

# Formation JAVA – Java Spring Boot et Spring Data

El Hadji Gaye

---

**Auteur** El Hadji Gaye

**Pour** Formations

**Date** 16/04/2021

---

**Objet** Java Spring-Boot et Spring Data

---

<b>I)</b>	<b>Vocabulaire .....</b>	<b>3</b>
<b>II)</b>	<b>Présentation de Spring Data .....</b>	<b>5</b>
1.	Spring Data Commons .....	6
a.	Les interfaces Spring Data Commons.....	7
b.	Exemples de requetes.....	8
2.	Spring Data JPA .....	10
a.	Requêtes personnalisées .....	11
b.	QueryDsl.....	12
<b>III)</b>	<b>Présentation de Spring Boot .....</b>	<b>13</b>
	L'auto-configuration .....	13
	Les Starters .....	14
	En résumé.....	18
<b>IV)</b>	<b>Création et initialisation d'un projet Spring Boot avec spring.io .....</b>	<b>19</b>
1.	Modification éventuelle du fichier pom.xml .....	24
2.	Modification de nom de classe.....	26
3.	Modification du fichier application.properties.....	27
4.	La classe MyEntity .....	29
5.	Suppression de la classe de test unitaire .....	31
6.	La base de données .....	32
7.	L'interface MyEntityRepository.....	33
8.	La classe MyEntityController.....	34
9.	Test de l'application.....	36
9.	Mise en place de la documentation Swagger.....	38
10.	Construire une image Docker complète avec MYSQL 8 + JAVA + MAVEN + Jar du Micro Service (Voir le cours sur Docker) .....	40
<b>V)</b>	<b>Exercices d'application maven-gestion-personnes-spring-boot .....</b>	<b>41</b>
1.	Créer le projet Maven Application jar maven-gestion-personnes-spring-boot .....	41
2.	Créer la base de donnée base_personnes avec le script script_base_personnes.sql .....	41
3.	Mettre à jour le fichier application.properties .....	41
4.	Créer l'entité com.cours.entities.Personne image de la table Personne.....	41
5.	Créer l'interface com.cours.repository.PersonneRepository .....	41
6.	Créer la classe com.cours.controller.PersonneController avec son CRUD au complet.....	41
7.	Mettre en place la documentation Swagger.....	41
8.	Construire une image Docker complète avec MYSQL 8 + JAVA + MAVEN + Jar du Micro Service.....	41
<b>VI)</b>	<b>Exercices d'application maven-gestion-personnes-jersey-spring-boot.....</b>	<b>42</b>
1.	Créer le projet Maven Application jar maven-gestion-personnes-jersey-spring-boot .....	42
2.	Créer la base de donnée base_personnes avec le script script_base_personnes.sql .....	44
3.	Mettre à jour le fichier application.properties .....	44
4.	Créer l'entité com.cours.entities.Personne image de la table Personne.....	44
5.	Créer l'interface com.cours.repository.PersonneRepository .....	44
6.	Créer la classe com.cours.rest.config.JerseyConfig.....	44
7.	Créer la classe com.cours.exception.MyRestException pour gérer les exception Rest.....	44
8.	Créer la classe com.cours.rest.GestionPersonnesJerseyResource avec son CRUD au complet .....	44
9.	Mettre en place la documentation Swagger.....	44
10.	Construire une image Docker complète avec MYSQL 8 + JAVA + MAVEN + Jar du Micro Service.....	44

## I) Vocabulaire

- **API** : Signifie Application Programming Interface. Ce qui veut dire que c'est un ensemble de bibliothèques et librairies dédié pour implémenter une fonctionnalité donnée.
- **ORM**: Object-Relational Mapping (**MOR** : Mapping Objet-Relationnel en français) est une technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé.
- **JPA** : Java Persistence API (abrégée en JPA), est une interface de programmation Java permettant aux développeurs d'organiser des données relationnelles dans des applications utilisant la plateforme Java.
- **JPQL** : Le langage JPQL (**Java Persistence Query Language**) est un langage de requête orienté objet, similaire à SQL, mais au lieu d'opérer sur les tables et colonnes, JPQL travaille avec des objets persistants et de leurs propriétés. Il est très proche du langage SQL dont il s'inspire fortement mais offre une approche objet. La grammaire de ce langage est définie par la spécification J.P.A.
- **HQL** : **Hibernate Query Language** est aussi un langage de requête orienté objet au même titre que JPQL. La principale différence avec le langage JQL est que le en HQL le « **Select** » sur l'objet n'est pas nécessaire. En fin de compte pour le JPQL on aura : **Select person from Personne person** alors que pour le HQL on aura : **from Personne**.
- **Bean** : le « **Bean** » (ou haricot en français) est une technologie de composants logiciels écrits en langage Java. Les **Beans** sont utilisés pour encapsuler plusieurs objets dans un seul objet. Le « **Bean** » regroupe alors tous les attributs des objets encapsulés. Ainsi, il représente une entité plus globale que les objets encapsulés de manière à répondre à un besoin métier.
- **Pattern IoC** : L'inversion de contrôle (inversion of control, IoC) est un patron d'architecture commun à tous les Frameworks (ou cadre de développement et d'exécution). Il fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du Framework ou de la couche logicielle sous-jacente. En effet selon un problème, il existe différentes formes, ou représentation d'IoC, le plus connu étant l'injection de dépendances (dependency injection) qui est un patron de conception permettant, en programmation orientée objet, de découpler les dépendances entre objets.
- **Pattern AOP** : L'AOP (Aspect Oriented Programming) ou POA (Programmation Orientée Aspect) est un paradigme de programmation ayant pour but de compléter la programmation orientée objet et permettre d'implémenter de façon plus propre les problématiques transverses à l'application. En effet, elle permet de factoriser du code dans des greffons et de les injecter en divers endroits sans pour autant modifier le code source des endroits en question.

- **GemFire** : GemFire est une infrastructure de gestion de données distribuée hautes performances qui se situe entre le cluster d'applications et les sources de données back-end. Avec GemFire, les données peuvent être gérées en mémoire, ce qui accélère l'accès.

## II) Présentation de Spring Data

Spring Data est un projet supplémentaire de Spring créé il y a quelques années pour répondre aux besoins d'écrire plus simplement l'accès aux données et d'avoir une couche d'abstraction commune à de multiples sources de données.

Les applications peuvent avoir besoin d'une base Neo4j et en même temps être couplées à Oracle (ou seulement à la base Oracle d'ailleurs), obligeant ainsi le développeur à adapter les modèles de données aux API de chacune des bases.

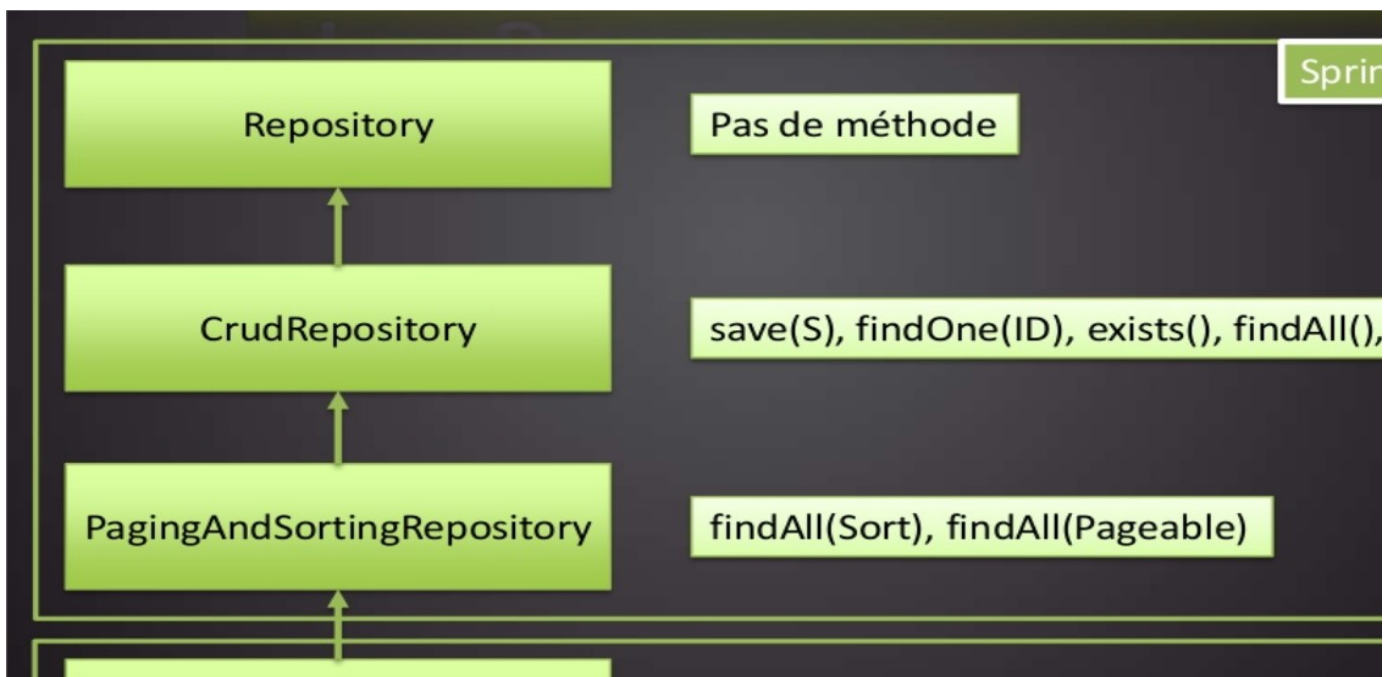
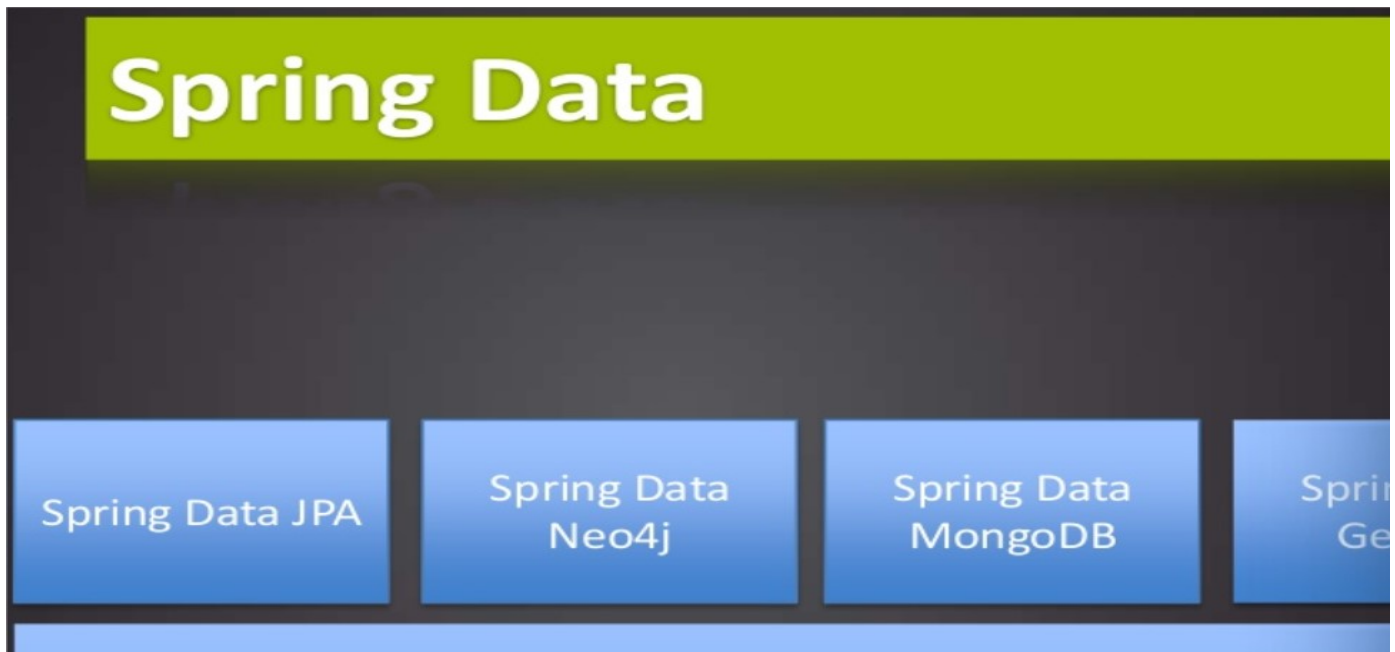
Se greffant par dessus celles-ci en implémentant les opérations de base et le design pattern Repository du DDD, Spring Data offre une API qui fait abstraction de ces API "bas niveau" (tout en prenant en compte les spécificités de chacune d'elles). Par exemple, pour bénéficier d'opérations CRUD de base, il suffit tout simplement d'étendre une interface de Spring Data pour y avoir accès.

Spring Data s'interface avec plusieurs sources de données parmi lesquelles JPA, Neo4j, MongoDB, GemFire, Hadoop, Elasticsearch, REST, Redis, Couchbase et quelques autres.

## 1. Spring Data Commons

Spring Data est découpé en une couche commune à toutes les sources de données sous-jacentes (Neo4j, MongoDB, JPA, ...), appelée Spring Data Commons, à laquelle s'ajoute une couche propre à la source de données. Donc si vous utilisez Spring Data Elasticsearch, vous aurez Spring Data Commons et Spring Data Elasticsearch.

Spring Data Commons contient les classes pour fonctionner, mais surtout les interfaces que l'utilisateur aura à étendre (directement ou pas) : **Repository**, **CrudRepository** et **PagingAndSortingRepository**.



### a. Les interfaces Spring Data Commons

En fonction des méthodes de base que le développeur voudra utiliser, ce dernier étendra une des trois interfaces. **Repository** ne sert qu'à dire que l'interface qui l'étend est un repository. Si l'utilisateur étend **CrudRepository**, il aura des méthodes CRUD pour la source de données sous-jacente (save, findOne, findAll, etc.). Si au lieu d'étendre **CrudRepository**, son interface étend **PagingAndSortingRepository** (cette dernière étend **CrudRepository** qui étend **Repository**), il aura accès, en plus des méthodes CRUD, à un ensemble de méthodes permettant de faire de la pagination et du tri. Ces interfaces prennent deux types paramétrés : le type de l'entité que l'on manipule et le type de l'identifiant de l'entité.

- **CrudRepository**

```
public interface PersonneRepository extends CrudRepository<Personne, Long> {}
```

```
public interface PersonneDao extends CrudRepository<Personne, Long> {}
```



Nom de l'entité qu'on manipule

- **PagingAndSortingRepository**

```
public interface PersonnePaginationRep extends PagingAndSortingRepository<Personne, Long> {}
```

## b. Exemples de requetes

Spring Data permet donc d'écrire des requêtes à partir des noms de méthode : vous écrivez la méthode avec certains mots-clés (And, Or, Containing, StartingWith, etc., définis dans les documentations de référence, par exemple celle-là), et il se charge de traduire ce nom de méthode en requête puis de l'exécuter au moment voulu, comme un grand :

```
public interface PersonneRepository extends CrudRepository<Personne, Long> {  
    // recherche une personne par son attribut "nom"  
    Personne findByNom(String nom);  
  
    // ici, par son "nom" ou "prenom"  
    Personne findByNomOrPrenom(String nom, String prenom);  
  
    List<Personne> findByNomAndPrenomAllIgnoreCase(String nom, String prenom);  
  
    List<Personne> findByNomOrderByPrenomAsc(String nom)  
}
```

Certains types sont reconnus par Spring Data, comme ceux de la pagination et du tri. Du coup, vous pouvez écrire ce genre de méthodes, sans forcément étendre l'interface **PagingAndSortingRepository** :

```
// Renvoie un résultat paginé avec des méta-données sur la recherche (nombre de pages, etc.)  
Page<User> findByLastname(String lastname, Pageable pageable);  
  
// Renvoie une liste d'utilisateurs triée selon le paramètre "sort"  
List<User> findByLastname(String lastname, Sort sort);  
  
// Renvoie une liste d'utilisateurs qui prend en compte les paramètres de pagination données en paramètre  
List<User> findByLastname(String lastname, Pageable pageable);
```

Bien que pratiques, les méthodes-requêtes peuvent devenir très longues et donc peu pratiques pour les requêtes un peu complexes. Spring Data possède une annotation **@Query** dans laquelle on peut mettre en paramètre la requête sous forme de String. Dans ce cas, le nom de la méthode n'est pas prise en compte.

```
// JPA  
@Query("from Personne p where p.nom = ?1 and p.prenom = ?2")  
public Personne maRequeteAvecQueryDeRechercheParNomEtPrenom(String nom, String prenom);  
  
// Neo4j  
@Query("start acteur=node({0}) " +  
"match acteur-[a:Realise]->film " +  
"return film")  
Iterable<Film> recupereMoiTousLesFilmsRealisesPar(Acteur acteur);
```

Si la requête fait des modifications sur les entités, il faut rajouter l'annotation **@Modifying**.



```
@Query("update Personne p set p.nom = :nom where p.id = :id")  
@Modifying  
public int metAJourNom(@Param("nom")String nom, @Param("id") Long id);
```

Tous les cas d'utilisation que nous avons évoqués fonctionnent quelque soit la base de données prise en compte par Spring Data. C'est-à-dire que vous utilisez vos modèles de données sous forme de beans, et selon votre repository et les annotations utilisées, vous taperez sur l'une ou l'autre des bases de données. Par exemple, dans un bean, vous pouvez mettre des annotations propres à Spring Data Neo4j et d'autres propres à JPA.

## 2. *Spring Data JPA*

Spring Data JPA offre une interface supplémentaire qui étend PagingAndSortingRepository : JpaRepository. Elle propose des méthodes comme la suppression en lots des entités (deleteInBatch()), le vidage du cache (flush()), etc.

```
<groupId>org.springframework.data</  
<artifactId>spring-data-ipa</artifa
```

### **a. Requêtes personnalisées**

Mais que se passe-t-il si vous avez une requête à écrire qui doit utiliser les particularités de l'ORM sous-jacent ? Un exemple que j'ai pu rencontrer : l'écriture d'une requête de recherche par l'exemple. Cette fonctionnalité n'existe pas en JPA mais en Hibernate. La documentation de Spring Data décrit comment faire pour ajouter des requêtes pouvant utiliser l'API Criteria (et nous avons pu le faire). Ceci écrit, si cela peut se faire assez facilement pour un repository (il faut, en plus de l'interface Repository, avoir une interface exposant la méthode plus une classe qui implémente cette méthode avec par exemple le code Hibernate), le faire pour tous les repositories devient un peu plus compliqué puisqu'il faut créer une interface qui étend JpaRepository, créer une classe implémentant cette interface et enfin créer et déclarer un factory bean.

## b. QueryDsl

Spring Data JPA partage avec Spring Data MongoDB l'intégration avec QueryDsl. Pour cela, vous devrez inclure le plugin maven de QueryDsl pour qu'il génère les classes à partir de vos entités. Puis il vous suffira d'étendre l'interface QueryDslPredicateExecutor. Celle-ci possède quelques méthodes de recherche qui prennent en paramètres des Predicate :

```
BooleanExpression ignoreGraviteDansLaMatrice =  
    heros.ignoreGraviteDansMatrice.eq(true);  
BooleanExpression estBaleze = heros.nbEnnemisMisKO.gt(10);  
  
customerRepository.findAll(ignoreGraviteDansLaMatrice.and(estBaleze));
```

Pour information, lors de la rédaction de cet article, l'utilisation de QueryDsl n'est pas expliquée dans la documentation de référence, mais dans le livre sur Spring Data mis en valeur sur le site du projet (et aussi dans la documentation de Spring Data MongoDB).

### III) Présentation de Spring Boot

Spring Boot est un framework qui facilite le développement d'applications fondées sur Spring en offrant des outils permettant d'obtenir une application packagée en jar, totalement autonome. Ce qui nous intéresse particulièrement, puisque nous essayons de développer des Microservices !

Quelle peut être l'avantage de Spring Boot par rapport à Spring MVC ?

Spring et Spring MVC sont de formidables outils quand on essaye de développer une application web. Néanmoins, un de leurs plus gros problèmes est la configuration. Si vous avez déjà développé une application avec ces outils, vous avez dû remarquer que votre application est bardée de fichiers XML qui indiquent les configurations des servlets, des vues, des contenus statiques, etc. Ces fichiers de configuration deviennent un vrai challenge lorsque vous avez une application complexe.

Comment Spring Boot va nous permettre de simplifier tout cela ?

Pour simplifier la configuration d'une application, Spring Boot propose 2 fonctionnalités principales que nous allons voir dans la suite :

- l'auto-configuration,
- les starters.

#### L'auto-configuration

Cette fonctionnalité est la plus importante de Spring Boot. Elle permet de **configurer automatiquement** votre application à partir des *jars* trouvés dans votre Classpath. En d'autres termes, si vous avez importé des dépendances, Spring Boot ira consulter cette liste puis produira la configuration nécessaire pour que tout fonctionne correctement.

Prenons l'exemple d'une application web dans laquelle vous avez les dépendances : Hibernate et Spring MVC. Normalement, vous devez créer les fichiers de configuration suivants :

- dispatcher-servlet.xml
- web.xml
- persistance.xml

Comme vous ne connaissez pas nécessairement la syntaxe de ces fichiers par cœur, il vous faut consulter la documentation ou vous inspirer d'un ancien projet. Vous devez ensuite écrire le code Java qui permet de lier ces fichiers XML à *ApplicationContext* de Spring.

Ensuite, vous adaptez et testez, puis ré-adaptez et re-testez encore pendant un bon moment avant que tout fonctionne... Bien sûr, dès que vous faites un changement dans votre application, il ne faut pas oublier de mettre à jour tous les fichiers de configuration, puis déboguer de nouveau. Cela devient très vite très fastidieux !

Voici l'équivalent de l'ensemble de ces étapes avec Spring MVC :

```
@EnableAutoConfiguration
```

Avec cette annotation, **Spring Boot ira scanner la liste de vos dépendances**, trouvant par exemple Hibernate. Ayant constaté que vous n'avez défini aucun autre *datasource*, **il créera la configuration nécessaire** et l'ajoutera à *ApplicationContext*.

## Les Starters

Les starters viennent compléter l'auto-configuration et font gagner énormément de temps, notamment lorsqu'on commence le développement d'un Microservice. Un starter va apporter à votre projet un **ensemble de dépendances**, communément utilisées pour un type de projet donné. Ceci va vous permettre de créer un **"squelette" prêt à l'emploi** très rapidement.

L'autre énorme avantage est la **gestion des versions**. Plus besoin de chercher quelles versions sont compatibles puis de les ajouter une à une dans le *pom.xml* ! Il vous suffit d'ajouter une simple dépendance au starter de votre choix. Cette dépendance va alors ajouter, à son tour, les éléments dont elle dépend, avec les bonnes versions.

Prenons l'exemple où vous souhaitez créer un Microservice. En temps normal, vous aurez besoin des dépendances suivantes :

- Spring ;
- Spring MVC ;
- Jackson (pour json) ;
- Tomcat ;
- ...

Avec Spring Boot, vous allez tout simplement avoir une seule dépendance dans votre *pom.xml* :

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

Remarque : Tous les starters de Spring Boot sont au format **spring-boot-starter-NOM\_DU\_STARTER**.

Ce starter va charger les dépendances présentes dans le pom suivant :

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      <modelVersion>4.0.0</modelVersion>
4      <parent>
5          <groupId>org.springframework.boot</groupId>
6          <artifactId>spring-boot-starters</artifactId>
7          <version>1.5.9.RELEASE</version>
8      </parent>
9      <artifactId>spring-boot-starter-web</artifactId>
10     <name>Spring Boot Web Starter</name>
11     <description>Starter for building web, including RESTful, applications using Spring
12         MVC. Uses Tomcat as the default embedded container</description>
13     <url>http://projects.spring.io/spring-boot/</url>
14     <organization>
15         <name>Pivotal Software, Inc.</name>
16         <url>http://www.spring.io</url>
17     </organization>
18     <properties>
19         <main.basedir>${basedir}/../../</main.basedir>
20     </properties>
21     <dependencies>
22         <dependency>
23             <groupId>org.springframework.boot</groupId>
24             <artifactId>spring-boot-starter</artifactId>
25         </dependency>
26         <dependency>
27             <groupId>org.springframework.boot</groupId>
28             <artifactId>spring-boot-starter-logging</artifactId>
29         </dependency>
30         <dependency>
31             <groupId>org.springframework.boot</groupId>
32             <artifactId>spring-boot-starter-test</artifactId>
33         </dependency>
34         <dependency>
35             <groupId>com.fasterxml.jackson.core</groupId>
36             <artifactId>jackson-databind</artifactId>
37         </dependency>
38         <dependency>
39             <groupId>org.springframework</groupId>
40             <artifactId>spring-web</artifactId>
41         </dependency>
42         <dependency>
43             <groupId>org.springframework</groupId>
44             <artifactId>spring-webmvc</artifactId>
45         </dependency>
```

**Figure 1** : Exemple de pom.xml.

En version copiable :

```
<?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>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starters</artifactId>
    <version>1.5.9.RELEASE</version>
  </parent>
  <artifactId>spring-boot-starter-web</artifactId>
  <name>Spring Boot Web Starter</name>
  <description>Starter for building web, including RESTful, applications using Spring
    MVC. Uses Tomcat as the default embedded container</description>
  <url>http://projects.spring.io/spring-boot/</url>
  <organization>
    <name>Pivotal Software, Inc.</name>
    <url>http://www.spring.io</url>
  </organization>
  <properties>
    <main.basedir>${basedir}/../..</main.basedir>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
    </dependency>
  </dependencies>
```



```
</project>
```

Ce starter Spring Boot pour application web met en place la configuration des éléments suivants : tomcat, hibernate-validator, jackson, spring MVC.

Tout cela est bien beau mais comme on gère les différentes versions ?

Nous vous rappelons les lignes 4-8 de **Figure 1 : Exemple de pom.xml** :

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starters</artifactId>
  <version>1.5.9.RELEASE</version>
</parent>
```

Ce tag, ajouté en haut du pom.xml, permet à votre pom d'hériter des propriétés d'un autre pom que vous trouverez sur <https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-starters/spring-boot-starter-parent/pom.xml> (qui lui-même hérite d'un autre pom : spring-boot-dependencies). Il permet de définir principalement :

- La version de Java à utiliser. Dans ce cas, c'est la 1.6, nous verrons dans le prochain chapitre comment la surcharger pour utiliser Java 1.8.
- Une liste complète des versions des dépendances prises en charge (<https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-dependencies/pom.xml>), ce qui permet d'ajouter des dépendances sans indiquer leur version comme dans le pom.xml du starter spring-boot-starter-web vu plus haut. Vous allez donc pouvoir ajouter les dépendances de votre choix, sans vous soucier des versions.

Bien entendu, *spring-boot-starter-web* n'est pas le seul starter disponible. Selon ce que vous comptez développer, vous trouverez pratiquement toujours un **starter adéquat**. Voici quelques exemples :

- spring-boot-starter-mail : pour les applications et services d'envoi de mails.
- spring-boot-starter-thymeleaf : si vous souhaitez créer une application qui offre une interface utilisateur en utilisant le moteur de template thymeleaf.
- spring-boot-starter-web-services : pour les services plus classiques utilisant SOAP.

Vous trouverez ici une liste de tous les starters existants sur <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter>.

## En résumé

- Spring Boot est un framework qui permet de démarrer rapidement le développement d'applications ou services en fournissant les dépendances nécessaires et en auto-configurant celles-ci.
- Pour activer l'auto-configuration, on utilise l'annotation *@EnableAutoConfiguration*. Si vous écrivez vos propres configurations, celles-ci priment sur celles de Spring Boot.
- Les starters permettent d'importer un ensemble de dépendances selon la nature de l'application à développer afin de démarrer rapidement.

## IV) Création et initialisation d'un projet Spring Boot avec spring.io

Les notions de Spring Boot et Data sont complexes dans la mise en œuvre c'est pour cette raison que Spring met à disposition un site pour pouvoir initialiser un projet Spring Data Boot facilement, pour cela se rendre sur <https://start.spring.io/>.

Dans cette espace vous pourrez selection le projet à selectionner avec deux choix possible : **Maven Project** ou **Gradle Project**.

Remplir les champs de l'image ci-dessous avec :

Project : **Maven Project**

Language : **Java**

Spring Boot : **2.3.4**

Group : **com.cours.spring.boot**

Artefact : **maven-first-app-spring-boot**

Options → Name : **maven-first-app-spring-boot**

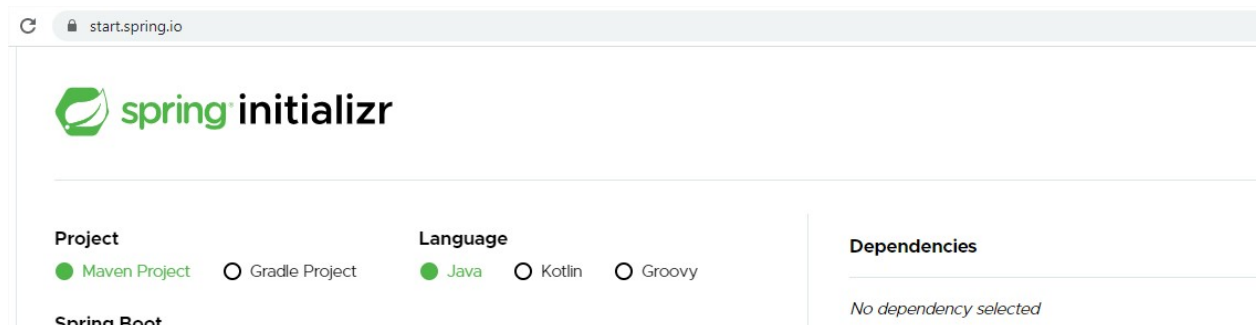
Options → Description : **My first Spring Boot project**

Options → Package Name : **com.cours**

Options → Packaging : **Jar**

Options → Java: **8**

Dependencies : **Spring Data JPA, Spring Web et MySQL Driver.**



The screenshot shows the Spring Initializr web interface. At the top, the URL 'start.spring.io' is visible in the browser's address bar. Below the 'spring initializr' logo, there are three main sections: 'Project', 'Language', and 'Dependencies'. In the 'Project' section, 'Maven Project' is selected with a radio button. In the 'Language' section, 'Java' is selected with a radio button. The 'Dependencies' section shows 'No dependency selected'. The 'Spring Boot' version is set to '2.3.4'.

Project	Language	Dependencies
<input checked="" type="radio"/> Maven Project <input type="radio"/> Gradle Project	<input checked="" type="radio"/> Java <input type="radio"/> Kotlin <input type="radio"/> Groovy	No dependency selected

Remplir les éléments d'identité de votre projet

Project

☒ Maven Project ☐ Gradle Project

Language

☒ Java ☐ Kotlin ☐

Spring Boot

☒ 2.3.4 ☐ 2.4.0 (SNAPSHOT) ☐ 2.4.0 (M3) ☐ 2.3.5 (SNAPSHOT) ☐ 2.2.11 (SNAPSHOT) ☐ 2.2.10 ☐ 2.1.18 (SNAPSHOT) ☐ 2.1.17

Project Metadata

Group

com.cours.spring.boot

Artifact

maven-first-app-spring-boot

Name

maven-first-app-spring-boot

Description

My first Spring Boot project

Package name

com.cours

Remplir les dépendances de votre projet.

Dependencies

ADD DEPENDENCIES

No dependency selected

Rechercher et ajouter les dependances : **Spring Web** puis **Spring Data JPA** et **MySQL Driver**.

Spring Web

Press

**Spring Web** WEB  
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default container.

**Spring Reactive Web** WEB  
Build reactive web applications with Spring WebFlux and Netty.

**Spring Web Services** WEB  
Facilitates contract-first SOAP development. Allows for the creation of flexible web services and many ways to manipulate XML payloads.

Spring Data

Press

**Spring Data JPA** SQL  
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

**Spring Data Elasticsearch (Access+Driver)** NOSQL  
A distributed, RESTful search and analytics engine with Spring Data Elasticsearch.

**Spring Data R2DBC** SQL

MySQL Driver

Press

**MySQL Driver** SQL  
MySQL JDBC and R2DBC driver.

On obtient donc dans l'onglet dependencies :

Dependencies

ADD DEPENDENCIES

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA

SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MySQL Driver

SQL

MySQL JDBC and R2DBC driver.

On obtient donc au final :

Project

☒ Maven Project ☐ Gradle Project

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☒ 2.3.4 ☐ 2.4.0 (SNAPSHOT) ☐ 2.4.0 (M3) ☐ 2.3.5 (SNAPSHOT) ☐ 2.2.11 (SNAPSHOT) ☐ 2.2.10 ☐ 2.1.18 (SNAPSHOT) ☐ 2.1.17

Project Metadata

Group

com.cours.spring.boot

Artifact

maven-first-app-spring-boot

Name

maven-first-app-spring-boot

Description

My first Spring Boot project

Package name

com.cours

Packaging

☒ Jar ☐ War

Dependencies

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Data JPA

SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

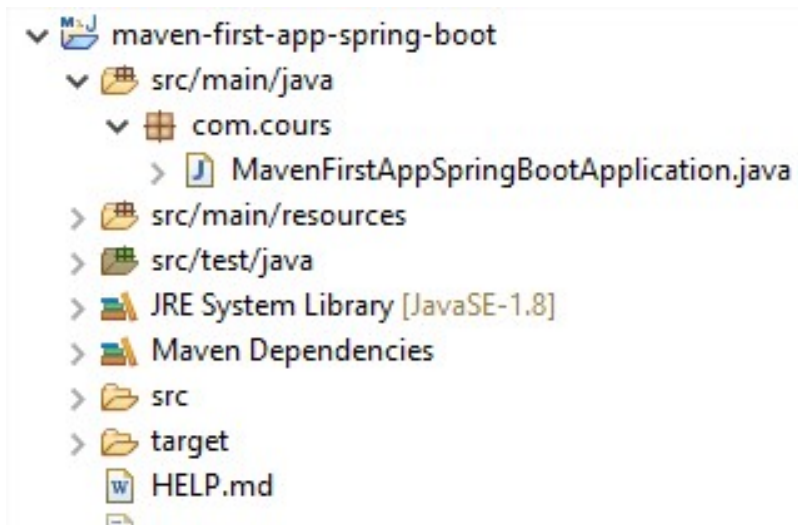
MySQL Driver

SQL

MySQL JDBC and R2DBC driver.

Cliquer sur **Generate the project** pour generer les projet final.

Ouvrir le projet squelette **maven-first-app-spring-boot** dans votre IDE.



## 1. Modification éventuelle du fichier pom.xml

**Attention si vous utilisez MYSQL 8 cette section ne vous concerne pas** mais en cas d'utilisation de MYSQL 5 il est nécessaire de modifier les dépendances du fichier **pom.xml** qui peuvent devenir :

```
1  <dependencies>
2      <dependency>
3          <groupId>org.springframework.boot</groupId>
4          <artifactId>spring-boot-starter-data-jpa</artifactId>
5      </dependency>
6      <dependency>
7          <groupId>org.springframework.boot</groupId>
8          <artifactId>spring-boot-starter-web</artifactId>
9      </dependency>
10     <!-- For MYSQL current (latest) : <scope>runtime</scope> -->
11     <!-- For MYSQL version 5.1.47 : <version>5.1.47</version> -->
12     <dependency>
13         <groupId>mysql</groupId>
14         <artifactId>mysql-connector-java</artifactId>
15         <version>5.1.47</version>
16     </dependency>
17     <dependency>
18         <groupId>org.springframework.boot</groupId>
19         <artifactId>spring-boot-starter-test</artifactId>
20         <scope>test</scope>
21         <exclusions>
22             <exclusion>
23                 <groupId>org.junit.vintage</groupId>
```

En version copiable :

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <!-- For MYSQL current (latest) : <scope>runtime</scope> -->
  <!-- For MYSQL version 5.1.47 : <version>5.1.47</version> -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
      <exclusion>
```



```
<groupId>org.junit.vintage</groupId>  
<artifactId>junit-vintage-engine</artifactId>  
</exclusion>  
</exclusions>  
</dependency>  
</dependencies>
```

## 2. Modification de nom de classe

Renommer la classe `com.cours.MavenFirstAppSpringBootApplication` en `com.cours.MainApp` puis lui ajouter l'annotation `@EnableTransactionManagement`.

Elle devient au final :

```
package com.cours;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@SpringBootApplication
@EnableTransactionManagement
public class MainApp {
    public static void main(String[] args) {
        SpringApplication.run(MainApp.class, args);
    }
}
```

### 3. Modification du fichier *application.properties*

Le fichier **maven-first-app-spring-boot/src/main/resources/application.properties** aura donc pour contenu finale :

```
1  # For MYSQL 8
2  spring.datasource.url=jdbc:mysql://localhost:3308/my_data_base?serverTimezone=UTC
3  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
4  # For MYSQL 5
5  #spring.datasource.url=jdbc:mysql://localhost:3306/my_data_base?useSSL=false
6  #spring.datasource.driver-class-name=com.mysql.jdbc.Driver
7  spring.datasource.username=application
8  spring.datasource.password=passw0rd
9  # For Hibernate version < 5.3.1.Final
10 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
11 # For Hibernate version > 5.3.1.Final
12 #spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
13 spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStra
```

En version copiable :

```
# For MYSQL 8
spring.datasource.url=jdbc:mysql://localhost:3308/my_data_base?serverTimezone=UTC
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
# For MYSQL 5
#spring.datasource.url=jdbc:mysql://localhost:3306/my_data_base?useSSL=false
#spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.username=application
spring.datasource.password=passw0rd
# For Hibernate version < 5.3.1.Final
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
# For Hibernate version > 5.3.1.Final
#spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.naming.physical-
strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
spring.jpa.hibernate.ddl-auto=update
# Manage Error Message
server.error.include-message = always
```

Si vous voulez avoir plus de détails sur les elements à mettre dans le fichier **application.properties** aller sur <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html#data-properties> :

Back to index

1. Core properties

2. Cache properties

3. Mail properties

4. JSON properties

5. Data properties

6. Transaction properties

7. Data migration properties

8. Integration properties

9. Web properties

10. Templating properties

11. Server properties

12. Security properties

13. RSocket properties

14. Actuator properties

15. Devtools properties

16. Testing properties

### 5. Data properties

Key	Default Value	Description
<code>spring.couchbase.connection-string</code>		Connection string used to lo
<code>spring.couchbase.env.io.idle-http-connection-timeout</code>	<code>4500ms</code>	Length of time an HTTP con closed and removed from th
<code>spring.couchbase.env.io.max-endpoints</code>	<code>12.0</code>	Maximum number of socket
<code>spring.couchbase.env.io.min-endpoints</code>	<code>1.0</code>	Minimum number of sockets
<code>spring.couchbase.env.ssl.enabled</code>		Whether to enable SSL sup "keyStore" is provided unles
<code>spring.couchbase.env.ssl.key-store</code>		Path to the JVM key store th
<code>spring.couchbase.env.ssl.key-store-password</code>		Password used to access th

Vous pouvez aussi ajouter de configurations serveur avec le lien : <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html#server-properties> :

Back to index

1. Core properties

2. Cache properties

3. Mail properties

4. JSON properties

5. Data properties

6. Transaction properties

7. Data migration properties

8. Integration properties

9. Web properties

10. Templating properties

11. Server properties

12. Security properties

13. RSocket properties

14. Actuator properties

15. Devtools properties

16. Testing properties

### 11. Server properties

Key	Default Value	Description
<code>server.address</code>		Network address to w
<code>server.compression.enabled</code>	<code>false</code>	Whether response cor
<code>server.compression.excluded-user-agents</code>		Comma-separated list should not be compre
<code>server.compression.mime-types</code>	<code>[text/html, text/xml, text/plain, text/css, text/javascript, application/javascript, application/json, application/xml]</code>	Comma-separated list compressed.
<code>server.compression.min-response-size</code>	<code>2KB</code>	Minimum "Content-Le compression to be pe
<code>server.error.include-binding-errors</code>	<code>never</code>	When to include "erro

#### 4. La classe *MyEntity*

Créer la classe **com.cours.entities.MyEntity** dont le contenu sera :

```
package com.cours.entities;

import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Version;
import javax.xml.bind.annotation.XmlRootElement;

@Entity
@Table(name = "MyEntity")
@XmlRootElement
public class MyEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "id")
    private Integer id;

    @Column(name = "field1")
    private String field1;

    @Column(name = "field2")
    private String field2;

    @Column(name = "version")
    @Version
    private Integer version;

    public MyEntity() {
    }

    public MyEntity(Integer id, String field1, String field2) {
        this.id = id;
        this.field1 = field1;
        this.field2 = field2;
    }

    public MyEntity(String field1, String field2) {
        this.field1 = field1;
    }
}
```

```
        this.field2 = field2;
    }

    public Integer getId() {
        return id;
    }

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

    public String getField1() {
        return field1;
    }

    public void setField1(String field1) {
        this.field1 = field1;
    }

    public String getField2() {
        return field2;
    }

    public void setField2(String field2) {
        this.field2 = field2;
    }

    public Integer getVersion() {
        return version;
    }

    public void setVersion(Integer version) {
        this.version = version;
    }
}
```

## *5. Suppression de la classe de test unitaire*

Supprimer la classe **com.cours.MavenFirstAppSpringBootApplicationTests** de votre projet car il pourrait empêcher le lancement de l'application s'il est mal écrit. Nous implémenterons ces tests unitaires correctement dans les exercices d'application.

## 6. La base de données

Créer la base de données **my\_data\_base** avec la table **MyEntity**.

```
/* Base de données: my_data_base */
DROP DATABASE IF EXISTS my_data_base;
/*DROP USER IF EXISTS 'application'@'localhost';*/
CREATE DATABASE my_data_base DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
CREATE USER IF NOT EXISTS 'application'@'localhost' IDENTIFIED BY 'passw0rd';
/* For MYSQL 8 */
GRANT ALL PRIVILEGES ON my_data_base.* TO 'application'@'localhost';
/* For MYSQL 5 */
/*GRANT ALL ON my_data_base.* TO 'application'@'localhost' IDENTIFIED BY 'passw0rd';*/
USE my_data_base;

SET FOREIGN_KEY_CHECKS = 0;
DROP TABLE IF EXISTS MyEntity;

CREATE TABLE MyEntity (
  id INTEGER PRIMARY KEY AUTO_INCREMENT,
  field1 VARCHAR(100),
  field2 VARCHAR(100),
  version int(15)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO MyEntity(field1,field2,version) VALUES ('field1-1', 'field2-1',0);
INSERT INTO MyEntity(field1,field2,version) VALUES ('field1-2', 'field2-2',0);
INSERT INTO MyEntity(field1,field2,version) VALUES ('field1-3', 'field2-3',0);
INSERT INTO MyEntity(field1,field2,version) VALUES ('field1-4', 'field2-4',0);
INSERT INTO MyEntity(field1,field2,version) VALUES ('field1-5', 'field2-5',0);
INSERT INTO MyEntity(field1,field2,version) VALUES ('field1-6', 'field2-6',0);
INSERT INTO MyEntity(field1,field2,version) VALUES ('field1-7', 'field2-7',0);
INSERT INTO MyEntity(field1,field2,version) VALUES ('field1-8', 'field2-8',0);
INSERT INTO MyEntity(field1,field2,version) VALUES ('field1-9', 'field2-9',0);
INSERT INTO MyEntity(field1,field2,version) VALUES ('field1-10', 'field2-10',0);
```



## 7. L'interface *MyEntityRepository*

Créer l'interface **com.cours.repository.MyEntityRepository** dont le contenu sera :

```
package com.cours.repository;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

import com.cours.entities.MyEntity;

@Repository
public interface MyEntityRepository extends CrudRepository<MyEntity, Integer> {

    Iterable<MyEntity> findByField1(String field1);

    Iterable<MyEntity> findByField2(String field2);
}
```

## 8. La classe *MyEntityController*

Créer la classe **com.cours.controller.MyEntityController** dont le contenu sera :

```
package com.cours.controller;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.server.ResponseStatusException;

import com.cours.entities.MyEntity;
import com.cours.repository.MyEntityRepository;

@RestController
@RequestMapping("/api")
public class MyEntityController {

    @Autowired
    private MyEntityRepository myEntityRepository;

    @GetMapping
    public Iterable<MyEntity> findAll() {
        return myEntityRepository.findAll();
    }

    /**
     *
     * Avec Java 8 vous pouvez utilisé les expressions lamda :
     */
    @GetMapping("findById/{id}")
    public MyEntity findById(@PathVariable Integer id) {
        return myEntityRepository.findById(id).orElseThrow(() -
> new ResponseStatusException(HttpStatus.NOT_FOUND,
        "Le MyEntity id = " + id + " est introuvable."));
    }

    /**
     *
     * Avec Java 7 nous auront simplement :
     */
    @GetMapping("findByIdBis/{id}")
```

```

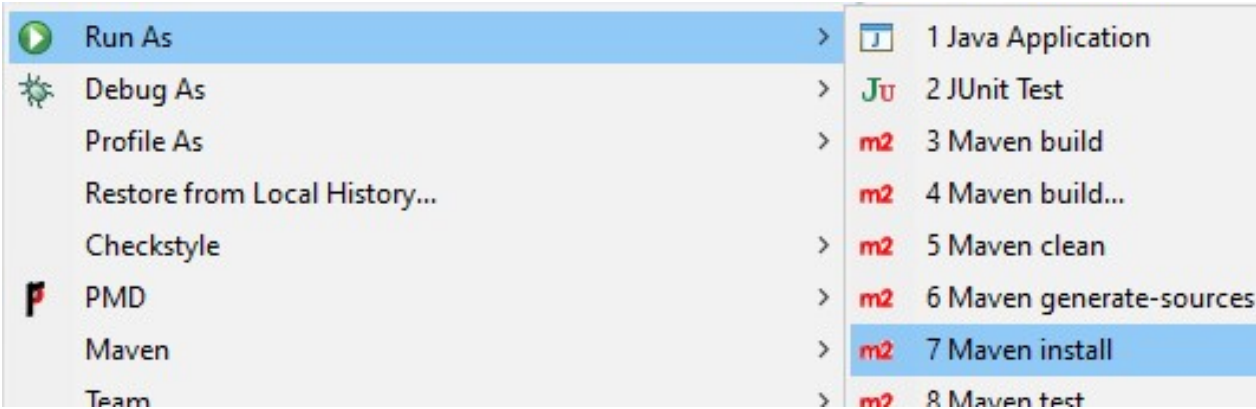
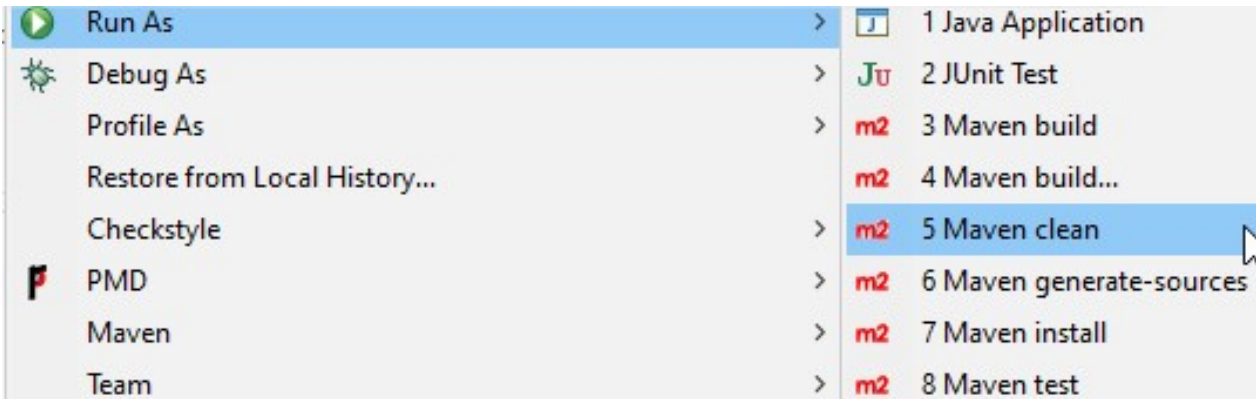
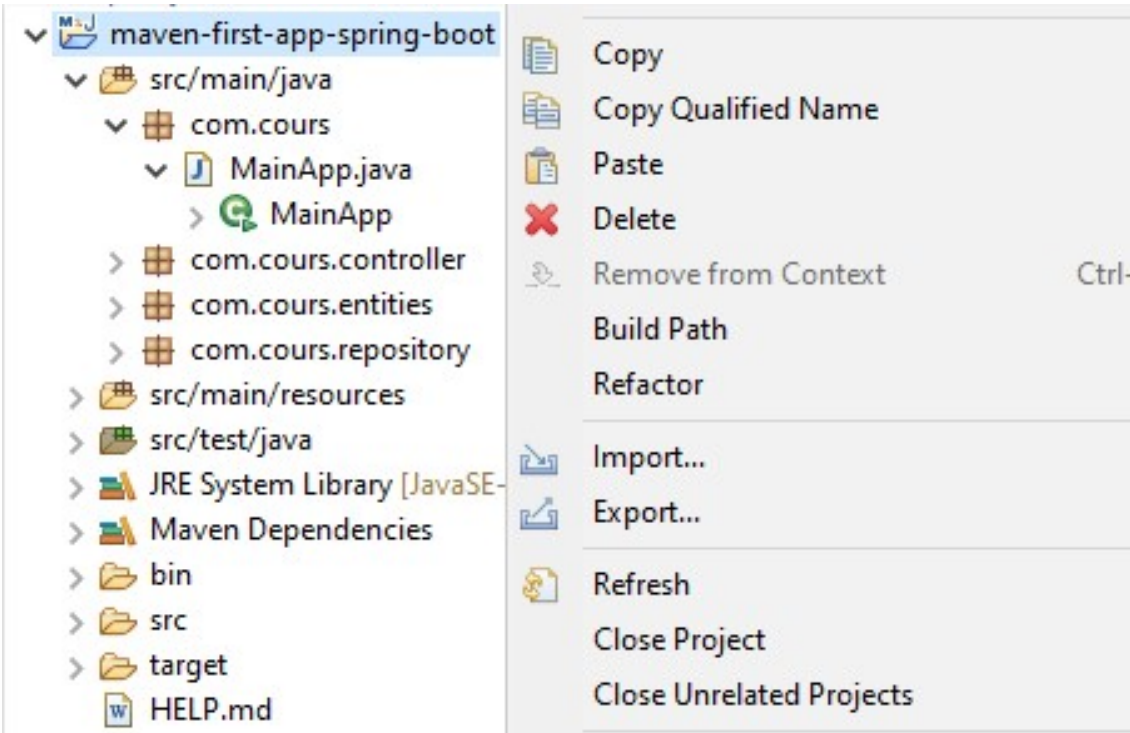
public ResponseEntity<MyEntity> findByIdBis(@PathVariable Integer id) {
    ResponseEntity<MyEntity> response = null;
    Optional<MyEntity> optMyEntity = myEntityRepository.findById(id);
    if (optMyEntity.isPresent()) {
        response = new ResponseEntity<MyEntity>(optMyEntity.get(), HttpStatus.OK);
    } else {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Le MyEntity id = " + id + " est introuvable.");
    }
    return response;
}

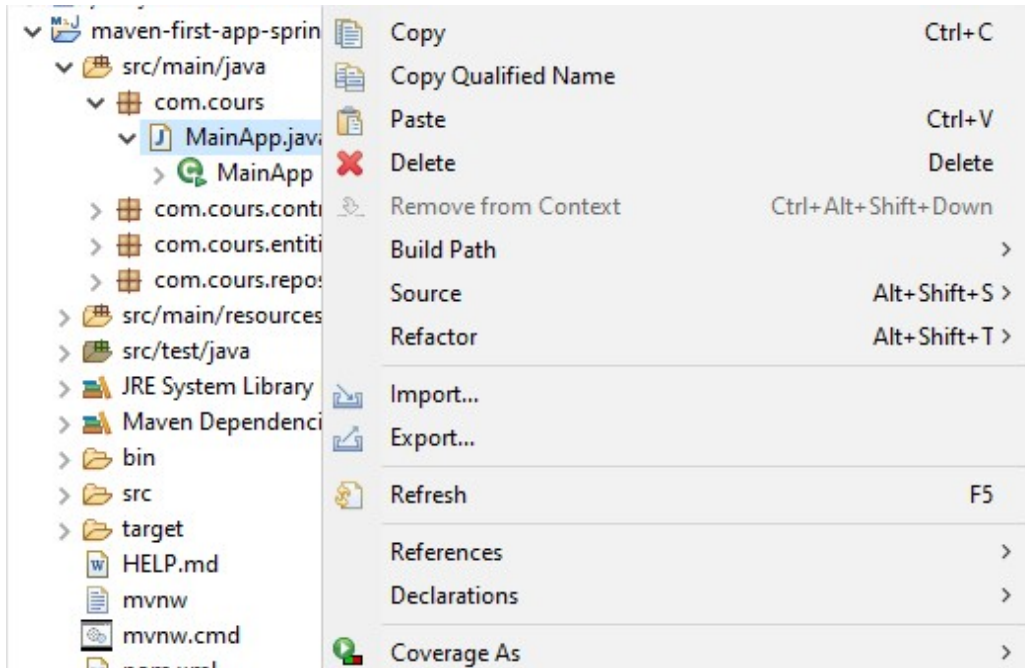
@GetMapping("findByField1/{field1}")
public Iterable<MyEntity> findByField1(@PathVariable String field1) {
    Iterable<MyEntity> itrMyEntities = myEntityRepository.findByField1(field1);
    if (itrMyEntities == null) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND,
            "Aucun MyEntity avec field1 = " + field1 + " n'a été trouvé.");
    }
    return itrMyEntities;
}

@GetMapping("findByField2/{field2}")
public Iterable<MyEntity> findByField2(@PathVariable String field2) {
    Iterable<MyEntity> itrMyEntities = myEntityRepository.findByField1(field2);
    if (itrMyEntities == null) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND,
            "Aucun MyEntity avec field2 = " + field2 + " n'a été trouvé.");
    }
    return itrMyEntities;
}
}

```

9. Test de l'application





MainApp (2) [Java Application] C:\Program Files\Java\jdk1.8.0\_131\bin\javaw.exe (27 sept. 2020 à 15:47:02)

```

Spring Boot
:: Spring Boot :: (v2.3.4.RELEASE)

2020-09-27 15:47:03.214 INFO 8684 --- [main] com.cours.MainApp : Starting MainApp on DESKTOP-04L8JGP with PID 8684 (C:\Users\elhad\Deskte
2020-09-27 15:47:03.218 INFO 8684 --- [main] com.cours.MainApp : No active profile set, falling back to default profiles: default
2020-09-27 15:47:04.297 INFO 8684 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFERRED mode.
2020-09-27 15:47:04.382 INFO 8684 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 71ms. Found 1 JPA repository
2020-09-27 15:47:05.747 INFO 8684 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2020-09-27 15:47:05.763 INFO 8684 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-09-27 15:47:05.764 INFO 8684 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.38]
2020-09-27 15:47:05.922 INFO 8684 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2020-09-27 15:47:05.923 INFO 8684 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 2624 ms
2020-09-27 15:47:06.260 INFO 8684 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-09-27 15:47:06.332 INFO 8684 --- [task-1] o.hibernate.jpa.internal.util.LogHelper : HHH0000204: Processing PersistenceUnitInfo [name: default]
2020-09-27 15:47:06.448 INFO 8684 --- [task-1] org.hibernate.Version : HHH0000412: Hibernate ORM core version 5.4.21.Final
2020-09-27 15:47:06.516 WARN 8684 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queri
2020-09-27 15:47:06.814 INFO 8684 --- [task-1] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.1.0.Final}
2020-09-27 15:47:07.027 INFO 8684 --- [task-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2020-09-27 15:47:07.558 INFO 8684 --- [task-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2020-09-27 15:47:07.610 INFO 8684 --- [task-1] org.hibernate.dialect.Dialect : HHH0000400: Using dialect: org.hibernate.dialect.MySQL5InnoDBDialect

```

Lancer sur un navigateur l'URL <http://localhost:8080/api/>



## 9. Mise en place de la documentation Swagger

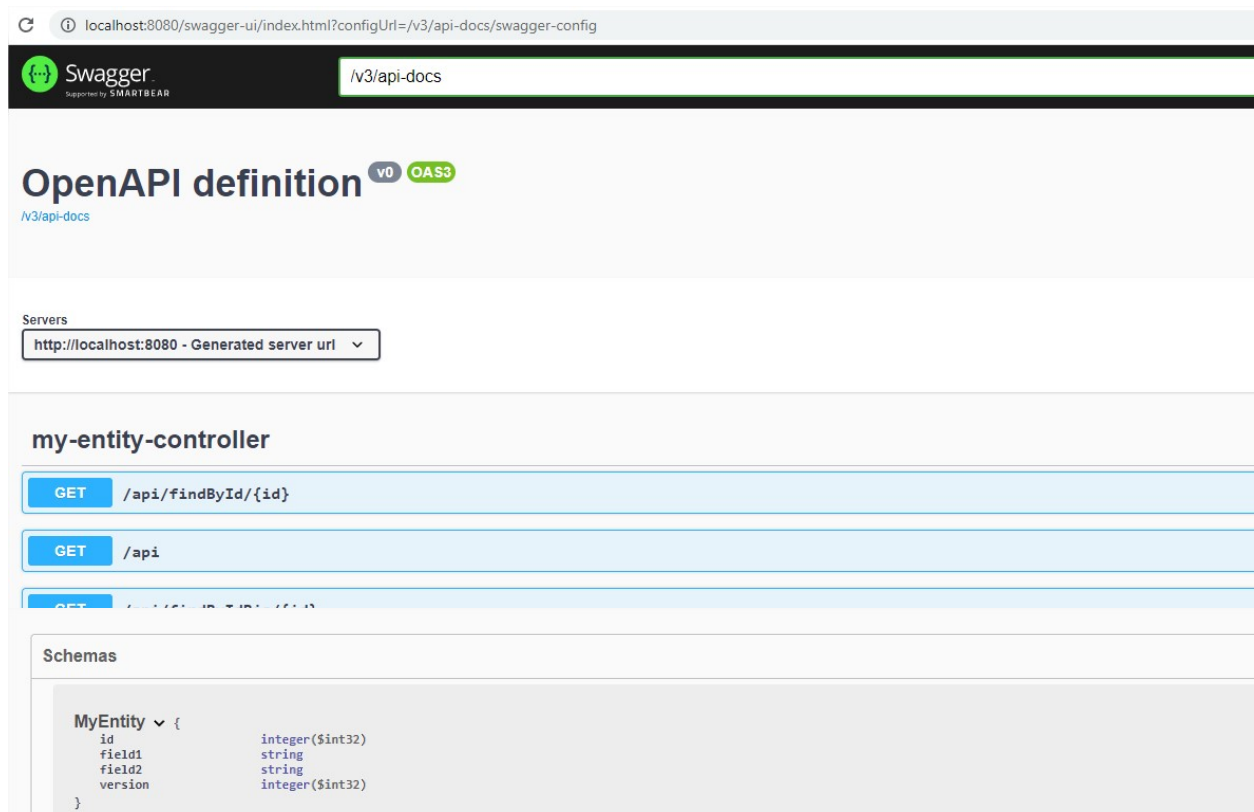
Modifier le fichier **pom.xml** en ajoutant la dépendance de Swagger :

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.4.6</version>
</dependency>
```

Modifier le fichier **application.properties** en ajoutant Swagger :

```
springdoc.swagger-ui.path=/swagger-ui.html
```

Lancer l'URL <http://localhost:8080/swagger-ui.html>



En cliquant sur **GET/api** on obtient :



localhost:8080/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config#/my-entity-controller/findAll

Servers

http://localhost:8080 - Generated server url

### my-entity-controller

GET /api/findById/{id}

GET /api

Parameters

localhost:8080/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config#/my-entity-controller/findAll

### my-entity-controller

GET /api/findById/{id}

GET /api

Parameters

No parameters

localhost:8080/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config#/my-entity-controller/findAll

Curl

```
curl -X GET "http://localhost:8080/api" -H "accept: */*"
```

Request URL

```
http://localhost:8080/api
```

Server response

Code	Details
200	<p>Response body</p> <pre>[   {     "id": 1,     "field1": "field1-1",     "field2": "field2-1",     "version": 0   },   {     "id": 2,     "field1": "field1-2",     "field2": "field2-2",     "version": 0   },   {     "id": 3,     "field1": "field1-3",     "field2": "field2-3",     "version": 0   },   {     "id": 4,     "field1": "field1-4",</pre>

**10. Construire une image Docker complète avec MYSQL 8 + JAVA + MAVEN + Jar du Micro Service (Voir le cours sur Docker)**



## V) Exercices d'application maven-gestion-personnes-spring-boot

1. Créer le projet Maven Application jar *maven-gestion-personnes-spring-boot*
2. Créer la base de donnée *base\_personnes* avec le script *script\_base\_personnes.sql*
3. Mettre à jour le fichier *application.properties*

C'est afin de configurer correctement votre application Spring Boot (Faites attention au numéro de port car sur **MYSQL 5** le port par défaut est **3306** alors que sur **MYSQL 8** c'est par défaut **3308** mais dans tous les cas ce sera à vérifier dans votre base de données).

4. Créer l'entité *com.cours.entities.Personne* image de la table *Personne*
5. Créer l'interface *com.cours.repository.PersonneRepository*
6. Créer la classe *com.cours.controller.PersonneController* avec son **CRUD** au complet

C'est-à-dire les méthodes *findAll()*, *findById(Integer id)*, *createOrUpdate(Personne person)*, *delete(Integer id)*, *findByPrenom(String prenom)*, *findByNom(String nom)*, *findByPrenomNom(String prenom, String nom)*, *findByCodePostal(String codePostal)*, *findByVille(String ville)* et *authenticate(String prenom, String nom)*.

7. Mettre en place la documentation *Swagger*
8. Construire une image *Docker* complète avec **MYSQL 8 + JAVA + MAVEN + Jar du Micro Service**

## VI) Exercices d'application maven-gestion-personnes-jersey-spring-boot

### 1. Créer le projet Maven Application jar *maven-gestion-personnes-jersey-spring-boot*

Se rendre sur <https://start.spring.io/>.

Dans cette espace vous pouvez selection le projet à selectionner avec deux choix possible : **Maven Project** ou **Gradle Project**.

Remplir les champs de l'image ci-dessous avec :

Project : **Maven Project**

Language : **Java**

Spring Boot : **2.3.4**

Group : **com.cours.spring.boot**

Artefact : **maven-gestion-personnes-jersey-spring-boot**

Options → Name : **maven-gestion-personnes-jersey-spring-boot**

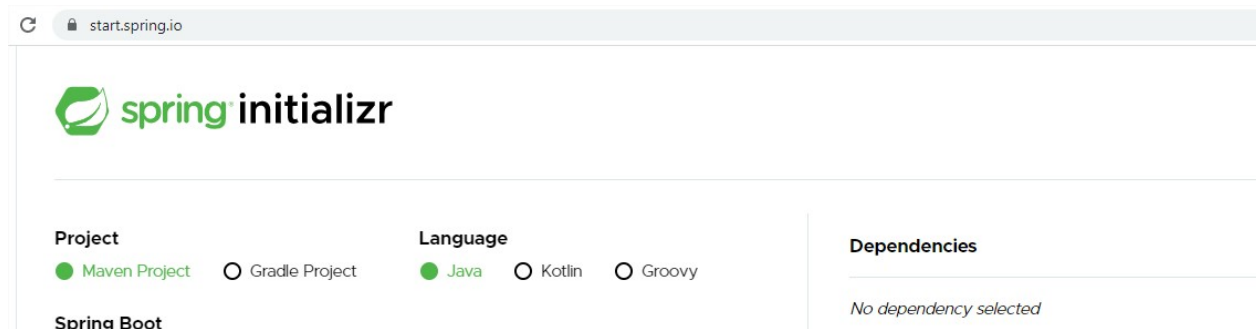
Options → Description : **My Spring Boot Jersey Project**

Options → Package Name : **com.cours**

Options → Packaging : **Jar**

Options → Java: **8**

Dependencies : **Jersey, Spring Data JPA, Spring Web et MySQL Driver.**



The screenshot shows the Spring Initializr web interface. At the top, the URL 'start.spring.io' is visible in the browser's address bar. Below the 'spring initializr' logo, there are three main sections: 'Project', 'Language', and 'Dependencies'. In the 'Project' section, 'Maven Project' is selected with a radio button. In the 'Language' section, 'Java' is selected with a radio button. In the 'Dependencies' section, 'No dependency selected' is displayed. The 'Spring Boot' version is also visible at the bottom left of the form.

start.spring.io

**Project**

☒ Maven Project
 ☐ Gradle Project

**Language**

☒ Java
 ☐ Kotlin
 ☐ Groovy

**Spring Boot**

☐ 2.4.0 (SNAPSHOT)
 ☐ 2.4.0 (M3)
 ☐ 2.3.5 (SNAPSHOT)
 ☒ 2.3.4
 ☐ 2.2.11 (SNAPSHOT)
 ☐ 2.2.10
 ☐ 2.1.18 (SNAPSHOT)
 ☐ 2.1.17

**Project Metadata**

Group

com.cours.jersey.spring.boot

Artifact

maven-gestion-personnes-jersey-spring-boot

Name

maven-gestion-personnes-jersey-spring-boot

Description

My Spring Boot Jersey Project

Package name

com.cours

**Dependencies**

**Jersey**

WEB

Framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs.

**Spring Web**

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

**Spring Data JPA**

SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

**MySQL Driver**

SQL

MySQL JDBC and R2DBC driver.

**Spring Boot Actuator**

OPS

**Dependencies**

**Jersey**

WEB

Framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs.

**Spring Web**

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

**Spring Data JPA**

SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

**MySQL Driver**

SQL

MySQL JDBC and R2DBC driver.

ADD DEPENDENCIES

Cliquer sur **Generate the project** pour generer les projet final.

2. Créer la base de donnée *base\_personnes* avec le script *script\_base\_personnes.sql*

3. Mettre à jour le fichier *application.properties*

C'est afin de configurer correctement votre application Spring Boot (Faites attention au numéro de port car sur **MYSQL 5** le port par défaut est **3306** alors que sur **MYSQL 8** c'est par défaut **3308** mais dans tous les cas ce sera à vérifier dans votre base de données).

4. Créer l'entité *com.cours.entities.Personne* image de la table *Personne*

5. Créer l'interface *com.cours.repository.PersonneRepository*

6. Créer la classe *com.cours.rest.config.JerseyConfig*

7. Créer la classe *com.cours.exception.MyRestException* pour gérer les exception Rest.

Cette classe héritera de la classe *javax.ws.rs.WebApplicationException*.

8. Créer la classe *com.cours.rest.GestionPersonnesJerseyResource* avec son CRUD au complet

C'est-à-dire les méthodes *findAll()*, *find(Integer id)*, *createOrUpdate(Personne person)*, *delete(Integer id)*, *findByPrenom(String prenom)*, *findByNom(String nom)*, *findByPrenomNom(String prenom, String nom)*, *findByCodePostal(String codePostal)*, *findByVille(String ville)* et *authenticate(String prenom, String nom)*.

9. Mettre en place la documentation Swagger

10. Construire une image Docker complète avec **MYSQL 8 + JAVA + MAVEN + Jar du Micro Service**