Documentation sur l'API

Introduction

Spring Boot est un projet ou un micro framework qui a notamment pour but de faciliter la configuration d'un projet Spring et de réduire le temps alloué au démarrage d'un projet. Pour arriver à remplir cet objectif, Spring Boot se base sur plusieurs éléments :

- Un site web (https://start.spring.io/) qui nous permet de générer rapidement la structure de notre projet en y incluant toutes les dépendances Maven nécessaires à notre application.
- L'auto-configuration, qui applique une configuration par défaut au démarrage de notre application pour toutes dépendances présentes dans celle-ci. Cette configuration s'active à partir du moment où nous avons annoté notre application avec « @EnableAutoConfiguration » ou « @SpringBootApplication ». Bien entendu cette configuration peut être surchargée via des propriétés Spring prédéfinie ou via une configuration Java. L'auto-configuration simplifie la configuration sans pour autant vous restreindre dans les fonctionnalités de Spring. Par exemple, si vous utilisez le starter « spring-boot-starter-security », Spring Boot vous configurera la sécurité dans votre application avec notamment un utilisateur par défaut et un mot de passe généré aléatoirement au démarrage de votre application.

Spring Boot offre d'autres avantages notamment en termes de déploiement applicatif. Habituellement, le déploiement d'une application Spring nécessite la génération d'un fichier .war qui doit être déployé sur un serveur comme un Apache Tomcat. Spring Boot simplifie ce mécanisme en offrant la possibilité d'intégrer directement un serveur Tomcat dans notre exécutable. Au lancement de celui-ci, un Tomcat embarqué sera démarré afin de faire tourner votre application.

Enfin, Spring Boot met à disposition des opérationnels, des métriques qu'ils peuvent suivre une fois l'application déployée en production. Pour cela Spring Boot utilise « Actuator » qui est un système qui permet de monitorer une application via des URLs spécifiques ou des commandes disponibles via SSH. Sachez, qu'il est possible de définir vos propres indicateurs très facilement.

Pom.xml

Chaque projet ou sous-projet est configuré par un POM qui contient les informations nécessaires à Maven pour traiter le projet (nom du projet, numéro de version, dépendances vers d'autres projets, bibliothèques nécessaires à la compilation, noms des contributeurs etc.). Ce POM se matérialise par un fichier pom.xml à la racine du projet. Cette approche permet l'héritage des propriétés du projet parent. Si une propriété est redéfinie dans le POM du projet, elle recouvre celle qui est définie dans le projet parent. Ceci introduit le concept de réutilisation de configuration. Le fichier pom du projet principal est nommé pom parent.

Les trois lettres POM sont l'acronyme de *Project Object Model*. Sa représentation XML est traduite par Maven en une structure de données qui représente le modèle du projet. Ces déclarations sont complétées par l'ensemble des conventions qui viennent ainsi former un modèle complet du projet utilisé par Maven pour exécuter des traitements.

La première partie du POM permet d'identifier le projet lui-même :

```
<?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
https://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
```

- L'élément *modelVersion* permet de savoir quelle version de la structure de données "modèle de projet" est représentée dans le fichier XML. Les futures versions de Maven pourront ainsi exploiter des versions différentes de modèles en parallèle et introduire si nécessaire des évolutions dans le format de ce fichier.
- L'identifiant de groupe *(groupId)* permet de connaitre l'organisation, l'entreprise, l'entité ou la communauté qui gère le projet. Par convention, on utilise le nom de domaine Internet inversé, selon la même logique que celle généralement recommandée pour les noms de packages Java.
- L'identifiant de composant *(artifactId)* est le nom unique du projet au sein du groupe qui le développe. En pratique et pour éviter des confusions, il est bon d'avoir un *artifactId* unique indépendamment de son *groupId*.
- Enfin l'élément *version* permet de préciser quelle version du projet est considérée.La plupart des projets utilisent la formule *<Version Majeure>.<Version Mineure>.<Correctif>*.

La deuxième partie du POM concerne les bibliothèques dont dépend le projet :

La dépendance spring-boot-starter-parent permet de rapatrier la plupart des dépendances du projet. Sans elle, le fichier pom.xml serait plus complexe.

La dépendance spring-boot-starter-web indique à Spring Boot qu'il s'agit d'une application web, ce qui permet à Spring Boot de rapatrier les dépendances comme SpringMVC, SpringContext, et même le serveur d'application Tomcat, etc.

Vous avez aussi constaté la présence de la dépendance spring-boot-starter-tomcat qui n'est pas obligatoire, mais comme il s'agit d'une application web, Spring Boot par anticipation intègre un serveur d'application Tomcat afin de faciliter le déploiement de l'application. Cette dépendance n'étant pas obligatoire, vous pouvez la supprimer. Dans notre cas, on va utiliser un serveur Tomcat externe à travers une configuration Eclipse.

Il faut noter aussi l'absence des versions dans les dépendances. Ceci est dû au fait que Spring Boot gère de manière très autonome les versions et nous n'avons plus besoin de les déclarer.

La dépendance spring-boot-starter-data-jpa est nécessaire pour le développement de la couche de persistance. Les autres dépendances seront ajoutées notamment pour les besoins de tests.

La troisième partie du POM concerne la construction du projet :

```
<build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
```

- L'approche déclarative utilisée par Maven permet de définir l'emplacement des fichiers sources. Si les conventions sur les noms de répertoires ont été respectées, ce bloc *<build>* n'est pas nécessaire.

CashManagerApplication.java et ServletInitializer.java

Le point d'entrée de l'application est la classe CashManagerApplication.java.

L'application est initialisée par la classe **ServletInitializer.**

La classe **ServletInitializer** permet l'initialisation de l'application. Elle étend la classe **SpringBootServletInitializer** qui est à l'origine de cette initialisation et remplace ainsi l'ancien fichier **web.xml**.

La classe **CashManagerApplication** contient la méthode ***void main(String[] args)*** nécessaire dans une application Spring Boot, et permet l'exécution de celle-ci : c'est le point d'entrée de l'application.

Annotation contenues dans ses fichiers :

@EnableJpaAuditing : Spring Data JPA est livré avec un écouteur d'entité qui peut être utilisé pour déclencher la capture des informations d'audit. Pour activer la fonctionnalité d'audit, il nous suffit d'ajouter cette annotation à notre configuration.

@SpringBootApplication : cette annotation regroupe les 3 annotations suivantes :

1. @Configuration pour activer la configuration basée sur Java
2. @ComponentScan pour activer l'analyse des composants.
3. @EnableAutoConfiguration pour activer la fonctionnalité de configuration automatique de Spring Boot.

Modèle :

Ce package contient les quatres classe (et donc tables de notre base de données) utilisées dans notre projet.

User.java : classe utilisateur avec implémentation de la table UTILISATEUR dans la base de données.

Panier.java : classe panier avec implémenation de la table PANIER dans la base de données.

Article.java : classe article avec implémentation de la table ARTICLE dans la base de données.

Paiment.java : classe paiment avec implémentation de la table PAIMENT dans la base de données.

Annotations contenues dans ces fichiers :

- les annotations @Entity @Table @Id @Column @GenerateValue de la bibliothèque javax.persistence permettent d'implementer la table UTILISATEUR dans la base de données.

- l'annotation @XmlElement de la bibliothèque javax.xml.bind.annotation : définit le nom de l'élément XML qui sera utilisé

- l'annotation @XmlRootElement javax.xml.bind.annotation mappe une classe ou un type enum sur un élément XML.

- Le mot-clé @override est utilisé pour définir une méthode qui est héritée de la classe parente. On ne l'utilise donc que dans le cas de l'héritage (Méthode toString hérite de la classe Object

- L'annotation ManyToMany spécifie une relation plusieurs à plusieurs

- La signification de CascadeType.ALL est que la persistance propagera (en cascade) toutes les EntityManageropérations (PERSIST, REMOVE, REFRESH, MERGE, DETACH) aux entités associées.

Repository

Continent 4 fichiers (interfaces) qui correspondent à chaque fichier du package modèle.

Repositories sont des interfaces héritant de l'interface Repository. L'objectif de ces interfaces consiste à rendre la création de la couche d'accès aux données (requêtes SELECT, UPDATE...) plus rapide.

On n'est donc pas obligés d'écrire des requêtes supplémentaires pour retrouver une entité, par exemple, par son identifiant ou de retrouver toutes les entités disponibles. Les méthodes findById() et findAll() sont là pour cela. En plus, on peut utiliser les méthodes liées au CRUD (Create, Read, Update, Delete) sans aucun effort supplémentaire.

En outre, la recherche des éléments par des simples clauses WHERE est possible grâce au modèle findByAddress() où "Address" signifie l'attribut address de l'entité qu'on veut récupérer.

On peut également créer des requêtes plus complexes. Pour cela, on emploiera l'annotation @Query directement au-dessus de la méthode qui doit exécuter la requête. On privilégiera cette technique dans notre application de test à cause de sa facilité de compréhension. Les requêtes seront écrites en JPQL.

Controller :

Package contenant quatre contrôleurs des quatre classes contenues dans modèle. Un contrôleur (Controller) contient la logique concernant les actions effectuées par l'utilisateur. Pour une classe user par exemple, UserController permettra d'afficher tous les utilisateurs de notre bases de données, ajouter d'autres utilisateurs, supprimer des utilisateurs ou mettre à jour à l'aide de route qu'on implémentera sur localhost:8080/api.

Annotations contenues dans ces fichiers :

L'annotation @GetMapping(value = "/") est une nouvelle annotation introduite par Spring qui remplace l'annotation classique @RequestMapping et correspond exactement à @RequestMapping(method=RequestMethod.GET, value = "/").

La classe est annotée @Controller afin de permettre à Spring d'enregistrer cette classe comme un contrôleur, et surtout de mémoriser les requêtes que cette classe est capable de gérer.

L'utilisation de la classe ResponseEntity de Spring permet de prendre en compte la gestion des statuts HTTP de réponses. On peut toutefois utiliser la classe ResponseBody pour traiter les réponses si on n'a pas besoin d'exploiter les codes de réponses HTTP. Dans notre cas, je préfère ResponseEntity. On peut donc dire que ResponseEntity = ResponseBody + HttpStatus

Application.properties

Les fichiers de propriétés sont utilisés pour conserver le nombre 'N' de propriétés dans un seul fichier afin d'exécuter l'application dans un environnement différent. Dans Spring Boot, les propriétés sont conservées dans le fichier **application.properties** sous le chemin de classe.

Le fichier application.properties se trouve dans le répertoire **src / main / resources**

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

Ces lignes servent à donner un login et un mot de passe et un URL pour notre base de données

spring.jpa.show-sql=true

spring.h2.console.enabled=true

Ces lignes servent à déclarer notre base de données de type h2 console.