**Dependency Injection**

le processus de création de l’objet Product et son affectation dans l’objet Command soient **délégués à une tierce partie**. Voilà que, par magie (ou presque), l’objet Command n’a plus à s’occuper de l’objet Product, et ce dernier ne se préoccupe d’ailleurs pas non plus de l’objet Command.

Comment mettre en place cette tierce partie magique ? Dois-je la développer ?

Non, l’hiver est passé, le printemps est là ! Spring Framework s’occupe de tout grâce à son **IoC container** ! Ce dernier est aussi appelé le ***contexte Spring***. Il vous permettra de créer des objets dynamiquement, et de les injecter dans d’autres objets. De plus, on pourra facilement modifier l’implémentation d’un objet, avec quasiment zéro impact sur les objets qui utilisent ce dernier.

IoC est le sigle de Inversion of Control. Cette expression indique un principe de programmation qui correspond au fait de déléguer à un framework le flux de construction et d’appels des objets.

**configuration**

Parmi nos problématiques en tant que développeur, il y a celle d’être **performant** dans le développement de notre projet. Cette performance peut être liée au **temps** qu’on passe à développer, mais aussi à la **qualité** du code produit. Et pour cela, Spring Framework nous offre un bel avantage : **la configuration**.

Une des particularités du framework est d’encourager (voire de forcer) les développeurs à implémenter certaines parties du code via de la configuration et non du développement.

Prenons un exemple. Pour vous connecter à une base de données en Java, vous avez besoin de :

* charger un driver de base de données (le driver MySQL, par exemple) ;
* créer différents objets (comme *java.sql.Connection*) ;
* manipuler tous ces objets dans le bon ordre ;
* gérer les exceptions, etc.

Bref, rien que de l’écrire, ça m’a fatigué !  C’est pas vraiment complexe, mais c’est fastidieux !

Mais Spring Framework va nous permettre de résoudre tout ça en **quelques lignes** ! Non pas en quelques lignes de code, mais de **configuration** ! Pour cela, il nous suffit d’utiliser les **bons composants Spring** (j’y reviendrai dans le prochain chapitre) et le bon fichier de configuration :

spring.datasource.url=jdbc:mysql://localhost/test

spring.datasource.username=dbuser

spring.datasource.password=dbpass

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

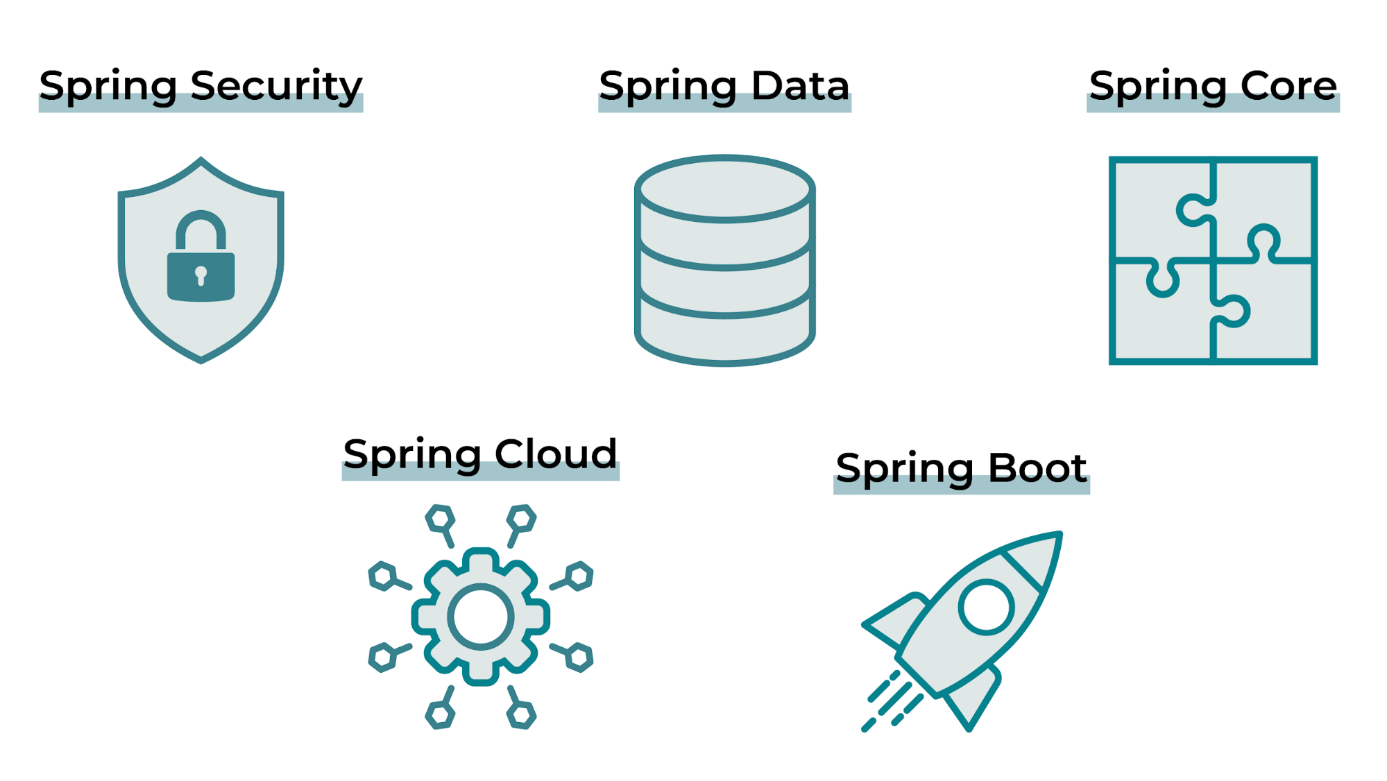
Voilà, c’est réglé : au sein de l’IoC container, tous les objets ont été créés, et on peut interagir avec notre base de données !

Qu’en dites-vous ? N’est-ce pas beaucoup plus performant ? ;-)

Et ce n’est qu’un exemple parmi tant d’autres !

Spring Framework est également un **outil très puissant**car il simplifie certaines actions, indispensables pour répondre à nos besoins actuels, que nous aurions beaucoup de mal à réaliser par nous-mêmes. Par exemple :

* Interagir avec une base de données.
* Traiter des requêtes HTTP et écrire des réponses HTTP.
* Exécuter des traitements par lots (batch).
* Gérer la sécurité de l’application.
* Etc.



### Spring Core

Ce composant est **la base de l’écosystème** Spring.

Il contient le “**core container**” (ce qui permet l’injection de dépendances vue précédemment), mais il contient également **Spring MVC** qui permet de faire du web et de Data Access qui fournit des éléments fondamentaux pour la communication avec les bases de données.

Pour avoir tout le détail, suivez [**la documentation officielle**](https://spring.io/projects/spring-framework) sur Spring Core.

### Spring Data

Ce composant permet de **communiquer avec de nombreux types de bases de données**. Par exemple, il offre la capacité de communiquer avec une base de données en implémentant uniquement des interfaces grâce à des conventions de nommage : bluffant !

Pour avoir tout le détail, suivez [**la documentation officielle**](https://spring.io/projects/spring-data) sur Spring Data.

### Spring Security

Pensez-vous que la sécurité soit un élément essentiel d’une application ? Moi, oui ! Et des millions de développeurs partagent ce point de vue. C’est pour ça que ce composant est l’un des plus critiques de Spring Framework, bien qu’il soit aussi l’un des plus complexes.  
Il permet de gérer **l’authentification**, **l’autorisation**, mais aussi la **sécurité des API**.

Pour avoir tout le détail, suivez [**la documentation officielle**](https://spring.io/projects/spring-security) sur Spring Security.

### Spring Cloud

Avez-vous entendu parler de l’**architecture microservice**? Si ce n’est pas le cas, ne vous inquiétez pas, mais cela va venir très vite car c’est le modèle d'architecture le plus prisé actuellement. Et pour répondre aux contraintes de cette architecture logicielle, Spring Framework fournit Spring Cloud.

Pour avoir tout le détail, suivez [**la documentation officielle**](https://spring.io/projects/spring-cloud) sur Spring Cloud.

### Spring Boot

C’est un composant très particulier de Spring Framework, dans la mesure où il nous permet de mettre en œuvre tous les autres. Ce cours vous montrera comment tirer profit de la puissance de Spring Boot, et de ses avantages qui sont :

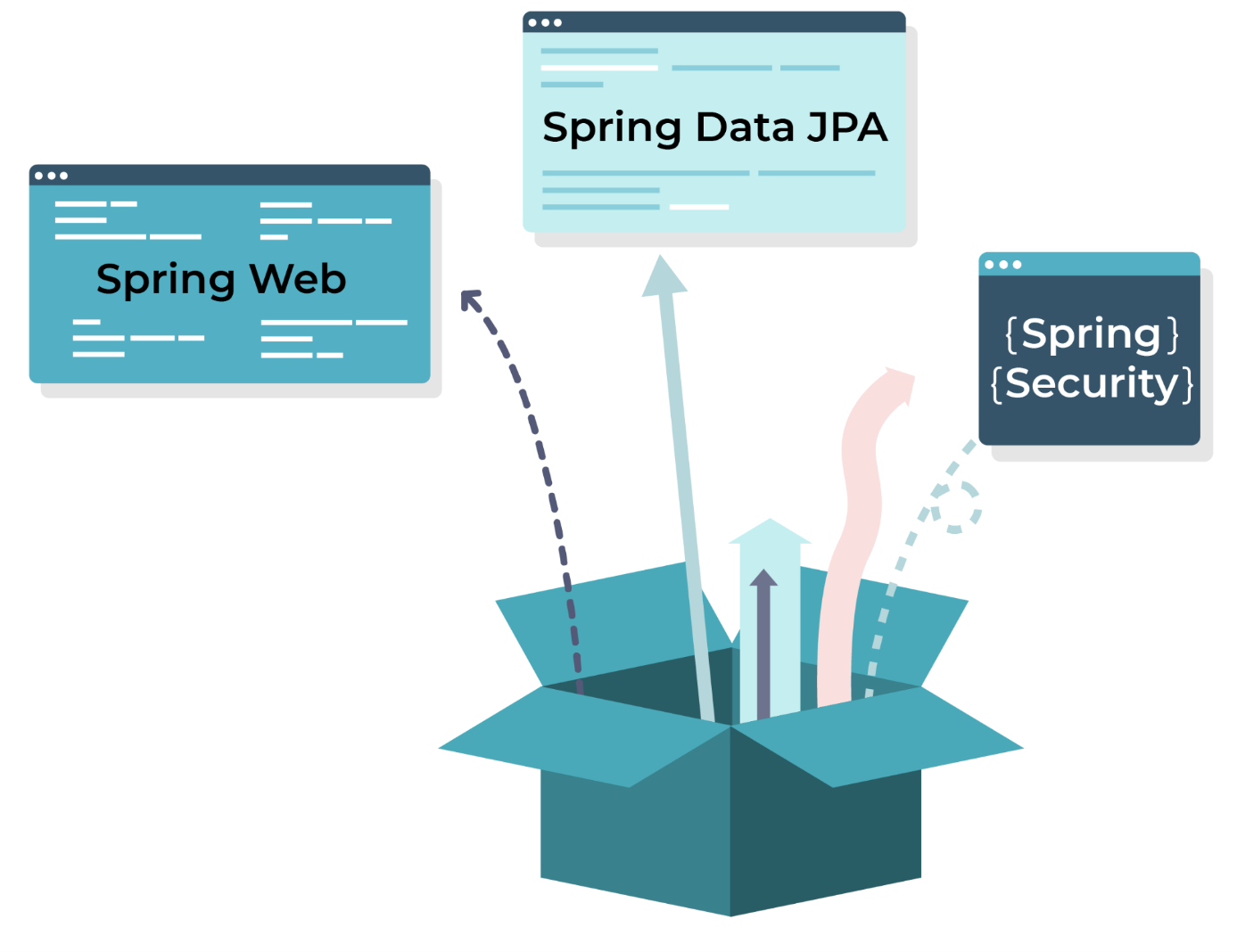
* l'**autoconfiguration** automatique de Spring ;
* des **starters de dépendances** ;
* des **endpoints Actuator** pour fournir des données sur l’application.

### Les avantages de Spring Boot

Reprenons notre souffle après cet exercice, et analysons les avantages de Spring Boot.

#### Avantage n° 1 : optimisation de la gestion des dépendances

Spring Boot nous fournit des **starters**, qui correspondent à un ensemble de dépendances homogénéisées (associations, versions). On peut les comparer à des **kits de dépendances**.

Les starters

Nul besoin de définir les versions des dépendances explicitement dans le pom.xml : Maven les déduit grâce à la version de Spring Boot utilisée.

Dans ce cours, nous allons apprendre à choisir les bons starters en fonction du besoin.

**Avantage n° 2 : l’autoconfiguration**

C’est peut-être l’avantage le plus important de Spring Boot ! L’exercice précédent l’a révélé : avec Spring Boot, il y a beaucoup moins de configuration (concernant la gestion des servlets, le chargement du contexte Spring, la connexion à la base de données). Ce n’est pas un hasard. L’utilisation de Spring Boot, et l’annotation @SpringBootApplication placée au niveau de la classe principale de notre projet (celle générée automatiquement), déclenchent automatiquement de nombreuses opérations en background qui nous sont nécessaires.

Le développeur peut alors **se concentrer sur le code métier**au lieu de passer un temps fou à configurer le framework qu’il utilise.

**Avantage n° 3 : la gestion des propriétés**

Spring Boot permet de **gérer les propriétés au sein d’un programme en toute simplicité**.

Dans l’exercice, vous avez pu voir le fichier**applications.properties**. Les informations qui étaient saisies ont été prises en compte par certaines classes, sans que nous ayons besoin d’agir. Ce fichier est **l’un des éléments clés** pour la gestion des propriétés de notre programme.

Mais cela ne se limite pas à ce simple fichier ; par exemple, il est facilement possible de récupérer même des variables d’environnement système, et de les fournir à nos classes.

**Avantage n° 4 : le monitoring et la gestion du programme**

Je ne souhaite pas rentrer dans le détail ici, mais sachez que **Spring Boot Actuator** correspond à une fonctionnalité de Spring Boot qui permet de **monitorer et de manager notre programme** pendant qu’il est en cours d’exécution.

Par exemple, grâce aux **endpoints Actuator**, on peut

+ modifier une propriété en live, et le programme en tiendra compte sans qu’on ait besoin de le redémarrer. Très utile !

**Avantage n° 5 : le déploiement**

Quel est l’artefact produit par un projet Spring Boot ? **Un simple fichier JAR**.

Que faut-il pour exécuter un JAR ? Vous le savez, une JRE, et c’est tout !

Mais comment est-ce possible pour une application web qui a forcément besoin d’un serveur de conteneur web ?

Un projet Spring Boot contient un tomcat embarqué au sein même du JAR généré. Le projet web peut donc être déployé au sein de ce tomcat embarqué.

Pour information, Spring Boot nous offre la possibilité de générer notre projet en WAR et non JAR, si vous en avez la nécessité.

## Créez votre projet

### les starters

La première étape implique de générer la base de votre projet. Vous devez savoir qu’on ne commence pas sur une feuille blanche. Spring Boot nous fournit une base de travail que l’on peut nommer **la *structure minimale***. On enrichira ensuite cette structure minimale en fonction des besoins de notre projet.

Pour obtenir cette structure minimale, il y a plusieurs solutions que l’on explorera dans la suite du chapitre. Mais avant de foncer tête baissée, sachez que Spring Boot va vous demander un certain nombre d’informations, comme :

* la version de Java ;
* Maven ou Gradle ;
* la version de Spring Boot ;
* des informations sur le projet (groupId, artifactId, nom du package) ;
* les dépendances.

Pour les premiers éléments, ce ne sera pas difficile ; ça pourrait par contre le devenir pour le choix des dépendances, car Spring Boot utilise quelque chose de nouveau : **les starters de dépendances**.

Spring Framework se découpe en de nombreux composants ; les utiliser implique de renseigner les bonnes dépendances pour notre projet. Ce n’est pas facile, car il faut savoir quelle dépendance est nécessaire à quelle autre dépendance, s’il y a des incompatibilités, et quelles sont les versions à utiliser.

Pour résoudre cette problématique, Spring Boot nous offre les starters de dépendances qui sont des kits de dépendances (vous vous souvenez de mon exemple avec les gammes de meubles, voilà !  ).

Par exemple, le starter **spring-boot-starter-data-jpa** va vous apporter différents JAR pour utiliser Spring et JPA, afin de communiquer avec une base de données.

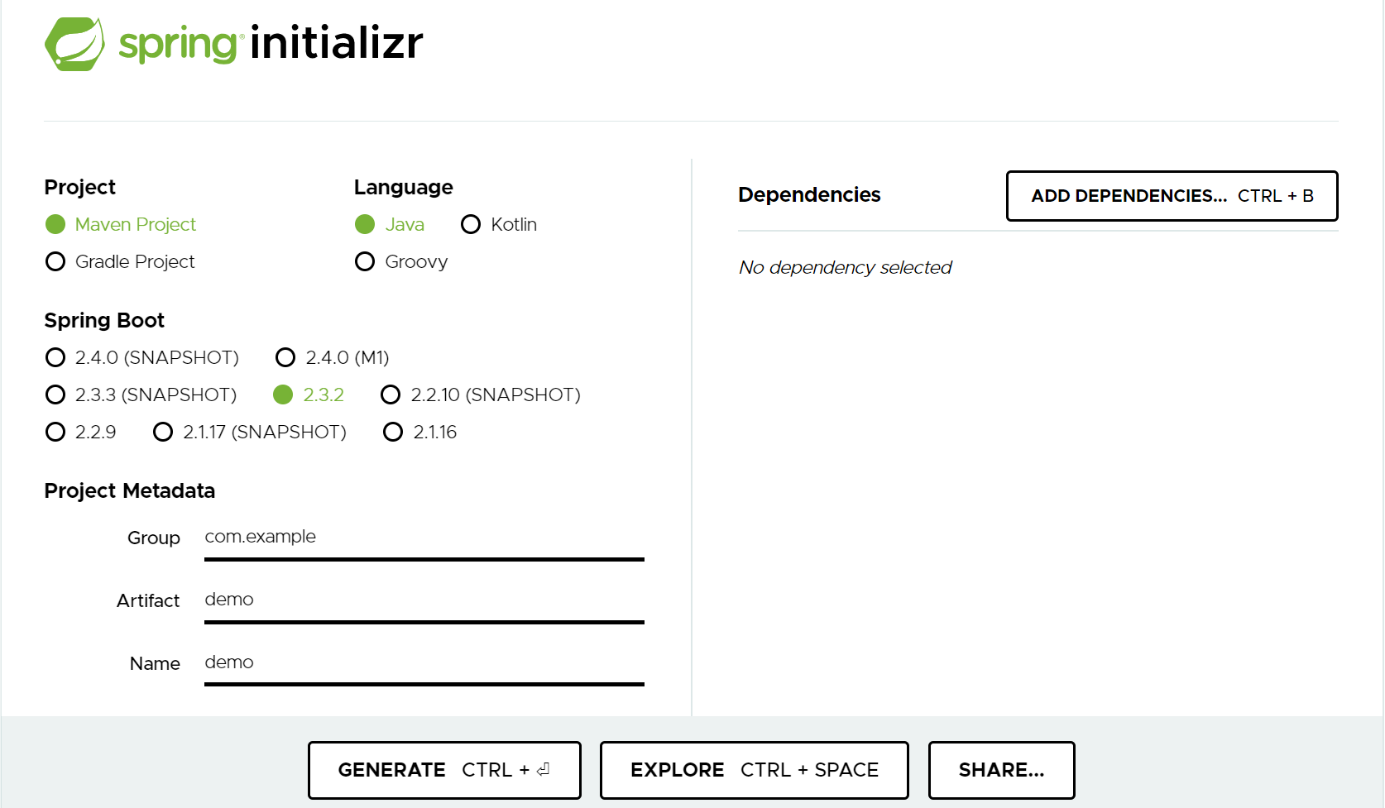
Tous les starters sont préfixés par “spring-boot-starter”. Voici quelques exemples de starters :

* spring-boot-starter-core ;
* spring-boot-starter-data-jpa ;
* spring-boot-starter-security ;
* spring-boot-starter-test ;
* spring-boot-starter-web.

Il existe deux façons de créer un projet : avec Spring Initializr et avec Spring Tool suite. Nous allons aborder ces deux approches ensemble !

### Créez votre projet avec Spring Initializr

Voyons comment utiliser le site web [Spring Initializr](https://start.spring.io/).



Laissez-moi vous montrer son fonctionnement :

C’était vraiment facile, n’est-ce pas ? 🙂 Récapitulons les étapes :

1. J’ai laissé par défaut Project / Language et Spring Boot.
2. J’ai saisi les informations suivantes dans Project Metadata :

* Group: com.openclassrooms.
* Artifact: HelloWorld.
* Name: HelloWorld.
* Description: Hello World project with Spring Boot.
* Package name: com.openclassrooms.helloworld.
* Packaging: Jar.
* Java: 8.

Aucune dépendance à ajouter, car le starter “**spring-boot-starter**”, qui contient toutes les fonctionnalités de base, est ajouté par défaut. J’ai cliqué sur “Generate”, et une archive ZIP avec le projet a été téléchargée. Après avoir dézippé l’archive, j’ai tout simplement importé ce projet existant dans mon IDE.

Voilà, notre première étape est finie ! 

### Créez votre projet avec Spring Tool Suite

Une deuxième façon d'atteindre le même résultat correspond à l’utilisation de l’outil Spring Tool Suite, téléchargeable [ici](https://spring.io/tools).

On retrouve les mêmes étapes, mais cette fois à travers un outil installé sur votre poste de travail, et non via un site web

Mais pourquoi utiliser Spring Tool Suite et non Spring Initializr, qui ne demande aucun outil supplémentaire ?

Pour la simple raison que STS (Spring Tool Suite) nous offre une fonctionnalité supplémentaire : le “**Boot Dashboard**”, qui permet de gérer plus précisément **le cycle de vie de l’exécution de l’application**. Et comme STS est tout simplement un Eclipse customisé, on peut directement développer notre projet au sein de l’outil. Assez pratique, finalement.

pom.xml

Vous y retrouverez tous les éléments saisis lors de la génération de votre projet. En voici d’ailleurs 2 extraits :

Extrait de la description du projet :

<groupId>com.openclassrooms</groupId>

<artifactId>helloworld</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>helloworld</name>

<description>Hello World project with Spring Boot</description>

Extrait des dépendances du projet :

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

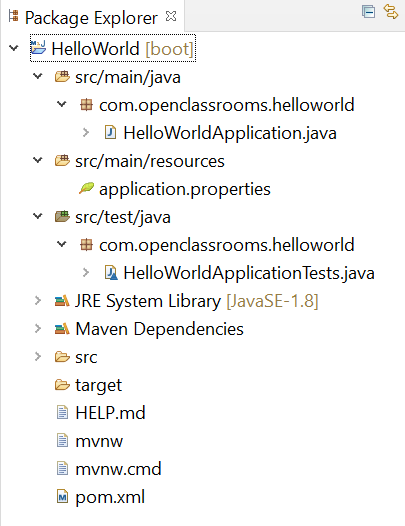
</dependency>

….

</dependencies>

## Observez la structure minimale

Voici le projet à travers la vue Package Explorer de STS. Qu'observez-vous? Comment les sources sont-elles organisées ?

Arborescence du projet via la vue Package Explorer

Voici quelques points à noter :

* Nous retrouvons les fichiers liés à Maven (pom.xml, mvnw et mvnw.cmd).
* Nos sources sont organisées suivant le schéma standard :
  + src/main/java : contient les packages et les classes Java ;
  + src/main/resources : contient les fichiers ressources, tels que les fichiers de propriétés ou des templates (HTML, par exemple) ;
  + src/test/java : contient les classes Java de test.
* Comme tout projet Java/Maven, on retrouve également la JRE et les Maven Dependencies.

N’hésitez pas à aller faire un tour dans les Maven Dependencies, pour voir tout ce que les starters incluent comme dépendances !

Pour ce qui de la classe HelloWorldApplicationTests.java, je ne souhaite pas m’y arrêter non plus à cette étape ; sachez juste pour le moment qu’elle vérifie que le contexte Spring se lance bien comme attendu.

**classe principale**

Maintenant, parlons de HelloWorldApplication.java, qui est **la classe principale** de votre projet. On y retrouve 2 choses importantes :

1. La méthode main dont je ne vous ferai pas l’offense de vous expliquer son rôle en Java !
2. L’annotation @SpringBootApplication qui est critique !

### Découvrez le rôle des annotations et leurs avantages

### Utilisez l’annotation @SpringBootApplication

@SpringBootApplication est un composé de 3 autres annotations :

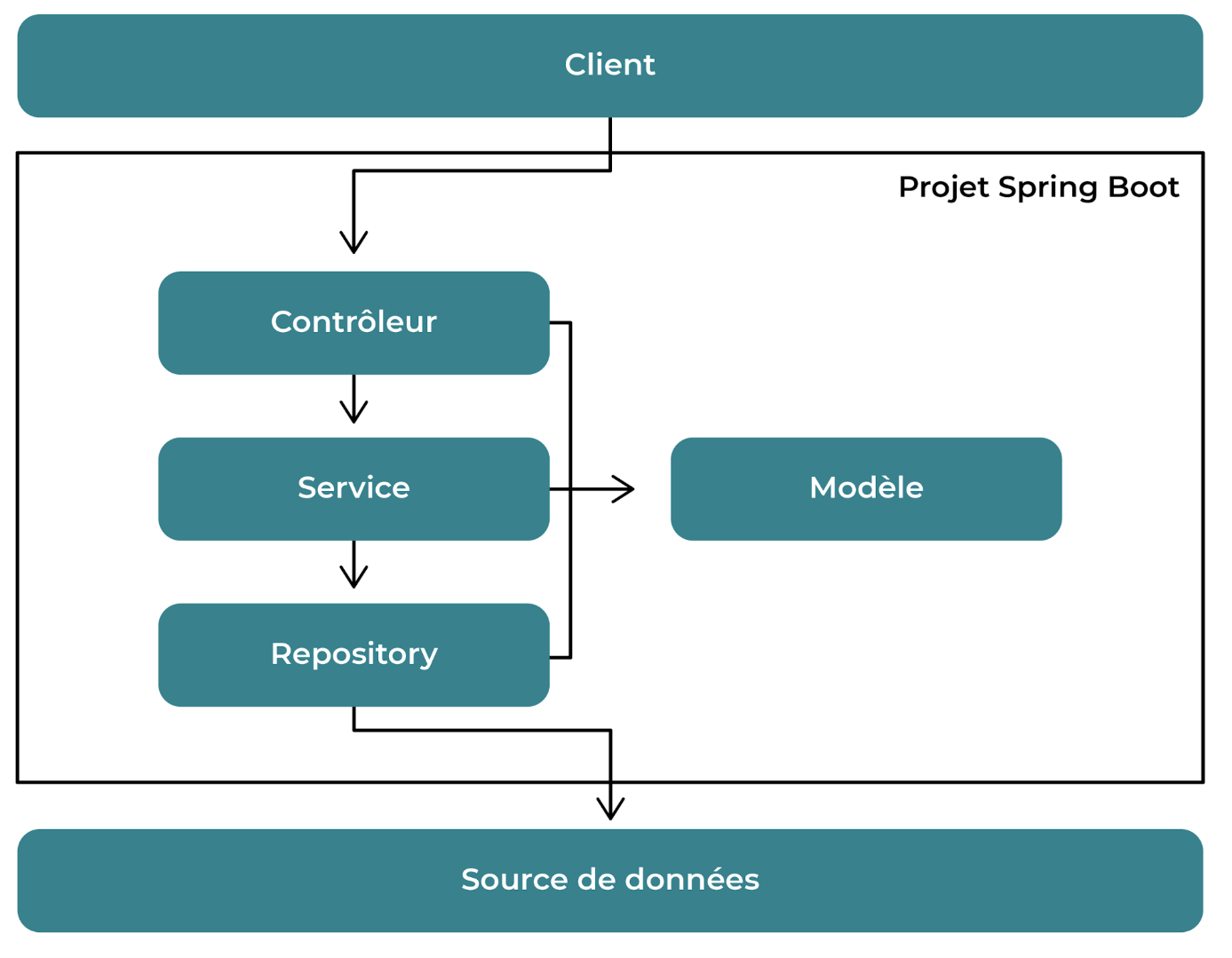
1. **@SpringBootConfiguration**: la classe sera utilisée comme une classe de configuration (on reviendra sur cette notion plus tard).
2. **@EnableAutoConfiguration**: active la fameuse fonctionnalité d’autoconfiguration de Spring Boot, que je vous ai tant vantée.
3. **@ComponentScan** : active le **“scanning”** de classes dans le package de la classe et dans ses sous-packages. Sans cette annotation, l’IoC container ne tiendra pas compte de vos classes, même si vous avez ajouté une annotation sur celles-ci.

**En résumé, cette classe, c’est ce qui déclenche toute la mécanique interne de Spring Boot et des composants Spring associés**. Et tout ça en moins de 15 lignes de code. 

## Structurez et configurez votre projet

### Structurez vos packages

* **couche Controller** : gestion des interactions entre l’utilisateur de l’application et l’application ;
* **couche Service** : implémentation des traitements métiers spécifiques à l’application ;
* **couche Repository** : interaction avec les sources de données externes ;
* **couche Model** : implémentation des objets métiers qui seront manipulés par les autres couches.

 Cette architecture standard correspondra à la majorité de vos projets, et vous la retrouvez très fréquemment.

D’autres packages peuvent être parfois nécessaires, par exemple un nommé “security” pour les classes dédiées à la sécurité, ou “configuration” pour des classes gérant les propriétés. Dans ce cas, cela s'ajoute à la structure existante.

**un bean est une classe au sein du contexte Spring** (l’IoC container).

## Écrivez votre premier Hello World

La méthode *main* sera théoriquement là où on écrirait notre code dans un programme Java simple. Mais en l'occurrence, cette dernière contient l’instruction**“SpringApplication.run(HelloWorldApplication.class, args);”**. Cette instruction permet de démarrer notre application, et ce n’est pas une bonne pratique d’ajouter autre chose dans la méthode *main*.

Oui, mais où met-on notre code, alors ?

Spring Boot fournit une interface nommée “**CommandLineRunner**”. En implémentant cette interface, la classe sera obligée de déclarer la méthode **“public void run(String... args) throws Exception“**. À partir de là, si la classe est un bean (c’est-à-dire chargée dans le contexte Spring), Spring Boot exécutera la méthode run à l’exécution du programme.

Vous pourriez :

* soit modifier la classe HelloWorldApplication afin qu’elle implémente CommandLineRunner et la méthode run, avec comme corps de méthode un “System.out.prinln(“Hello World!”)” ;
* soit créer une nouvelle classe qui implémente CommandLineRunner, la méthode run (même corps de méthode), et qui aura une annotation @Component (au-dessus du nom de la classe).

## 2. Spring ApplicationContext

Before we can understand the value of @Component, we first need to understand a little bit about the [Spring ApplicationContext](https://www.baeldung.com/spring-application-context).

Spring ApplicationContext is where Spring holds instances of objects that it has identified to be managed and distributed automatically. These are called beans.

Bean management and the opportunity for dependency injection are some of Spring's main features.

Using the [Inversion of Control principle](https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring), **Spring collects bean instances from our application and uses them at the appropriate time.** We can show bean dependencies to Spring without needing to handle the setup and instantiation of those objects.

The ability to use annotations like [@Autowired](https://www.baeldung.com/spring-autowire) to inject Spring-managed beans into our application is a driving force for creating powerful and scalable code in Spring.

So, how do we tell Spring about the beans we want it to manage for us? **We should take advantage of Spring's automatic bean detection by using stereotype annotations on our classes.**

## 3. @Component

@Component is an annotation that allows Spring to automatically detect our custom beans.

In other words, without having to write any explicit code, Spring will:

* Scan our application for classes annotated with @Component
* Instantiate them and inject any specified dependencies into them
* Inject them wherever needed

However, most developers prefer to use the more specialized stereotype annotations to serve this function.

### 3.1. Spring Stereotype Annotations

Spring has provided a few specialized stereotype annotations: @Controller, @Service and @Repository. They all provide the same function as @Component.

**They all act the same because they are all composed annotations with @Component as a meta-annotation for each of them.** They are like @Component aliases with specialized uses and meaning outside of Spring auto-detection or dependency injection.

If we really wanted to, we could theoretically choose to use @Component exclusively for our bean auto-detection needs. On the flip side, we could also [compose our own specialized annotations](https://www.baeldung.com/java-custom-annotation) that use @Component.

However, there are other areas of Spring that look specifically for Spring's specialized annotations to provide additional automation benefits. **So, we should probably just stick with using the established specializations most of the time.**

### Abordez la dernière étape : les tests et le déploiement

“Déployer” revient à mettre en route notre application. Autrement dit, il s’agit de **passer de l’environnement de développement à l’environnement de production**.

### Découvrez Spring Boot Test

**@SpringBootTest** est une annotation fournie par Spring Boot. Elle **permet lors de l’exécution des tests d’initialiser le contexte Spring**. Les beans de notre application peuvent alors être utilisés.

Rappelons qu’un test s’exécute de façon unitaire, presque comme une application à part entière. Par défaut, notre test n’a donc aucune connaissance du contexte Spring. Dans le cas d’une application Spring Boot, c’est un vrai problème !

Mais le problème est résolu grâce à l’annotation @SpringBootTest.

La méthode contextLoads est **annotée @Test** (annotation qui provient de JUnit ; quand je vous disais que d’autres frameworks utilisent les annotations 😉), et n’a pas de contenu.

Pourquoi n’a-t-elle pas de contenu ?

Tout simplement parce que son unique objectif est de**vérifier que le contexte Spring se charge bien**.

Sans méthode de tests (c’est-à-dire sans méthode avec @Test), notre classe de test ne peut être exécutée, même si elle est annotée @SpringBootTest. Pour parer à cela, Spring Boot génère une méthode vide annotée @Test, et qui sera donc toujours success pour JUnit (car elle est vide).

Ainsi, lors de l’exécution de cette méthode, le contexte Spring sera chargé, et si ce dernier rencontre une erreur, alors l’exécution de la classe de test retournera une erreur.

Très bien ! Maintenant, testons notre HelloWorld !

D’accord, mais on teste quoi ?

Le but de ce cours n’est pas de vous expliquer les méthodologies de test. Mais rappelez-vous toujours qu’au sein d’un projet, on va **tester les traitements métiers** pour s’assurer qu’ils correspondent bien aux attendus.

Dans notre cas, le traitement métier est représenté par la méthode getHelloWorld() de la classe BusinessService.

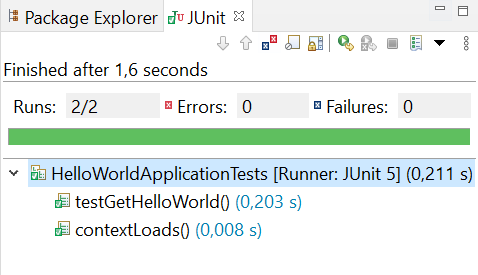
Donc, nous allons tester cette méthode.

Je vous laisse essayer, voici quelques indices :

1. Ajoutez une nouvelle méthode de test.
2. Injectez une instance de BusinessService dans la classe de test.
3. Vérifiez que l’attribut “value” de l’objet HelloWorld contient bien le texte “Hello World!”.

Ensuite, il ne vous reste plus qu’à faire un clic droit sur la classe, “Run As”, “JUnit Test”.

La vue JUnit au sein de STS devrait ressembler à ça :

Fenêtre JUnit de l’IDE qui affiche le résultat des tests

Avez-vous réussi ? Super ! Félicitations :-)

Voici tout de même une correction :

package com.openclassrooms.helloworld;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.context.SpringBootTest;

import com.openclassrooms.helloworld.service.BusinessService;

@SpringBootTest

class HelloworldApplicationTests {

@Autowired

private BusinessService bs;

@Test

void contextLoads() {

}

@Test

public void testGetHelloWorld() {

String expected = "Hello World!";

String result = bs.getHelloWorld().getValue();

assertEquals(expected, result);

}

}

### Découvrez les méthodes de déploiement

Notre objectif est désormais de déployer et ainsi d’exécuter notre application.

Le déploiement correspond aux étapes qui permettent de passer de l’environnement de développement à l’environnement d’exécution.

L’environnement de développement correspond généralement à notre IDE. Dans le cas de ce cours, on utilise STS.

L'environnement d'exécution peut varier. Il peut s'agit d'un environnement de tests ou d'un environnement de production. Notons que l’environnement de développement peut également être un environnement d’exécution.

Quel prérequis doit avoir un environnement d’exécution ?

Je suis sûr que vous connaissez la réponse : tout simplement une JRE, qui permet ainsi l’exécution d’un programme Java.

Même si c’est une application Spring, la simple JRE suffit ?

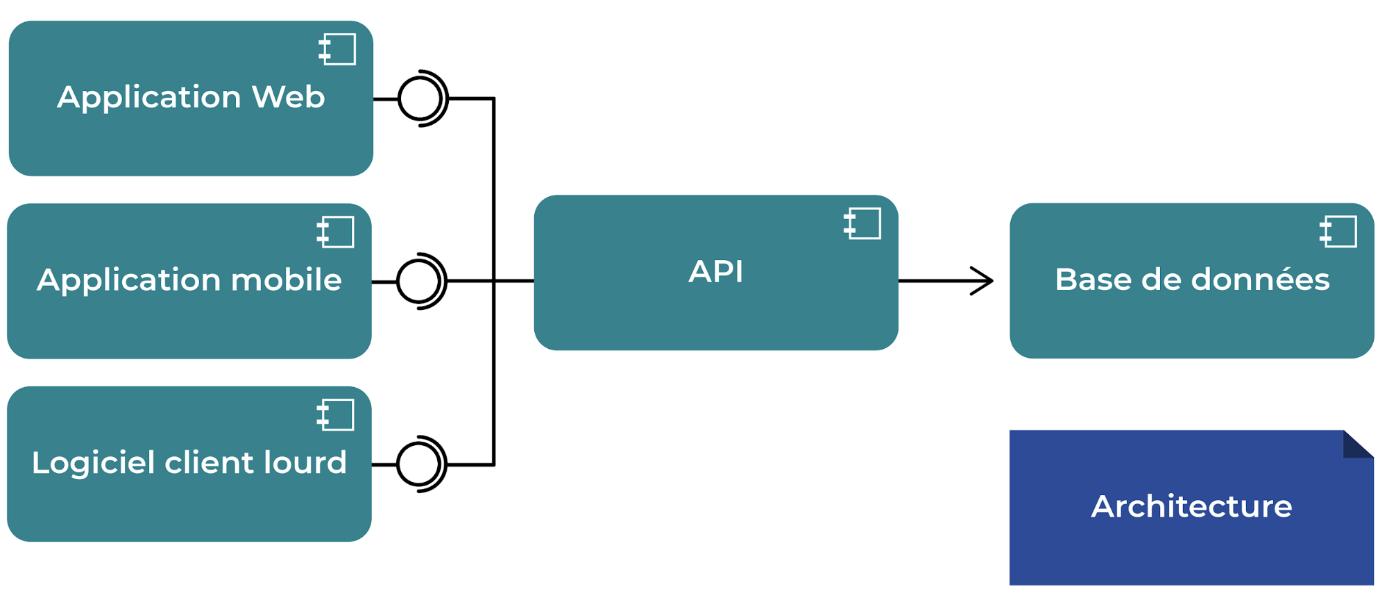
Tout à fait, et c’est une très bonne nouvelle ! Rappelons que parmi les avantages de Spring Boot, il y a sa facilité de déploiement car le JAR qui résulte de la compilation embarque tout.

Par exemple, même le serveur Tomcat qui permet d’exécuter une application web est embarqué. Nul besoin d’installer un serveur Tomcat, il est déjà là !

* Il existe plusieurs méthodes pour exécuter notre application Spring Boot :
  + Via l’IDE directement.
  + Grâce à Maven et au goal spring-boot:run.
  + En exécutant la commande java -jar.

## Créez l'API avec les bons starters

Voici l’architecture imaginée :

Vous allez réaliser une **API**. Rappelons-le, une API est un programme qui a pour vocation de**communiquer avec d’autres programmes**. L’idée étant de gérer **des employés**, l’API devra donc offrir un **CRUD**(Create, Read, Update, Delete) pour les données des employés.

Les données seront dans**une base de données H2** (pour ceux qui sont un peu initiés aux bases de données, n’hésitez pas à faire du MySQL, PostgreSQL ou autre).

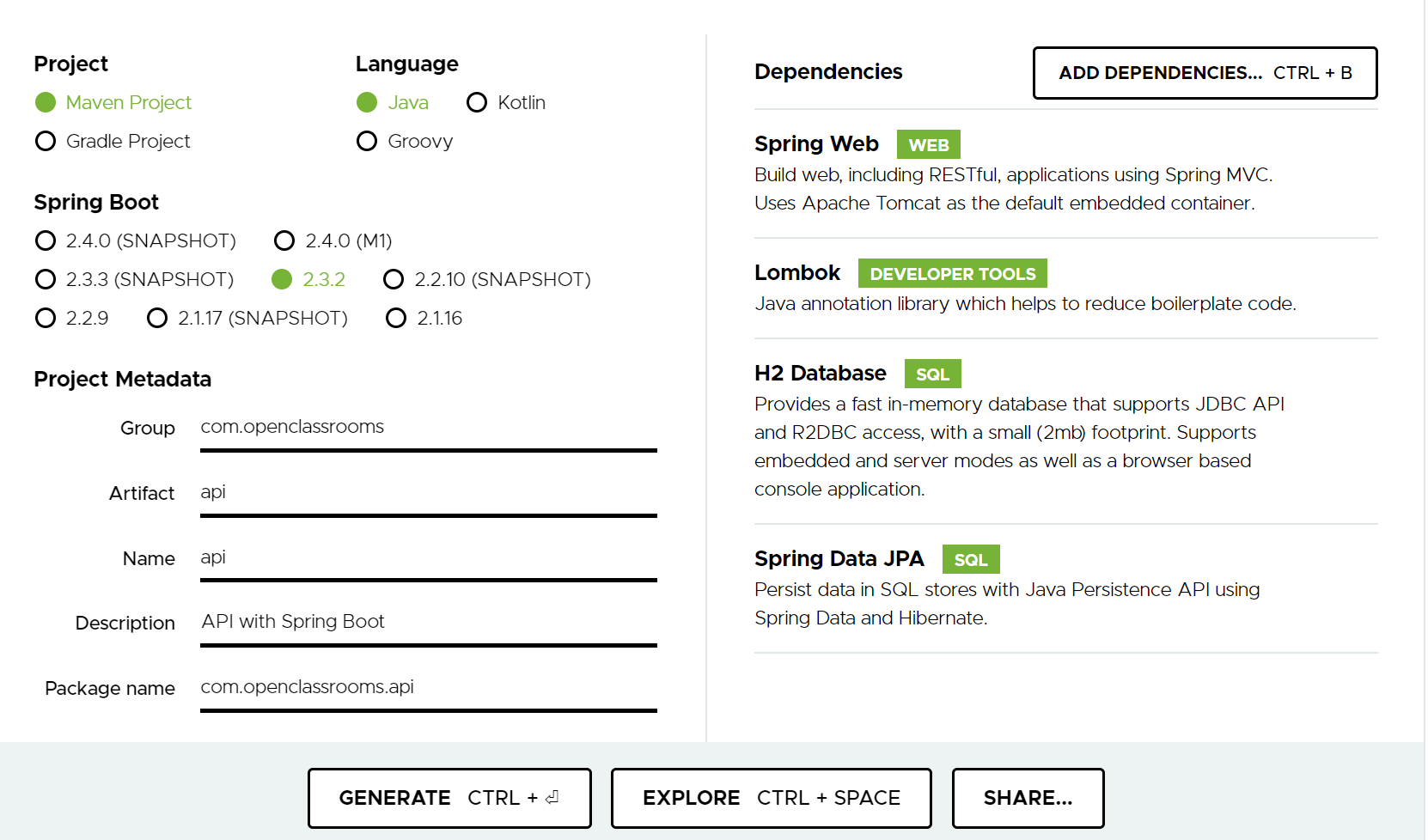
H2 est une base de données relationnelle Java très légère, qui par défaut fonctionne en “in memory”. Cela signifie qu’au démarrage du programme, la structure de la base est construite ; et lorsque le programme s’arrête, le contenu de la base de données est supprimé. On ne conserve donc pas les modifications apportées d’une exécution à l’autre du programme.

Notre API devra donc**exposer des endpoints** correspondant aux actions du CRUD, et **communiquer avec la base de données** pour récupérer ou modifier les informations des employés. À noter que l’API sera de type **REST**.

### identifiez les bons starters pour créer l'API

Alors, avez-vous trouvé votre bonheur dans les starters de Spring Boot ?

Voici ma solution :

Formulaire du site Spring Initializr complété pour le projet

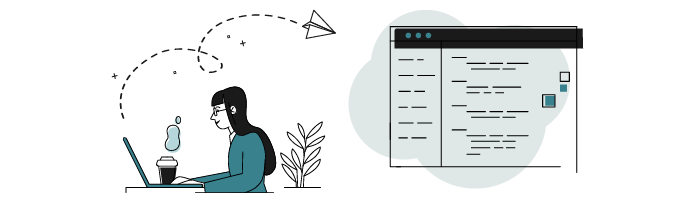
Quelques explications :

* Pour les “Project Metadata”, je suis resté simple :
  + group: com.openclassrooms (un standard) ;
  + artifact: api (on “appelle un chat un chat”, n’est-ce pas ?).
* Pour les dépendances :
  + **Spring Web**: comme la description l’indique, permet de faire du RESTful, ce qui correspond à notre API pour exposer des endpoints.
  + **Lombok** : vous ne pouviez pas le deviner, c’est une librarie pour optimiser certaines classes, je vous en parle un peu plus loin dans le chapitre.
  + **H2 Database** : comme on dit, “C'est comme le Port-Salut, c'est écrit dessus” ; on veut faire du H2, alors on prend cette dépendance.
  + **Spring Data JPA**: permet de gérer la persistance des données avec la base de données ; peut-être le plus difficile à identifier pour vous si vous n’avez jamais fait de persistance de données avec Spring.
* La structure des packages reste le standard : **controller / service / repository / model**.

[Licence](https://creativecommons.org/licenses/by-sa/4.0" \t "_blank)

Mis à jour le 27/09/2021

## Créez un contrôleur REST pour gérer vos données



Dans ce chapitre, oublions le mode exercice/correction, je ne veux pas vous faire chercher des heures et des heures si vous n’avez jamais eu l’occasion de développer une API REST avec Spring. Je vous propose donc de vous montrer **pas à pas comment arriver au résultat final**, et je vous invite à appliquer les étapes en même temps que moi.

Let’s go !

### Modélisez la table en entité Java

Première chose à faire, afin de manipuler les données persistées dans la base de données, nous allons créer une classe Java qui est une entité. Cela signifie que**la classe représente la table**. Chaque ligne de la table correspondra à une instance de la classe. Créons la classe **Employee**dans le package com.openclassrooms.api.model.

Voici la classe en question :

package com.openclassrooms.api.model;

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 lombok.Data;

@Data

@Entity

@Table(name = "employees")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name="first\_name")

private String firstName;

@Column(name="last\_name")

private String lastName;

private String mail;

private String password;

}

Arrêtons-nous sur ses spécificités :

* @Data est une annotation Lombok. Nul besoin d’ajouter les getters et les setters. La librairie Lombok s’en charge pour nous. Très utile pour alléger le code.
* @Entity est une annotation qui indique que la classe correspond à une table de la base de données.
* @Table(name=”employees”) indique le nom de la table associée.
* L’attribut id correspond à la clé primaire de la table, et est donc annoté @Id. D’autre part, comme l’id est auto-incrémenté, j’ai ajouté l’annotation @GeneratedValue(strategy = GenerationType.IDENTITY).
* Enfin, firstName et lastName sont annotés avec @Column pour faire le lien avec le nom du champ de la table.

Je n’ai pas eu besoin de mettre cette annotation pour mail et password, car le nom du champ de la table est identique.

### Implémentez la communication entre l’application et la base de données

Une fois que l’entité est créée, nous devons implémenter le code qui déclenche les actions pour communiquer avec la base de données. Bien évidemment, ce code se servira de notre classe entité.

Le principe est simple, notre code fait une requête à la base de données, et le résultat nous est retourné sous forme d’instances de l’objet Employee.

Quel est ce code à implémenter ?

**Une interface !**

Mais une interface ne contient pas de code, comment peut-elle exécuter des requêtes ?

C’est là toute la puissance du composant **Spring Data JPA**! Il nous permet d’exécuter des requêtes SQL, sans avoir besoin de les écrire.

Dans le package com.openclassrooms.api.repository, créez une **interface** nommée **EmployeeRepository**. Le code sera le suivant :

package com.openclassrooms.api.repository;

import org.springframework.data.repository.CrudRepository;

import org.springframework.stereotype.Repository;

import com.openclassrooms.api.model.Employee;

@Repository

public interface EmployeeRepository extends CrudRepository<Employee, Long> {

}

Comme à notre habitude, expliquons : @Repository est une annotation Spring pour indiquer que la classe est un bean, et que son rôle est de communiquer avec une source de données (en l'occurrence la base de données).

En réalité, **@Repository** est une spécialisation de **@Component**. Tout comme @Component, elle permet de déclarer auprès de Spring qu’une classe est un bean à exploiter. Par son nom, on fournit au développeur une indication sémantique sur le rôle de la classe. Vous pourriez cependant utiliser l’annotation @Component, et cela fonctionnerait très bien, mais il est recommandé d’utiliser les annotations qui offrent cette valeur sémantique supplémentaire.

CrudRepository est une interface fournie par Spring. Elle fournit des méthodes pour manipuler votre entité. Elle utilise la généricité pour que son code soit applicable à n’importe quelle entité, d’où la syntaxe “CrudRepository**<Employee, Long>**” (je ne vous fais pas un cours sur la généricité, cela fait partie de vos acquis de développeur Java ).

La classe entité fournie doit être annotée @Entity, sinon Spring retournera une erreur.

Ainsi, vous pouvez utiliser les méthodes définies par l’interface CrudRepository. Pour en avoir la liste complète, rendez-vous sur [la documentation](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html). Je vous en dis plus dans la partie qui suit !

JpaRepository vs CrudRepository

[JpaRepository](http://static.springsource.org/spring-data/data-jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html) extends [PagingAndSortingRepository](http://static.springsource.org/spring-data/data-commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html) which in turn extends [CrudRepository](http://static.springsource.org/spring-data/data-commons/docs/current/api/org/springframework/data/repository/CrudRepository.html).

Their main functions are:

* [CrudRepository](http://static.springsource.org/spring-data/data-commons/docs/current/api/org/springframework/data/repository/CrudRepository.html) mainly provides CRUD functions.
* [PagingAndSortingRepository](http://static.springsource.org/spring-data/data-commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html) provides methods to do pagination and sorting records.
* [JpaRepository](http://static.springsource.org/spring-data/data-jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html) provides some JPA-related methods such as flushing the persistence context and deleting records in a batch.

Because of the inheritance mentioned above, JpaRepository will have all the functions of CrudRepository and PagingAndSortingRepository. So if you don't need the repository to have the functions provided by JpaRepository and PagingAndSortingRepository , use CrudRepository.

### Créez un service métier

Si nous analysons la structure créée, nous avons tiré profit des couches model et repository, mais rien n’a été fait dans les autres couches. Il reste donc “service” et “controller” à implémenter.

La couche “service” est dédiée au **“métier”**. C’est-à-dire appliquer des traitements dictés par les règles fonctionnelles de l’application. Imaginez par exemple que votre application doive sauvegarder tous les noms des employés en majuscules, mais que le nom nous arrive en minuscules.

À quel endroit de notre code allons-nous effectuer l’opération de mise en majuscules ? Vous l’avez compris, dans la couche “métier”.

Ainsi, lorsqu’une requête est réceptionnée, la couche controller délègue les traitements métiers à exécuter. La couche service pourra ensuite faire appel à la couche repository. La couche service est également un pont entre le controller et le repository. De ce fait, nous allons créer une classe EmployeeService dont voici le code :

package com.openclassrooms.api.service;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.openclassrooms.api.model.Employee;

import com.openclassrooms.api.repository.EmployeeRepository;

import lombok.Data;

@Data

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public Optional<Employee> getEmployee(final Long id) {

return employeeRepository.findById(id);

}

public Iterable<Employee> getEmployees() {

return employeeRepository.findAll();

}

public void deleteEmployee(final Long id) {

employeeRepository.deleteById(id);

}

public Employee saveEmployee(Employee employee) {

Employee savedEmployee = employeeRepository.save(employee);

return savedEmployee;

}

}

Vous noterez avec intérêt que chaque méthode a pour unique objectif d’appeler une méthode de l'employeeRepository.

Un petit zoom sur l’annotation **@Service**: tout comme l’annotation @Repository, c’est **une spécialisation de @Component**. Son rôle est donc le même, mais son nom a une valeur sémantique pour ceux qui lisent votre code. Pour le reste, ce code devrait être à votre portée.

### Exposez les endpoints REST

Le starter “spring-boot-starter-web” nous fournit justement tout le nécessaire pour créer un endpoint. Laissez-moi vous donner la recette. Il faut :

* une classe Java annotée **@RestController**;
* que les méthodes de la classe soient annotées. Chaque méthode annotée devient alors un endpoint grâce aux annotations **@GetMapping, @PostMapping, @PatchMapping, @PutMapping, @DeleteMapping, @RequestMapping**.

Premièrement, @RestController atteint 2 objectifs :

1. Comme @Component, elle permet d’indiquer à Spring que cette classe est un bean.
2. Elle indique à Spring d**’insérer le retour de la méthode au format JSON dans le corps de la réponse HTTP**. Grâce à ce deuxième point, les applications qui vont communiquer avec votre API accéderont au résultat de leur requête **en parsant la réponse HTTP**.