Développement Web avec Java (et Spring)

Introduction

Pour faire du Web en Java, il ya plusieurs possibilités les principales sont :

- JakartaEE (anciennement JavaEE) : un ensemble de spécification pour des APIs de développement d'applications d'entreprise en Java
- Spring : un framework Java pour faciliter le développement de tout types d'applications

Ici, nous allons utiliser Spring car il est très facile à aborder et on est très rapidement productif avec. Il est très utilisé dans l'industrie. JakartaEE est aussi intéressant à connaître, vous aurez un cours d'introduction dessus au S4.

Spring est un très grand framework avec beaucoup de fonctionnalités, dont des fonctionnalités de développement d'application web MVC, que nous allons utiliser dans ce tutoriel.

Mettre en place un environnement de développement Java

Installer un JDK

Avec Winget

```
winget install Microsoft.OpenJDK.17
```

Avec Chocolatey

```
choco install openjdk17
```

Avec APT

```
sudo apt install openjdk-17-jdk
```

Avec Homebrew

```
brew install --cask oracle-jdk
```

Manuellement

Suivez ce lien, téléchargez et exécutez l'installeur correspondant à votre plateforme.

Environnement de développement

Ce tutoial n'est pas dépendant d'un IDE particulier, il utilise la CLI Maven pour exécuter les projets. Ainsi, vous pouvez utiliser n'importe quel IDE ou éditeur de texte. Je recommande les suivants :

- Jetbrains IDEA : IDE Jetbrains dédié au développement Java
- Visual Studio Code avec le pack de plugins Java" de RedHat

Présentation de Spring et Spring Boot

Spring Boot ?

Spring Boot est une des fonctionnalités de Spring qui permet de créer une application "stand-alone" (sans avoir besoin de serveur d'application) et de l'autoconfigurer pour des besoin simple. Cela permet de démarrer rapidement sans passer trop de temps à configurer son projet.

Le pattern MVC

Comme son nom l'indique, ce framework se base sur le pattern MVC, qui divise l'architecture de l'application en trois parties :

- Modèle : les données de l'application
- Vue : les pages web de l'application

- Controlleur : gestion des actions des utilisateurs, coordonne données et pages

Créer un projet Web

D'abord rendez vous sur start.spring.io, il s'agit d'un configurateur de projets spring, très utile pour démarrer un projet rapidement.

Voici les différence options à prendre :

- Project : Maven Project
- Language : Java
- Spring Boot : 2.6.0
- Projet Metadata :
  - Group : sous la forme `fr.<votre pseudo>` (c'est une convention)
  - Artifact : nom de votre projet
  - Name : nom de votre projet
  - Description : un texte libre décrivant votre projet
  - Package Name : laissez la valeur générée
  - Packaging : JAR
  - Java : 17 (vérifiez bien que vous avez Java 17 si vous l'aviez déjà avant ce tuto)
- Dépendencies : cliquez sur "Add Dependencies" et séléctionnez "Spring Web", ainsi que "Thymeleaf" (Thymleaf est le moteur de template que nous allons utiliser pour les vues).

Explication du layout du projet

- Dossier `.mvn` : wrapper Maven qui nous permet d'utiliser Maven (le build tool du projet) sans avoir besoin de l'installer sur notre système (on ne va jamais y toucher).
- Dossier `src/main/java/<package name de votre projet>` : package racine de nos source, on vas y mettre nos packages et nos classes Java.
- `src/main/java/<nom du projet>Application.java` : fichier Java qui contient la méthode `main` de notre application Spring
- Dossier `src/main/resources/static` : fichiers servis statiquement par notre application
- Dossier `src/main/resources/templates` : c'est ici qu'on va ranger nos templates de vues
- Fichier `src/main/resources/application.properties` : dans ce fichier on va définir la configuration de notre application
- Dossier `src/main/test` : Dossier contenant les classes de tests. Nous ne couvrirons pas le sujet des tets dans ce tuto
- Fichier `mvnw` et `mvnw.cmd` : script du wrapper maven, on va l'appeler pour lancer et construire notre projet (`mvnw` pour mac/linux et `mvw.cmd` pour windows).
- Ficheir `pom.xml` : descripteur Maven de notre projet, il contient les méta-données du projet, ainsi que les dépendances

Controlleurs & Routes

Commencez par créer un nouveau package `controller` (sous `src/main/java/<package name de votre projet>`). Pour créer un controlleur, créez une classe dont le nom finit par "Controller" dans ce package. Par exemple, "HelloWorldController", et annotez cette classe avec `@Controller` par exemple :

```
package fr.arsenelapostolet.springmvcapptuto.controller;


import org.springframework.stereotype.Controller;


@Controller
public class HelloWorldController {


}
```

Pour ajouter un préfixe de route à votre controleur, mettez lui l'attribut [RequestMapping("maroute")], par exemple :

```
@RequestMapping("hello-world")
@Controller
public class HelloWorldController {


}
```

Ensuite, vous pouvez créer une méthode endpoint dans la classe controleur, avec l'annotation `@GetMapping` :

```
    @GetMapping
    @ResponseBody
    public String HelloWorld() {
        return "Hello world !";
    }
```

> Ici, on doit également ajouter `@ResponseBody` pour que notre string soit utilisée telle qu'elle et non comme un nom de template. Nous n'aurons plus besoin de la rajouter quand nous utiliserons les templates.

La route de ce endpoint est donc `GET /hello-world`. On peut constater en lançant l'application avec un :

```
./mvnw.cmd spring-boot:run
```

Et en en allant à https://localhost:8080/hello-world

On peut donner des suites de route particulières aux méthodes endpoint :

```
@GetMapping("fr")
@ResponseBody
public string HelloWorldFr() {
    return "Bonjour, le monde";
}

@GetMapping("en"))
@ResponseBody
public string HelloWorldEn() {
    return "Hello, world";
}
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `./mvnw.cmd spring-boot:run`). Et en allant à :

- [https://localhost:8080/hello-world/fr]
- [https://localhost:8080/hello-world/en]

Paramètres de requête et endpoints

Paramètre de requête

Pour récupérer une paramètre de requête, il suffit de rajouter un paramètre annoté avec `@RequestParam` à la méthode endpoint, et de passer en paramètre de l'annotation le nom du paramètre :

```
@GetMapping
@ResponseBody
```

```
    public String HelloWorld(@RequestParam("name") String name) {
        return "Hello, " + name + " !";
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `./mvnw.cmd spring-boot:run`). Et en allant à https://localhost:8080/hello-world?name=Arsène.

Paramètre de chemin

Pour récupérer un paramètre de chemin, il suffit de mettre ce paramètre entre accolades dans la route du endpoint, puis ajouter un paramètre annoté avec `@PathVariable` à la méthode endpoint, et de passer en paramètre de l'annotation le nom du paramètre :

```
    @GetMapping("{id}")
    @ResponseBody
    public String HelloWorld(@PathVariable("id") int id) {
        return "Id : " + id;
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `./mvnw.cmd spring-boot:run`). Et en allant à https://localhost:8080/hello-world/1.

Vues

Avant de retourner une vue, il faut d'abord en créer dans le dossier `src/main/resources/templates`. On va donc créer dedans un fichier `hello-world.html`, qui sera notre template :

```
<h1>Hello, world</h1>
```

Pour le retourner depuis une méthode endpoint, il faut retourner le nom du fichier (sans l'extension), et ne pas mettre l'annotation `@RequestBody` :

```
    @GetMapping
    public String HelloWorld() {
        return "hello-world";
    }
```

Templater une vue

Avec des données quelleconques

Pour templater une vue, il faut ajouter l'objet `Model` en paramètre de la méthode endpoint, et utiliser la méthode `addAttribute` pour ajouter des objets au modèle :

```
    @GetMapping
    public String HelloWorld(Model model) {
        model.addAttribute("today", LocalDate.now());

        return "hello-world";
    }
```

On peut ensuite utiliser la syntaxe suivante pour accèder au modèle dans la vue :

```
<h1>The date is : <span th:text="${today}"></span></h1>
```

Le moteur de template va appeler la méthode `toString` de l'objet `LocalDate` pour l'afficher.

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `./mvnw.cmd spring-boot:run`). Et en allant à https://localhost:8080/hello-world.

Avec un objet de modèle

Avec un objet de modèle, la syntaxe est la même, on peut accèder aux attributs avec un point. Ce qui va appeler les getters & setters. Par exempe avec la classe suivante :

```

public class Person {

    private String firstName;
    private String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
```

```
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

On peut l'ajouter au modèle de la même façon :

```
    @GetMapping
    public String HelloWorld(Model model) {
        Person p = new Person();
        p.setFirstName("John");
        p.setLastName("Shepard");

        model.addAttribute("today", LocalDate.now());
        model.addAttribute("person", p);

        return "hello-world";
    }
```

Et y accèder dans la vue :

```
<h1>The date is : <span th:text="${today}"></span></h1>

Hello, <span th:text="${person.firstName}"></span> <span th:text="${person.lastName}"></span>
```

Rendu Conditionnel

Pour faire du rendu conditionnel, il faut utilise l'attribut `th:if` :

```
<span th:if="${(20 % 2) == 0}">20 is even</span>
<span th:if="${(21 % 2) == 0}">21 is even</span>
```

Seulement "20 is even" sera rendu sur la page.

Boucles

Pour faire du rendu en boucle, il faut utiliser l'attribut `th:each`, exemple :

```
<div th:each="person: ${persons}">
<span th:text="${person.firstName}"></span> <span th:text="${person.lastName}"></span>
</div>
```

Soumission de formulaire

Pour soumettre un formulaire avec Thymleaf, il faut d'abord sur le formulaire :

- L'attribut `th:action` qui va indiquer l'URL de l'endpoint qui gère la soumission du formulaire
- L'attribut `th:object` qui va indiquer le nom de l'objet du modèle que l'on va modifier dans le formulaire
- L'attribut `method` qui indique le verbe HTTP utiliser pour la soumission du formulaire, il faut utiliser POST. La méthode endpoint qui gère le formulaire doit aussi gérer le verbe HTTP POST et va donc utiliser l'annotation `PostMapping`

Sur les champs du formulaire, il faut l'attribut `th:field` afin d'indiquer à quel attribut de la classe modèle le champs correspond.

Dans la méthode endpoint qui sert le formulaire, il faut ajouter dans le modèle l'objet que l'on veut modifier avec le formulaire (son nom doit correspondre avec l'attribut `th:object` du formulaire).

Exemple :

Les deux templates, pour le formulaire et le résultat :

`hello-world-form.html` :

```
<form th:action="@{/hello-world/salute}"  th:object="${person}" method="post">
    <input th:field="*{firstName}" type="text"/>
    <input th:field="*{lastName}" type="text"/>
    <input type="submit" value="Say hello !"/>
</form>
```

`hello-world.html` :

```
Hello, <span th:text="${person.firstName}"></span> <span th:text="${person.lastName}"></span>
```

Les méthodes endpoint :

```
    @PostMapping("salute")
    public String SayHello(@ModelAttribute Person person) {
```

```
        return "hello-world";
    }

    @GetMapping
    public String HelloWorld(Model model) {
        model.addAttribute("person", new Person());
        return "hello-world-form";
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `./mvnw.cmd spring-boot:run`). Et en allant à https://localhost:8080/hello-world.

Session HTTP

On peut stocker des données dans la session HTTP. Pour cela, ajoutez simplement une `HttpSession` en paramètre de votre méthode endpoint :

```
@GetMapping
    public String EndpointWithSession(HttpSession session){
      ...
    }
```

Pour enregistrer des données dans la session :

```
session.setAttribute("clé","valeur");
```

Pour récupérer des données de la session :

```
String value = session.getAttribute("clé");
```

Interaction avec la base de donnée

L'interaction avec la base de donnée se fait avec un ORM. Ici on va utiliser une implémentation de JPA (Java Persistence API). JPA est la spécification de JarkataEE pour les ORMs. Nous allons utiliser Hibernate, son implémentation la plus connue.

Installer Hibernate

Il faut installer le package nugger d'hibernate, ainsi que le driver de base de donnée en dépendance Maven. Pour ce faire, ouvrez votre `pom.xml` et ajoutez les balises suivantes à la balise `<dependencies>` :

Hibernate :

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

Driver JDBC MySQL :

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.22</version>
</dependency>
```

Configurer la connexion

Afin de configurer la connexion à la base de donnée, ouvrez le fichier `application.properties` situé dans `src/main/resources`. Et ajoutez les entrées suivantes :

```
spring.datasource.url=${DATABASE_URL}
spring.datasource.username=${DATABASE_USERNAME}
spring.datasource.password=${DATABASE_PASSWORD}

spring.jpa.generate-ddl=false
spring.jpa.open-in-view=false
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.temp.use_jdbc_metadata_defaults=false
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL55Dialect
```

Dans le fichier `application.properties`, la syntaxe `${}` permet de récupérer la valeur d'une variable d'environnement.

> L'URL JDBC de la base de donnée MySQL se consruit comme cecii :
> `jdbc:mysql://<hote>:3306/<nomDeLaBase>`

Les Entités

Les entités sont les objets qui sont persistés par l'ORM dans la base de donnée. Pour définir nos classes d'objets qui seront enregistrées dans la base de donnée, il faut les définir en tant qu'entité. Cela se fait à l'aide de l'annotation `@Entity` sur la classe. Une entité doit avoir un champs qui correspondra à se clé primaire dans la base de donnée, il est désigné avec l'annotation `@Id`. l'ORM peut le générer automatiquement, et même aléatoirement dans le cas d'une UUID (voir exemple).

Les autres champs de la classe entités sont persistés automatiquement. Pour qu'un champs ne soit pas persisté, il faut l'annoter avec `@Transient`.

Exemple d'entité pour notre Todo :

```
@Entity
public class Person {
  @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;

  private String firstName;
    private String lastName;

  ... getters & setters ...
}
```

Les annotations `@GeneratedValue` et `GenericGenerator` permet de générer automatiquement un UUID aléatoire pour l'entité.

On peut aussi définir des contraintes sur les colonnes en rajoutant l'annotation `@Column` sur le champs et en lui passant des paramètres, par exemple :

```
@Column(unique = true, length = 32)
```

Cela permet de mettre la contrainte "unique" sur le champs et d'imposer une longueur maximum de 32 caractère

Les Repositories

Pour interagir avec les entités, il faut créer des Repository. Ce sont des interface que nous allons définir, mais qui seront implémentées non pas nous, mais pas l'ORM Hibernate. Ils peuvent ensuite être injectés dans les services par le conteneur d'injection de dépendances de Spring. Une interface Repository gère les interactions pour une entité et doit étendre l'interface `JpaRepository` en fournissant en paramètre de type, le type de l'entité, ainsi que le type de son Id.

Exemple :

```
public interface PersonsRepository extends JpaRepository<Todo, String> {


}
```

Voilà, juste en étendant cette interface, on peut accèder à tout un tas de méthodes intéressantes. Les principales sont :

- `save` : Sauvegarder (créer ou mettre à jours) une instance d'une entités
- `findById` : récupérer une entité à partir de son Id
- `findAll` : récupérer toutes les entités de ce type contenues dans la base (⚠ attention ça peut faire beaucoup de tout charger dans la mémoire)
- `deletebyId` : supprimer une entité à partir de son Id

Et bien d'autre qui peuvent être utile, à retrouver dans la javadoc de l'interface

Utiliser le repository dans le contrôleur

Pour interagir avec la base de donnée depuis notre controleur, on peut demander au framework de nous l'injecter :

```
@Controller
@RequestMapping("persons")
public class PersonsController {


    private final PersonsRepository repository;


    public PersonsController(PersonsRepository repository) {
        this.repository = repository;
    }
}
```

Créons maintenant un endpoint servant une vue avec :

- Une liste des personnes dans la base
- Un formulaire pour ajouter une personne

```
<div th:each="person: ${persons}">
    <span th:text="${person.firstName}"></span> <span th:text="${person.lastName}"></span>
</div>

<form th:action="@{/persons}" th:object="${person}" method="post">
    <input th:field="*{firstName}" type="text"/>
    <input th:field="*{lastName}" type="text"/>
    <input type="submit" value="Save"/>
</form>
```

D'abord, pour récupérer les personnes, nous allons appeler la méthode `findAll` de notre repository, pour l'ORM, cela revient à faire une requête pour récuprer tous les enregistrement de la table :

```
    @GetMapping
    public String persons(Model model){
        List<Person> persons = repository.findAll();

        model.addAttribute("persons", persons);
        model.addAttribute("person", new Person());

        return "persons";
    }
```

Ensuite, on peut ajouter notre endpoint d'insertion, qui redirige ensuite vers l'endpoint d'affichage :

```
    @PostMapping
    public RedirectView createPerson(@ModelAttribute Person person){
        this.repository.save(person);

        return new RedirectView("persons");
    }
```

On ajoute la `Person` récupérée depuis le formulaire au repository avec la méthode `save`.

> Vous pouvez regarder dans la console les requêtes effectuées par Hibernate sur la base de données, qui correspondent à nos appels

On va ensuite ajouter une vue pour voir une seule personne, avec un lien dans la liste. Créons d'abord notre endpoint :

```
@GetMapping("{id}")
public String personDetails(@PathVariable("id") String id, Model model){
    Person person = repository.getById(id);

    model.addAttribute("person", person);

    return "person";
}
```

On utilise un paramètre de route pour récupérer l'identifiant, ainsi que la méthode `getById` du repository pour récupérer l'enregistrement.

On retourne ensuite la vue suivante :

```
<span th:text="${person.firstName}"></span> <span th:text="${person.lastName}"></span>
```

Enfin on modifie la vue d'affichage pour ajouter un lien :

```
<div th:each="person: ${persons}">
    <a th:href="'/persons/' + ${person.id}"><span  th:text="${person.firstName}"></span> <span
th:text="${person.lastName}"></span></a>
</div>
```

Relations

Les relations permettent de faire des liens entre les entités. Il existe quatres types de relations, définies par des annotations :

- `@ManyToOne` : plusieurs instances de cette classe entité sont en relation avec une unique instance d'une autre (ex: Pages d'un livre - plusieurs pages sont reliées à un unique livre)
- `OneToMany` : une unique instance de cette classe entité est en relations avec plusieurs instances d'une autre (ex: Livre qui contient des pages - un unique livre est relié à plusieurs pages)
- `@ManyToMany` : plusieurs instance de cette classe entité sont en relations avec plusieurs isntance d'une autre (ex: Classes et Professeurs - Les professeurs ont plusieurs classes et

les classes ont plusieurs professeurs.
- `@OneToOne` : une unique instance de cette classe entités est reliée à une unique instance d'une autre (ex: Dircteur et Ecole - une directeur dirige une seule école et une école est dirrigée par un seul directeur).

Exemple :

```
@Entity
public class Todo {
  ... autres propriétés ...

  @ManyToOne
  private ApplicationUser owner;

  ... getters & setters ....
}
```

```
@Entity
public class ApplicationUser {
    @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;

  private String name;

  @OneToMany
    private Set<Todo> todos;

    ... getters & setters ...
}
```

Les annotations de relation possèdes plusieurs paramètres utiles à connaitres :

- `fetch` :
  - `FetchType.EAGER` : charge les données de la relation directement (défaut pour `@ManyToOne` et `@OneToOne`)
  - `FtechType.LAZY` : charge les données de la relation que quand le getter est appelé (défaut pour `@OneToMany` et `@ManyToMany`
- `orphanRemoval` : `true` ou `false`, détermine dans un `@OneToMany` si les enfant doivent être supprimés quand le parent est supprimé → un enfant ne peut exister sans parent.

Dockeriser une application Spring Boot

D'abord on build un JAR exécutable de notre application :

`./mvnw clean package`

Cela va vous construire un JAR exécutable dans `target/build`.

On crée ensuite le Dockerfile suivante à la racine du projet :

```
FROM openjdk:17.0-oracle
EXPOSE 8080
ADD ./target/mon-projet-0.0.1-SNAPSHOT.jar app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

On peut ensuite build l'image :

```
docker build -t mon-image -f Dockerfile .
```

Et enfin lancer l'image :

> Attention, si votre base de donnée est aussi un conteneur Docker tournant sur votre machine, dans l'hote de la base de donnée, notez non pas "localhost" mais l'adresse locale de votre machine pour permettre à la connexion de loopback entre vos conteneurs.

```
docker run -p 80:8080 -e DATABASE_URL=<URL JDBC de votre base de donnée> -e DATABASE_USERNAME=root -e DATABASE_PASSWORD=monMdpTresSecret -t mon-image
```

Sources

- https://spring.io/guides/gs/serving-web-content/
- https://www.baeldung.com
- https://www.logicbig.com/tutorials/spring-framework/spring-web-mvc/http-session-param.html

Revision #12
Created 24 November 2021 16:00:08 by Arsène Lapostolet
Updated 27 November 2021 22:44:08 by Arsène Lapostolet