GetMapping(value = "/Produits/{id}")
public String afficherUnProduit(@PathVariable int id) {
    return "Vous avez demandé un produit avec l'id " + id;
}
```

Le code devient plus élégant et lisible quand vous avez une dizaine de méthodes par exemple. Vous avez aussi, les mêmes équivalents pour les autres types : `PostMapping`, `DeleteMapping`, etc.

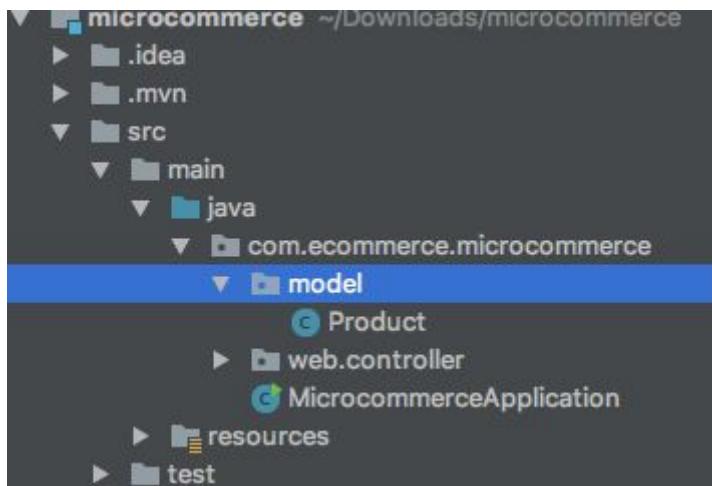
Tout cela est très bien, mais nous aimerions bien renvoyer une liste de vrais produits au format JSON et plus seulement des phrases inutiles. C'est ce que nous allons voir dans la prochaine section.

Renvoyez une réponse JSON

Nous allons commencer par créer une classe qui représente un produit. Cette classe est souvent appelée *Bean* ou *JavaBean*. **Un Bean est une classe classique** qui doit être "sérialisable" et avoir au minimum :

- un constructeur public sans argument,
- des getters et setters pour toutes les propriétés de la classe.

Commencez par créer une nouvelle classe `Product` que vous allez placer dans un package "model" sous le package *microcommerce* :



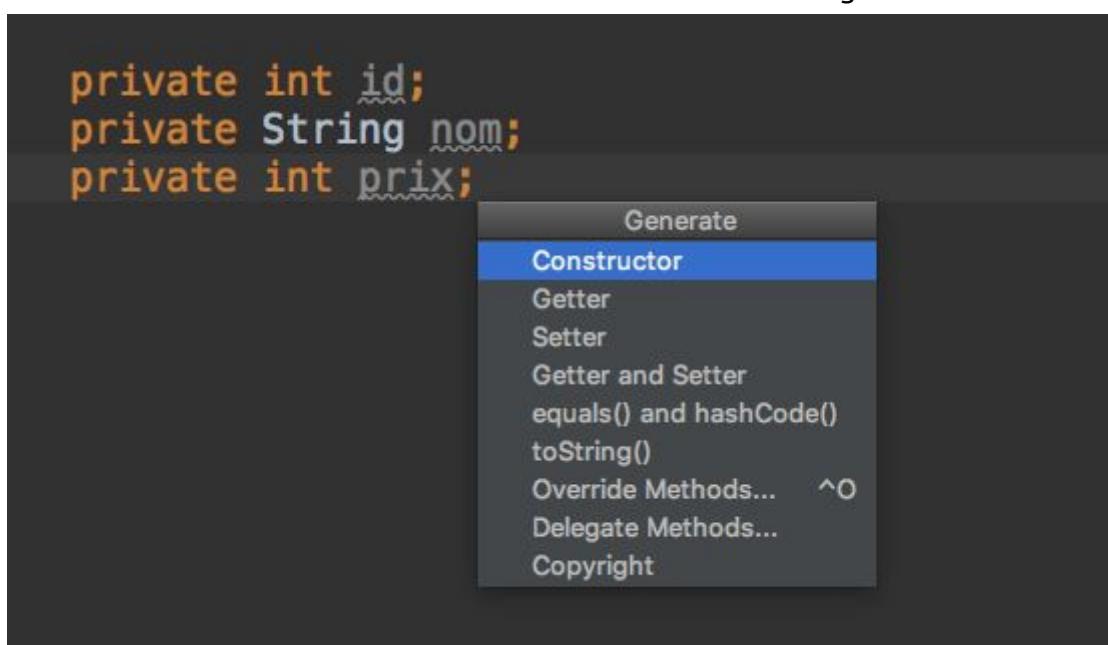
classe Product

Créez ensuite les propriétés de base de la classe :

```
package com.ecommerce.microcommerce.model;

public class Product {
    private int id;
    private String nom;
    private String nom;
}
```

Vous allez maintenant générer le constructeur et les getters et setters. Faites Cmd + N sous Mac ou Ctrl +N sous Windows afin d'afficher la fenêtre de génération de code :



Fenêtre de génération du code

Générez le constructeur sans argument puis les getters et setters pour toutes les propriétés ainsi que la méthode `toString`. Pour nos tests, nous allons ajouter un

constructeur afin d'obtenir des instances de produits pré-remplies avec des informations de tests. Vous obtenez ceci :

```
package com.ecommerce.microcommerce.model;
```

```
package com.ecommerce.microcommerce.model;
public class Product {
    private int id;
    private String nom;
    private int prix;

    //constructeur par défaut
    public Product() {
    }

    //constructeur pour nos tests

    public Product(int id, String nom, int prix) {
        this.id=id;
        this.nom=nom;
        this.prix=prix;
    }

    public int getId() {
        return id;
    }

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

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom=nom;
    }

    public int getPrix() {
        return prix;
    }

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

    @Override
    public String toString(){
        return "Product{"
            + "id=" + id +
            ", nom='"+ nom + '\'' +
            ", prix=" + prix+ '}';
    }
}
```



Très bien ! Maintenant à chaque fois que quelqu'un appelle notre URI "/Produits/{id}", nous voudrions renvoyer un produit au format JSON qui correspond à notre classe Product. Retournez sur le code précédent et remplacer la méthode *afficherUnProduit* par celle-ci :

```
//Récupérer un produit par son Id  
@RequestMapping(value="/Produits/{id}")  
public Product afficherUnProduit(@PathVariable int id) {  
    Product product=new Product(id, new String("Aspirateur"), 100 );  
    return product;  
}
```

Explications

Tout d'abord, nous avons indiqué que notre méthode va retourner un *Product* au lieu de *String*.

Normalement, nous sommes censés aller chercher le produit par l'id que nous avons reçu dans la base de données et le retourner à l'utilisateur. Comme nous n'avons pas encore de base de données, nous avons tout simplement instancié un objet *Product* grâce au constructeur que nous avons défini plus tôt. Nous simulons donc la récupération d'un produit dans la base de données.

Une fois notre instance de *Product* prête, nous la retournons.

Voici le contrôleur complet :

```
package com.ecommerce.microcommerce.web.controller;  
import com.ecommerce.microcommerce.model.Product;  
import org.springframework.web.bind.annotation.*;  
  
@RestController  
public class ProductController {  
  
    //Récupérer la liste des produits  
    @RequestMapping(value="/Produits", method=RequestMethod.GET)  
    public String listeProduits() {  
        return "Un exemple de produit";  
    }  
  
    //Récupérer un produit par son Id  
    @GetMapping(value="/Produits/{id}")  
    public Product afficherUnProduit(@PathVariable int id) {  
        Product product=new Product(id, new String("Aspirateur"), 100 );  
        return product;  
    }  
}
```

Lancez de nouveau le Microservice puis rendez-vous par exemple sur <http://localhost:9090/Produits/27> pour regarder ce qui se passe.



The screenshot shows a browser window with the URL `localhost:9090/Produits/27`. The page content is a JSON object with the following structure:

```
1 // 20180115190207
2 // http://localhost:9090/Produits/27
3
4 {
5     "id": 27,
6     "nom": "Aspirateur",
7     "prix": 100
8 }
```

Retour de l'API REST

Vous obtenez une belle réponse formatée en *JSON* comme par magie .

Comment est-ce possible ?

Vous avez indiqué au début que cette classe est un contrôleur *REST* grâce à l'annotation `@RestController`. Spring sait alors que les réponses aux requêtes qu'il vous passe devront être très probablement en format *JSON*.

L'auto-configurateur va alors chercher si vous avez dans votre classpath une dépendance capable de transformer un objet *Java* en *JSON*, et inversement. Bingo ! Il y a justement Jackson qui a été importé avec le starter que nous avons utilisé. Le *Bean Product* que nous renvoyons est donc transformé en *JSON* puis servi en réponse.

Vous venez donc de créer un **premier Microservice REST** sans avoir à manipuler *JSON* ni à parser les requêtes HTTP.

Si la réponse *JSON* n'est pas aussi présentable dans votre navigateur que celle de ma capture d'écran, je vous conseille d'installer une extension comme **JSON Viewer** pour chrome, afin de rendre tout cela plus lisible .

Créez le DAO

Nous allons nous rapprocher un peu plus d'un cas d'utilisation réel en créant la DAO nécessaire pour **communiquer avec une base de données**. Nous allons simuler celle-ci grâce à des données statiques.

Créez un package et nommez-le `dao` puis créez dedans une interface nommée `ProductDao`, dans laquelle vous allez déclarer les opérations que nous allons implémenter.

```
package com.ecommerce.microcommerce.dao;
import com.ecommerce.microcommerce.model.Product;
import java.util.List;

public interface ProductDao {
    public List<Product>findAll();
    public ProductfindById(int id);
    public Productsave(Product product);
}
```

Nous indiquons dans cette interface que les opérations suivantes sont possibles :

- findAll : renvoie la liste complète de tous les produits ;
- findById : renvoie un produit par son Id ;
- save : ajoute un produit.

Les noms des méthodes ne sont pas choisis au hasard. En effet, il faut suivre [les conventions citées ici par exemple](#) afin de bénéficier plus tard de certaines fonctionnalités qui vous feront gagner beaucoup de temps.

Maintenant que notre interface est prête, nous allons pouvoir créer son **implémentation**. Créez une classe *ProductDaoImpl* qui implémente l'interface que nous venons de créer. Générez ensuite automatiquement cette implémentation en appuyant sur Cmd +N sur Mac ou Ctrl +N sur Windows, puis *Implement Methods*. Vous obtenez alors ceci :

```
package com.ecommerce.microcommerce.dao;
import com.ecommerce.microcommerce.model.Product;
import java.util.List;

public class ProductDaoImpl implements ProductDao {

    @Override
    public List<Product>findAll() {
        return null;
    }

    @Override
    public ProductfindById(int id) {
        return null;
    }

    @Override
    public Productsave(Product product) {
        return null;
    }
}
```

Normalement , cette classe est censée communiquer avec la base de données pour récupérer les produits ou en ajouter. Nous allons simuler ce comportement en créant des Produits en dur dans le code.

Nous obtenons donc ceci :

```
package com.ecommerce.microcommerce.dao;
import com.ecommerce.microcommerce.model.Product;
import org.springframework.stereotype.Repository;
import java.util.ArrayList;
import java.util.List;

@Repository
public class ProductDaoImplementsProductDao {
    public static List<Product> products=newArrayList<>();
    static {
        products.add(newProduct(1, newString("Ordinateur portable"), 350));
        products.add(newProduct(2, newString("Aspirateur Robot"), 500));
        products.add(newProduct(3, newString("Table de Ping Pong"), 750));
    }

    @Override
    public List<Product>findAll() {
        return products;
    }

    @Override
    public ProductfindById(int id) {
        for (Product product : products) {
            if(product.getId() ==id){
                return product;
            }
        }
        return null;
    }

    @Override
    public Product save(Product product) {
        products.add(product);
        return product;
    }
}
```

Explication du code

@Repository : cette annotation est appliquée à la classe afin d'indiquer à *Spring* qu'il s'agit d'une classe qui gère les données, ce qui nous permettra de profiter de certaines fonctionnalités comme les translations des erreurs. Nous y reviendrons.

Pour tester avec des données statiques , vu que nous n'en sommes pas encore à la base de données, nous définissons ici un tableau de *Product* dans lequel nous ajoutons 3 produits statiques. Les méthodes suivantes sont ensuite redéfinies afin de renvoyer les données adéquates :

1. *findAll* : renvoie tous les produits que nous avons créés ;
2. *findById* : vérifie s'il y a un produit avec l'id donnée dans notre liste de produits et le renvoie en cas de correspondance ;
3. *save* : ajoute le produit reçu à notre liste.

Très bien, votre couche DAO est prête ! Vous allez pouvoir l'utiliser pour simuler la récupération et l'ajout de produits depuis une base de données.



Interagir avec les données

Nous allons maintenant modifier notre contrôleur afin qu'elle utilise notre **couche DAO** pour manipuler les produits.

Voici la classe modifiée :

```
package com.ecommerce.microcommerce.web.controller;
import com.ecommerce.microcommerce.dao.ProductDao;
import com.ecommerce.microcommerce.model.Product;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
public class ProductController {

    @Autowired
    private Product DaoproductDao;

    //Récupérer la liste des produits
    @RequestMapping(value="/Produits", method=RequestMethod.GET)
    public List<Product> listeProduits() {
        return productDao.findAll();
    }

    //Récupérer un produit par son Id
    @GetMapping(value="/Produits/{id}")
    public Product afficherUnProduit(@PathVariable int id) {
        return productDao.findById(id);
    }
}
```

Explication du code

Tout d'abord, nous avons créé une variable de type *ProductDao*, que nous avons annotée avec **@Autowired** afin que Spring se charge d'en fabriquer une instance. *ProductDao* a désormais accès à toutes les méthodes que nous avons définies.

Nous avons changé *listeProduits* afin qu'elle nous retourne une liste de produits *List*. Ensuite, il a suffi d'invoquer la méthode *findAll* créée précédemment pour qu'elle nous retourne tous les produits.

De même, pour *afficherUnProduit* qui fait appel à *findById*.

Testez !

Lancez le Microservice et rendez-vous à <http://localhost:9090/Produits/>, vous obtiendrez ceci :



```
← → ⌂ ⓘ localhost:9090/Produits/  
1 // 20180116024524  
2 // http://localhost:9090/Produits/  
3  
4 [  
5 {  
6   "id": 1,  
7   "nom": "Ordinateur portable",  
8   "prix": 350  
9 },  
10 {  
11   "id": 2,  
12   "nom": "Aspirateur Robot",  
13   "prix": 500  
14 },  
15 {  
16   "id": 3,  
17   "nom": "Table de Ping Pong",  
18   "prix": 750  
19 }  
20 ]
```

Résultat de l'appel de

l'API REST

Bingo ! Vous avez la liste des produits que vous avez définis au format JSON prête à être consommée par n'importe quel Microservice REST.

Rendez-vous ensuite à <http://localhost:9090/Produits/1> pour afficher produit par produit.

Le processus pour obtenir le résultat est le suivant :

1. L'utilisateur envoie une requête GET vers /Produits/2.
2. Le dispatcheur cherche dans votre contrôleur la méthode qui répond au pattern "/Produits/{id}" et l'exécute.
3. La méthode (dans ce cas *listeProduits*) fait appel au DAO pour qu'il communique avec la base de données. Il récupère les informations sur le produit puis il crée une instance de *Product* qu'il renvoie ensuite à votre méthode.

Votre méthode retourne l'instance reçue, qui est transformée à la volée en JSON grâce à Jackson.

Créez les Microservices e-commerce et leur client

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 **ultraminimaliste** 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 GIT** <https://github.com/marwensaid/mcommerce.git>

Pour chaque section du cours, vous aurez une branche dans le dépôt que je vous indiquerai.

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

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

Pour importer le projet suivez ces indications :

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/TestBranch -> 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.

Explorer l'application

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

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 que je vous donne durant le cours. Le premier projet que vous avez créé plus haut doit servir surtout à voir et à analyser la version du cours 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.

```
1  @RestController
2  public class ProductController {
3
4      @Autowired
5      ProductDao productDao;
6
7      // Affiche la liste de tous les produits disponibles
8      @GetMapping(value = "/Produits")
9      public List<Product> listeDesProduits(){
10
11         List<Product> products = productDao.findAll();
12
13         if(products.isEmpty()) throw new ProductNotFoundException("Aucun produit n'est disponible à
la vente");
14
15         return products;
16
17     }
18
19     //Récuperer un produit par son id
20     @GetMapping( value = "/Produits/{id}")
21     public Optional<Product> recupererUnProduit(@PathVariable int id) {
22
23         Optional<Product> product = productDao.findById(id);
24
25         if(!product.isPresent()) throw new ProductNotFoundException("Le produit correspondant à l'id
" + id + " n'existe pas");
26
27         return product;
28     }
29 }
```

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 :



```

1  @RestController
2  public class CommandeController {
3
4      @Autowired
5      CommandesDao commandesDao;
6
7      @PostMapping (value = "/commandes")
8      public ResponseEntity<Commande> ajouterCommande(@RequestBody Commande commande){
9
10         Commande nouvelleCommande = commandesDao.save(commande);
11
12         if(nouvelleCommande == null) throw new ImpossibleAjouterCommandeException("Impossible
d'ajouter cette commande");
13
14         return new ResponseEntity<Commande>(commande, HttpStatus.CREATED);
15     }
16
17     @GetMapping(value = "/commandes/{id}")
18     public Optional<Commande> recupererUneCommande(@PathVariable int id){
19
20         Optional<Commande> commande = commandesDao.findById(id);
21
22         if(!commande.isPresent()) throw new CommandeNotFoundException("Cette commande n'existe pas");
23
24         return commande;
25     }
26 }
```

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`. Vous pouvez [en savoir plus ici](#). `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 :

```

1  @RestController
2  public class PaiementController {
3
4      @Autowired
5      PaiementDao paiementDao;
6
7      @PostMapping(value = "/paiement")
8      public ResponseEntity<Paiement> payerUneCommande(@RequestBody Paiement paiement){
9
10
11         //Vérifions s'il y a déjà un paiement enregistré pour cette commande
12         Paiement paiementExistant = paiementDao.findByIdCommande(paiement.getIdCommande());
13         if(paiementExistant != null) throw new PaiementExistantException("Cette commande est déjà
14             payée");
15
16         //Enregistrer le paiement
17         Paiement nouveauPaiement = paiementDao.save(paiement);
18
19         if(nouveauPaiement == null) throw new PaiementImpossibleException("Erreur, impossible
d'établir le paiement, réessayez plus tard");
20
21
22         //TODO Nous allons appeler le Microservice Commandes ici pour lui signifier que le paiement
est accepté
23
24         return new ResponseEntity<Paiement>(nouveauPaiement, HttpStatus.CREATED);
25
26     }
27 }

```

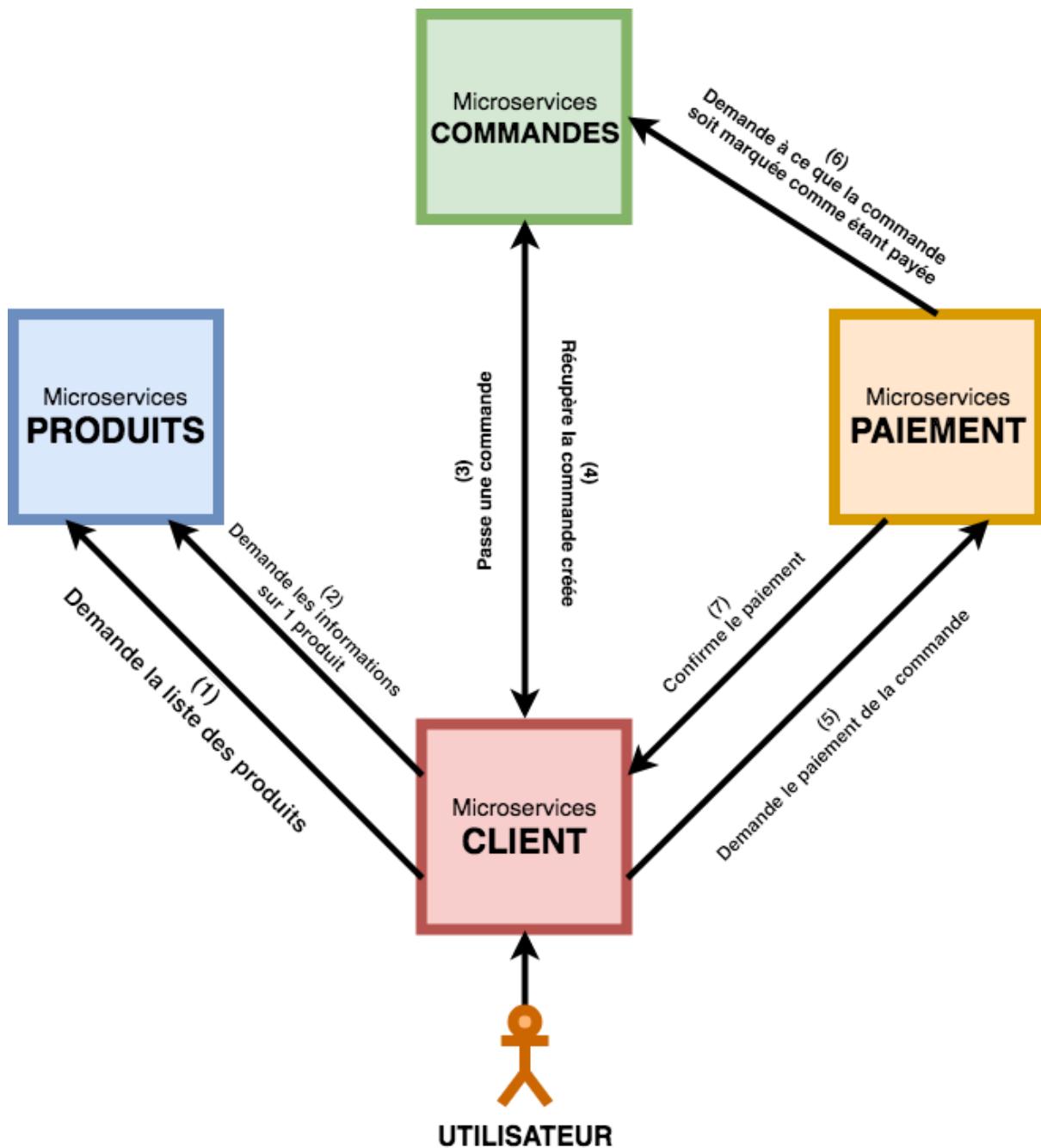
Il dispose d'une seule opération : `payerUneCommande`. Une commande ne peut être payée qu'une seule fois, on vérifie alors si le paiement à effectuer n'a pas déjà eu lieu. Pour cela, nous devons chercher s'il y a un paiement correspondant à l'id de commande `idCommande`. On crée donc une méthode dans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

```
1 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.

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 que vous pouvez voir ou revoir dans le cours précédent sur le Microservice.

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".

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 :

```

1 package com.clientui.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.RequestMapping;
6
7
8 @Controller
9 public class ClientController {
10
11     @RequestMapping("/")
12     public String accueil(Model model){
13
14         return "Accueil";
15     }
16
17 }
```

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`:

```

1 <dependency>
2   <groupId>org.webjars</groupId>
3   <artifactId>bootstrap</artifactId>
4   <version>4.0.0-2</version>
5 </dependency>
```

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

À noter qu'il est possible d'utiliser la version hébergée par Google par exemple, mais comme nous souhaitons que notre Microservice soit autonome, nous allons garder les fichiers nécessaires en local.

Ajoutez ensuite ce contenu HTML à `Accueil.html`:

```

1 <!DOCTYPE HTML>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4   <title>Mcommerce</title>
5
6   <link rel="stylesheet" type="text/css" href="webjars/bootstrap/4.0.0-2/css/bootstrap.min.css" />
7
8 </head>
9 <body>
10
11
12 <div class="container">
13
14   <h1>Application Mcommerce</h1>
15
16 </div>
17
18 <script type="text/javascript" src="webjars/bootstrap/4.0.0-2/js/bootstrap.min.js"></script>
19
20
21 </body>
22 </html>
```

Relancez le client et vérifiez que tout fonctionne correctement.

La branche pour récupérer le projet avec ce squelette de client est [SqueletteClient](#).



Faites communiquer vos 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>

```

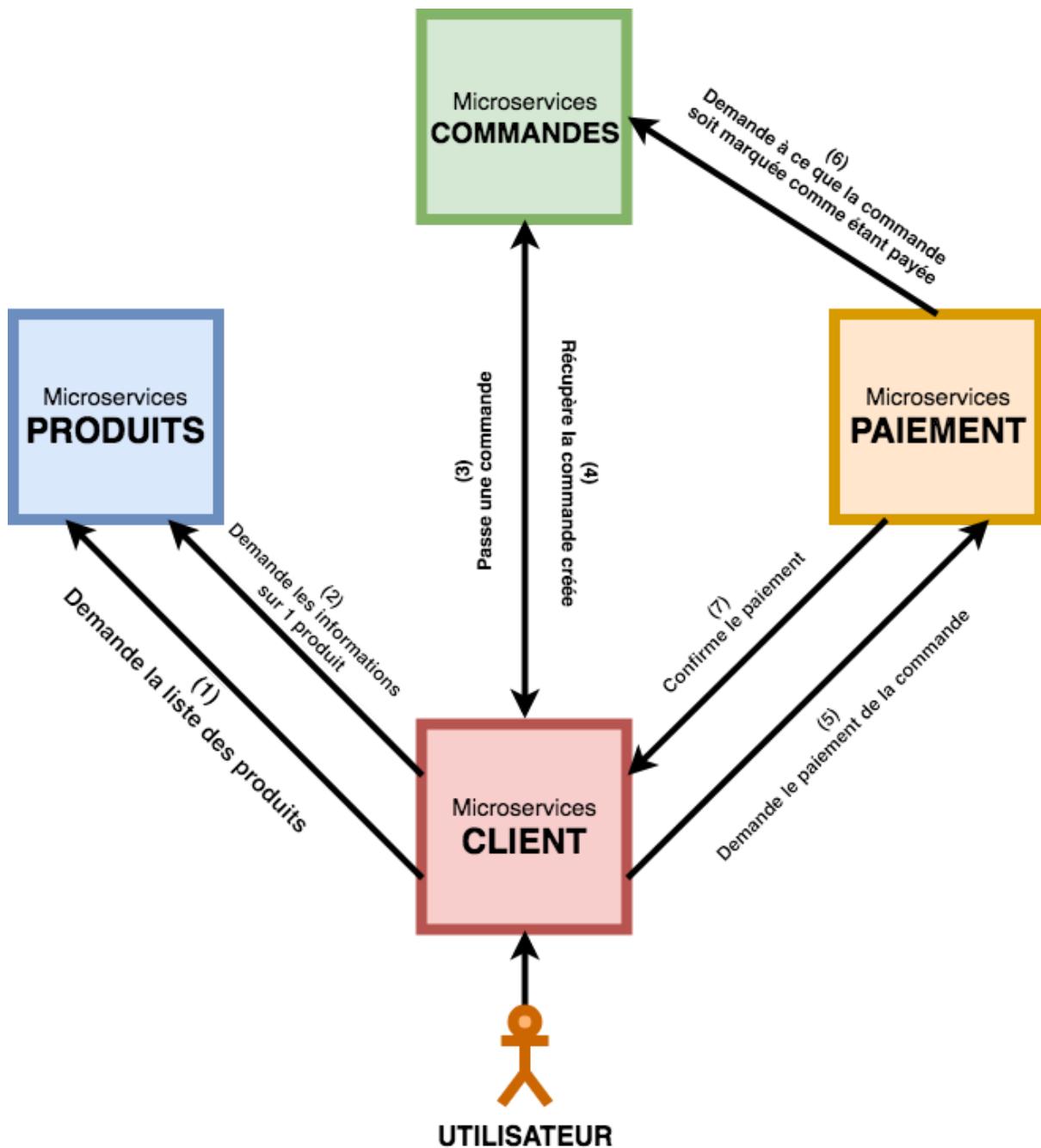


Afin d'activer Feign dans ce Microservice, rendez-vous à *ClientUiApplication* et ajoutez l'annotation **@EnableFeignClients** :

```
1 @SpringBootApplication
2 @EnableFeignClients("com.clientui")
3 public class ClientUiApplication {
4
5     public static void main(String[] args) {
6         SpringApplication.run(ClientUiApplication.class, args);
7     }
8 }
```

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 :



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` :

```
1 package com.clientui.beans;
2
3
4 public class ProductBean {
5
6     private int id;
7
8     private String titre;
9
10    private String description;
11
12    private String image;
13
14    private Double prix;
15
16    public ProductBean() {
17    }
18
19    public int getId() {
20        return id;
21    }
22
23    public void setId(int id) {
24        this.id = id;
25    }
26
27    public String getTitre() {
28        return titre;
29    }
30
31    public void setTitre(String titre) {
32        this.titre = titre;
33    }
34
35    public String getDescription() {
36        return description;
37    }
38
```



```

39  public void setDescription(String description) {
40      this.description = description;
41  }
42
43  public String getImage() {
44      return image;
45  }
46
47  public void setImage(String image) {
48      this.image = image;
49  }
50
51  public Double getPrix() {
52      return prix;
53  }
54
55  public void setPrix(Double prix) {
56      this.prix = prix;
57  }
58
59  @Override
60  public String toString() {
61      return "ProductBean{" +
62          "id=" + id +
63          ", titre='" + titre + '\'' +
64          ", description='" + description + '\'' +
65          ", image='" + image + '\'' +
66          ", prix=" + prix +
67          '}';
68  }
69 }
```

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 :

```

1 package com.clientui.proxies;
2
3 import com.clientui.beans.ProductBean;
4 import org.springframework.cloud.openfeign.FeignClient;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.PathVariable;
7
8 import java.util.List;
9 import java.util.Optional;
10
11 @FeignClient(name = "microservice-produits", url = "localhost:9001")
12 public interface MicroserviceProduitsProxy {
13
14     @GetMapping(value = "/Produits")
15     List<ProductBean> listeDesProduits();
16
17     @GetMapping( value = "/Produits/{id}")
18     ProductBean recupererUnProduit(@PathVariable("id") int id);
19
20 }

```

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 :

```

1 spring.application.name=microservice-produits
2
3 server.port 9001
4
5 #Configurations H2
6 spring.jpa.show-sql=true
7 spring.h2.console.enabled=true
8
9 #défini l'encodage pour data.sql
10 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 :**

```
1 @Controller
2 public class ClientController {
3
4     @Autowired
5     private MicroserviceProduitsProxy ProduitsProxy;
6
7     @RequestMapping("/")
8     public String accueil(Model model){
9
10        List<ProductBean> produits = ProduitsProxy.listeDesProduits();
11
12        model.addAttribute("produits", produits);
13
14        return "Accueil";
15    }
16}
17 }
```

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* :



```

1 <!DOCTYPE HTML>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4     <title>Mcommerce</title>
5
6     <link rel="stylesheet" type="text/css" href="webjars/bootstrap/4.0.0-2/css/bootstrap.min.css" />
7
8 </head>
9 <body>
10
11
12 <div class="container">
13
14     <h1>Application Mcommerce</h1>
15
16     <div class="row">
17         <div th:each="produit : ${produits}" class="col-md-4 my-1">
18             <a th:href="@{/details-produit/${produit.id}}|}" >
19                 
20                 <p th:text= "${produit.titre}"></p>
21             </a>
22         </div>
23     </div>
24
25 </div>
26
27
28 <script type="text/javascript" src="webjars/bootstrap/4.0.0-2/js/bootstrap.min.js"></script>
29
30 </body>
31 </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*.

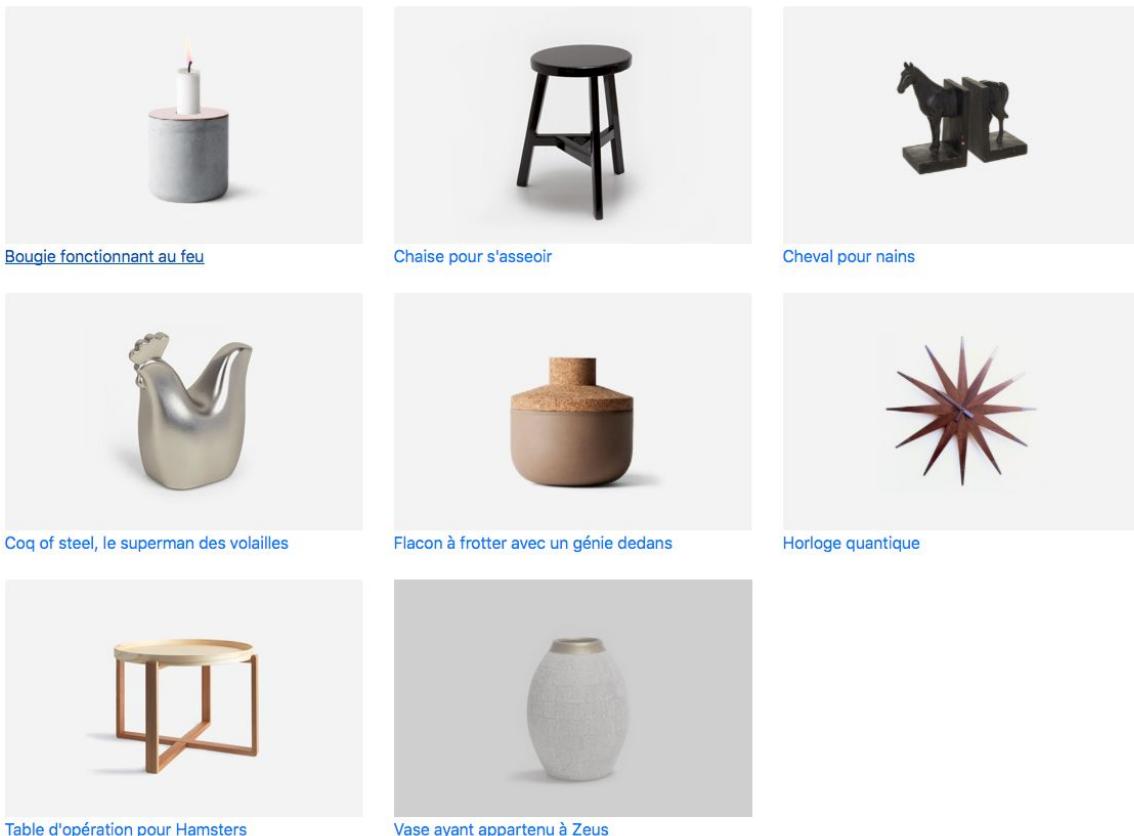
On a accès ensuite aux attributs de chaque objet pour créer le lien vers chaque produit sous le format "*/details-produit/id_produit_ici*". Nous créerons ensuite la méthode nécessaire pour cet URI dans notre contrôleur.

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 :

Application Mcommerce



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.

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. Essayez de le faire avant de lire la suite ! 😊

Nous avons défini l'URL vers chaque produit dans le HTML grâce à :

```
1 <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

```
1  @RequestMapping("/details-produit/{id}")
2  public String ficheProduit(@PathVariable int id, Model model){
3
4      ProductBean produit = ProduitsProxy.recupererUnProduit(id);
5
6      model.addAttribute("produit", produit);
7
8      return "FicheProduit";
9  }
```

Explications :

Nous récupérons donc l'id passée 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 :



```

1  <!DOCTYPE HTML>
2  <html xmlns:th="http://www.thymeleaf.org">
3  <head>
4      <title>Mcommerce</title>
5
6      <link rel="stylesheet" type="text/css"
7          href="http://localhost:8080/webjars/bootstrap/4.0.0-2/css/bootstrap.min.css"/>
8
9  </head>
10 <body>
11
12
13 <div class="container">
14
15     <h1 class="text-center">Application Mcommerce</h1>
16
17     <div class="row">
18         <div class="col-md-4 mx-auto mt-5 text-center">
19
20             
21
22             <p th:text="${produit.titre}" class="font-weight-bold"></p>
23
24             <p th:text="${produit.description}"></p>
25
26             <p>
27                 <a th:href="@{/details-produit/commander-produit/${produit.id}}/" class="font-
weight-bold">COMMANDER</a>
28             </p>
29
30         </div>
31     </div>
32
33 </div>
34
35
36 <script type="text/javascript" src="http://localhost:8080/webjars/bootstrap/4.0.0-2
/ja/bootstrap.min.js"></script>
```

xpliations :

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 :

Application Mcommerce



Chaise pour s'assoiré

Chaise rare avec non pas 1 ni 2 mais 3 pieds

[COMMANDER](#)

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.

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 :**

```
1 @ResponseStatus(HttpStatus.BAD_REQUEST)
2 public class ProductNotFoundException extends RuntimeException {
3
4
5     public ProductNotFoundException(String message) {
6         super(message);
7     }
8 }
9
10
```

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 :

```
1 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 :

```
1 package com.clientui.exceptions;
2
3 import feign.Response;
4 import feign.codec.ErrorDecoder;
5
6
7 public class CustomErrorDecoder implements ErrorDecoder {
8
9     private final ErrorDecoder defaultErrorDecoder = new Default();
10
11     @Override
12     public Exception decode(String invoqueur, Response reponse) {
13
14         if(reponse.status() == 400 ) {
15             return new ProductBadRequestException(
16                 "Requête incorrecte "
17             );
18         }
19
20         return defaultErrorDecoder.decode(invoqueur, reponse);
21     }
22
23 }
24
```

Explications :

- On hérite de ErrorDecoder.
- On récupère une instance de ErrorDecoder . Si nous ne sommes pas en 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 :

```

1 package com.clientui.exceptions;
2
3 import org.springframework.http.HttpStatus;
4 import org.springframework.web.bind.annotation.ResponseStatus;
5
6 @ResponseStatus(HttpStatus.BAD_REQUEST)
7 public class ProductBadRequestException extends RuntimeException{
8
9     public ProductBadRequestException(String message) {
10         super(message);
11     }
12 }
13

```

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 :

```

1 package com.clientui.configuration;
2
3 import com.clientui.exceptions.CustomErrorDecoder;
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.Configuration;
6
7 @Configuration
8 public class FeignExceptionConfig {
9
10     @Bean
11     public CustomErrorDecoder mCustomErrorDecoder(){
12         return new CustomErrorDecoder();
13     }
14 }
15

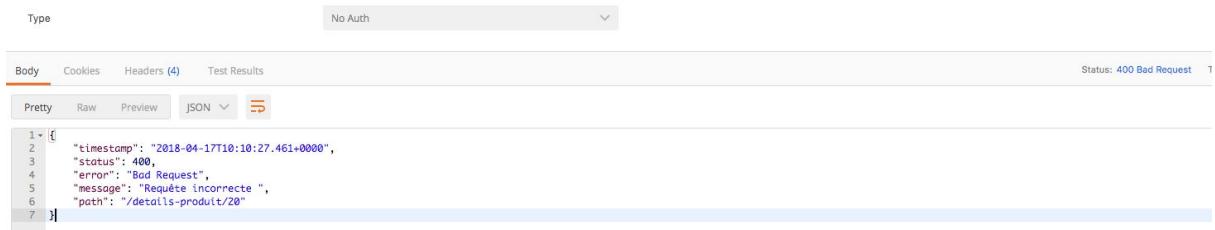
```

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 :



The screenshot shows a REST client interface with the following details:

- Type: No Auth
- Body tab selected.
- Headers (4) tab.
- Test Results tab.
- Status: 400 Bad Request
- Content type: application/json
- Pretty JSON response:

```
1 {  
2     "timestamp": "2018-04-17T10:10:27.461+0000",  
3     "status": 400,  
4     "error": "Bad Request",  
5     "message": "Requête incorrecte ",  
6     "path": "/details-produit/20"  
7 }
```

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 :

```
1 else if(reponse.status() > 400 && reponse.status() <=499 ) {  
2     return new Product4XXException(  
3             "Erreur de au format 4XX "  
4     );  
5 }
```

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 :

```
1 @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 :



```

1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <title>Commerce</title>
5     <link rel="stylesheet" type="text/css"
6       href="http://localhost:8080/webjars/bootstrap/4.0.0-2/css/bootstrap.min.css"/>
7   </head>
8   <body>
9     <div class="container">
10       <h1 class="text-center">Application Mcommerce</h1>
11       <div class="row">
12         <div class="col-md-4 mx-auto mt-5 text-center">
13           <img src="" class="card-img-top"/>
14           <p class="font-weight-bold"></p>
15           <p></p>
16           <a href="/commander-produit/0/null" class="font-weight-bold">COMMANDER</a>
17         </div>
18       </div>
19     </div>
20     <script type="text/javascript" src="http://localhost:8080/webjars/bootstrap/4.0.0-2/js/bootstrap.min.js"></script>
21   </body>
22 </html>

```

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 cours, 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 :

```

1 package com.clientui.exceptions;
2
3 import feign.Response;
4 import feign.codec.ErrorDecoder;
5
6
7 public class CustomErrorDecoder implements ErrorDecoder {
8
9     private final ErrorDecoder defaultErrorDecoder = new Default();
10
11    @Override
12    public Exception decode(String invoqueur, Response reponse) {
13
14        if(reponse.status() == 400 ) {
15            return new ProductBadRequestException(
16                "Requête incorrecte "
17            );
18        }
19
20        else if (reponse.status() == 404 ) {
21            return new ProductNotFoundException(
22                "Produit non trouvé "
23            );
24        }
25
26        return defaultErrorDecoder.decode(invoqueur, reponse);
27    }
28
29 }
30

```

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`.