Spring Boot : les fondamentaux

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com

Plan

1. Introduction
2. Un premier projet Spring Boot
3. Le contrôleur
4. DevTools
5. La vue
   - Model, ModelMap et ModelAndView
   - Paramètres de requête et variables de chemin
6. Le modèle
7. Thymeleaf
8. L'internationalisation (i18n)
9. Les services web Rest

Spring Boot : Introduction

Spring MVC

- un des premiers framework Spring
- basé sur l'API Servlet de Java JEE
- permettant de simplifier le développement d'applications web en respectant le patron de conception MVC 2

Problèmes

- trop de dépendance à gérer (ce qui pose souvent un problème d'incompatibilité entre les versions)
- beaucoup de configuration (JPA, Sécurité, contrôleurs, vues...)

Spring Boot : Introduction

Spring Boot : encore de l'abstraction

Pour éviter les problèmes de **Spring MVC**, **Spring Boot** propose :

- Les démarreurs (`starter`) : un démarreur est une dépendance, contenant un paquet de dépendance, permettant de réaliser un type de projet (Web, Rest...). Ainsi, le développeur n'a plus à gérer, lui même le problème d'incompatibilité entre les versions.
- l'auto-configuration : c'est-à-dire laisser **Spring Boot** configurer le projet à partir de dépendances ajoutées par le développeur.

Spring Boot : Introduction

Exemple, pour créer un projet web, il faut ajouter la dépendance *Spring Boot* suivante :

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

Spring Boot : Introduction

Exemple, pour créer un projet web, il faut ajouter la dépendance *Spring Boot* suivante :

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

Les démarreurs **Spring Boot** ont la forme

`spring-boot-starter-*`

Spring Boot : Introduction

Exemple, pour créer un projet web, il faut ajouter la dépendance *Spring Boot* suivante :

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

Les démarreurs **Spring Boot** ont la forme

`spring-boot-starter-*`

Pour consulter la liste des starters, aller sur

https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-build-systems.html

La dépendance `spring-boot-starter-web` inclut les six dépendances suivantes :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-json</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
</dependency>
<dependency>
  <groupId>org.hibernate.validator</groupId>
  <artifactId>hibernate-validator</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
</dependency>
```

Spring Boot : Introduction

La dépendance `spring-boot-starter-web` permet donc de créer un projet web contenant :

- un serveur **Apache Tomcat**
- Spring Framework et Spring MVC
- les validateurs d'Hibernate
- jackson pour les données sous format JSON
- ...

Spring Boot : les fondamentaux

Création de projet Spring Boot

- Aller dans `File > New > Other`
- Chercher `Spring`, dans `Spring Boot` sélectionner `Spring Starter Project` et cliquer sur `Next >`
- Saisir
  - `FirstSpringBoot` dans `Name`,
  - `com.spring` dans `Group`,
  - `FirstSpringBoot` dans `Artifact`
  - `com.spring.demo` dans `Package`
- Cliquer sur `Next >`
- Chercher et cocher les cases correspondantes aux `Web` puis cliquer sur `Next >`
- Valider en cliquant sur `Finish`

Spring Boot : les fondamentaux

Pourquoi a-t-on coché la case `Web` à la création du projet ?

- pour ajouter la dépendance `spring-boot-starter-web`

Contenu de la section `dependencies` de `pom.xml`

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

Spring Boot : les fondamentaux

Pour la compatibilité d'*Apache Tomcat* avec les JSP, on ajoute la dépendance suivante

```
<!-- https://mvnrepository.com/artifact/org.apache.
   tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
```

La version d'**Apache Tomcat** doit être la même que celles qui existent dans `Maven Dependencies`

Spring Boot : les fondamentaux

Remarques

- Le package contenant le point d'entrée de notre application (la classe contenant le `public static void main`) est `com.spring.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Spring Boot : les fondamentaux

Le point de démarrage de l'application

```
package com.spring.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class FirstSpringBootApplication {

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

Spring Boot : les fondamentaux

Explication

- `SpringApplication` : la classe de démarrage d'une application Spring et qui va créer une instance de la classe `ApplicationContext`
- `ApplicationContext` : l'interface centrale d'une application Spring permettant de fournir des informations de configuration à l'application.
- `@SpringBootApplication` : contient les 3 annotations suivantes
  - `@Configuration` : fait partie du noyau de Spring Framework et indique que la classe annoté peut contenir des méthodes annotées par `@Bean`. Ainsi,`Spring Container` peut traiter la classe et générer des beans qui seront utilisés par l'application.
  - `@EnableAutoConfiguration` : permet, au démarrage de Spring, de générer automatiquement les configurations nécessaires en fonction des dépendances ajoutées.
  - `@ComponentScan` : demande de scanner ce package contenant de Beans de configuration

Spring Boot : les fondamentaux

Pour exécuter

- Faire un clic droit sur `FirstSpringBoot` dans `Package Explorer`
- Aller dans `Run As` et cliquer sur `Spring Boot App`

Spring Boot : les fondamentaux

Pour exécuter

- Faire un clic droit sur `FirstSpringBoot` dans `Package Explorer`
- Aller dans `Run As` et cliquer sur `Spring Boot App`

La console nous indique

```
Tomcat started on port(s): 8080 (http) with context path ''
```

Spring Boot : les fondamentaux

Pour exécuter

- Faire un clic droit sur `FirstSpringBoot` dans `Package Explorer`
- Aller dans `Run As` et cliquer sur `Spring Boot App`

La console nous indique

```
Tomcat started on port(s): 8080 (http) with context path ''
```

Allons donc à `http://localhost:8080/`

Spring Boot : les fondamentaux

Résultat : message d'erreur

- On a créé un projet web, mais on n'a aucune page HTML, JSP ou autre
- Spring Boot, comme Spring MVC, implémente le patron de conception MVC, donc il nous faut au moins un contrôleur et une vue.

Le contrôleur

Le contrôleur

- un des composants du modèle MVC
- une classe Java annotée par `Controller` ou `RestController`
- Il reçoit une requête du contrôleur frontal et communique avec le modèle pour préparer et retourner une réponse à la vue

Le contrôleur

Création du contrôleur

- Faire clic droit sur le projet
- Aller dans `new > class`
- Choisir le package `com.spring.demo.controller`
- Saisir `HomeController` comme nom de classe
- Ensuite valider

Le contrôleur

Remplaçons le contenu du `HomeController` par le code suivant :

```
package com.spring.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class HomeController {

  @RequestMapping(value="/hello", method = RequestMethod.GET)
  public void sayHello() {

    System.out.println("Hello␣World!");

  }
}
```

Le contrôleur

Explication

- La première ligne indique que notre contrôleur se trouve dans le package `com.spring.demo.controller`
- Les trois imports concernent l'utilisation des annotations
- L'annotation `@Controller` permet de déclarer que la classe suivante est un contrôleur Spring
- La valeur de l'annotation `@RequestMapping` indique la route (`/hello` ici) et la méthode permet d'indiquer la méthode HTTP (`get` ici, c'est la méthode par défaut). On peut aussi utiliser le raccourci `@GetMapping(value="/url")`

Le contrôleur

Remplaçons le contenu du `HomeController` par le code suivant :

```
package com.spring.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {

  @GetMapping(value="/hello")
  public void sayHello() {

    System.out.println("Hello World!");

  }
}
```

Le contrôleur

Testons tout cela

- Démarrer le serveur Apache Tomcat
- Aller sur l'url `http://localhost:8080/hello` et vérifier qu'un `Hello World!` a bien été affiché dans la console (d'Eclipse)

Le contrôleur

Remarque

- On peut aussi annoter le contrôleur par le `@RequestMapping`

```
@Controller
@RequestMapping("/hello")
public class HelloController {
   ...
}
```

DevTools

DevTools

- outil de développement
- fonctionnant en mode développement
- permettant de redémarrer le projet après chaque changement

DevTools

DevTools

- outil de développement
- fonctionnant en mode développement
- permettant de redémarrer le projet après chaque changement

Dépendance Maven à ajouter

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

Devtools

Avant utilisation, il faut

- vider le cache du projet
- le mettre à jour

Les vues

Constats

- Dans une application web Spring MVC, le rôle du contrôleur n'est pas d'afficher des informations dans la console
- C'est plutôt de communiquer avec les différents composants
- Afficher la réponse est le rôle de la vue

Les vues

Les vues sous Spring

- Permettent d'afficher des données
- Communiquent avec le contrôleur pour récupérer ces données
- Doivent être créées dans le répertoire `views` dans `WEB-INF`
- Peuvent être créées avec un simple code `JSP`, `JSTL` ou en utilisant un moteur de template comme `Thymeleaf`...

Les vues

Par défaut

- **Spring** cherche les vues dans un répertoire `webapp` situé dans `src/main`.
- Le répertoire n'existe pas, il faut le créer.

Les vues

Par défaut

- **Spring** cherche les vues dans un répertoire `webapp` situé dans `src/main`.
- Le répertoire n'existe pas, il faut le créer.

Créons une première vue que nous appelons `hello.jsp` dans `webapp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>first jsp called from controller</title>
  </head>
  <body>
    <h1>first jsp called from controller</h1>
   </body>
</html>
```

Les vues

Appelons `hello.jsp` à partir du contrôleur

```
package com.spring.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping
   ;

@Controller
public class HomeController {
        @GetMapping(value="/hello")
        public String sayHello() {
                return "hello.jsp";
        }
}
```

Dans le `return`, on précise le nom de la vue à afficher (ici c'est `hello.jsp`)

Les vues

Remarques

- On peut préciser un autre répertoire pour les vues (il faut qu'il soit dans `webapp`)
- Pour éviter de préciser chaque fois l'extension de la vue, on peut l'indiquer dans `application.properties` situé dans `src/main/resources`

Nouveau contenu d'`application.properties`

```
spring.mvc.view.prefix=/views/
spring.mvc.view.suffix=.jsp
```

Toutes les propriétés possibles de `application.properties` sont ici : https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

Les vues

Nouveau contenu du contrôleur

```
package com.spring.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping
   ;

@Controller
public class HomeController {
        @GetMapping(value="/hello")
        public String sayHello() {
                return "hello";
        }
}
```

N'oublions pas de déplacer `hello.jsp` dans `views` qu'il faut le créer dans `webapp`

Les vues

Deux questions

- Comment passer des données d'une vue à un contrôleur et d'un contrôleur à une vue ?
- Une vue peut-elle appeler un contrôleur ?

Les vues

Comment le contrôleur envoie des données à la vue ?

```
package com.spring.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping
   ;
@Controller
public class HomeController {
  @GetMapping(value="/hello")
  public String sayHello(Model model) {
    model.addAttribute("nom", "Wick");
    return "hello";
  }
}
```

Dans la déclaration de la méthode, on injecte l'interface `Model` qui nous permettra d'envoyer des attributs à la vue

Les vues

Comment la vue récupère les données envoyées par le contrôleur ?

```
<%@ page language="java" contentType="text/html; charset=
   UTF-8"  pageEncoding="UTF-8"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
       charset=UTF-8">
    <title>first jsp called from controller</title>
  </head>
  <body>
    <h1>first jsp called from controller</h1>
    Je m'appelle ${ nom }
  </body>
</html>
```

Exactement comme dans la plateforme JEE

Les vues

Comment la vue récupère les données envoyées par le contrôleur ?

```
<%@ page language="java" contentType="text/html; charset=
   UTF-8"  pageEncoding="UTF-8"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
       charset=UTF-8">
    <title>first jsp called from controller</title>
  </head>
  <body>
    <h1>first jsp called from controller</h1>
    Je m'appelle ${ nom }
  </body>
</html>
```

Exactement comme dans la plateforme JEE

Ajouter `isELIgnored="false"` s'il ne reconnait pas les Expressions de langage

Les vues

Une deuxième façon en utilisant `ModelAndView`

```
package com.spring.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.ui.ModelMap;

@Controller
public class HomeController {

  @RequestMapping(value="/hello")
  public String sayHello(ModelMap model) {

    model.addAttribute("nom", "Wick");
    return "hello";
  }
}
```

Les vues

Une troisième façon en utilisant `ModelAndView`

```
package com.spring.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class HomeController {

  @RequestMapping(value="/hello")
  public ModelAndView sayHello(ModelAndView mv) {

    mv.setViewName("hello");
    mv.addObject("nom", "wick");
    return mv;
  }
}
```

Les vues

`Model` vs `ModelMap` vs `ModelAndView`

- `Model` : est une interface permettant d'ajouter des attributs et de les passer à la vue
- `ModelMap` : est une classe implémentant l'interface `Map` et permettant d'ajouter des attributs sous forme de `key - value` et de les passer à la vue. On peut donc chercher un élément selon la valeur de la clé ou de la valeur
- `ModelAndView` : est un conteneur à la fois d'un `ModelMap` pour les attributs et d'un `View Object`. Le contrôleur pourra ainsi retourner une seule valeur.

Les paramètres de requête et les variables de chemin

Les paramètres de requête

- Ce sont les paramètres qui s'écrivent sous la forme `/chemin?param1=value1&param2=value2`

Les paramètres de requête et les variables de chemin

Les paramètres de requête

- Ce sont les paramètres qui s'écrivent sous la forme `/chemin?param1=value1&param2=value2`

Les variables de chemin

- Ce sont les paramètres qui s'écrivent sous la forme `/chemin/value`

Les paramètres de requête et les variables de chemin

Comment le contrôleur récupère les paramètres de requête ?

```
package com.spring.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class HomeController {

  @GetMapping(value="/hello")
  public String sayHello(@RequestParam(value = "nom") String
     nom, Model model) {
    model.addAttribute("nom", nom);
    return "hello";
  }
}
```

Les paramètres de requête et les variables de chemin

Pour tester, il faut aller sur l'URL

`localhost:8080/hello?nom=wick`

Explication

- `@RequestParam(value = "nom") String nom` : permet de récupérer la valeur du paramètre de la requête HTTP est de l'affecter au paramètre `nom` de la méthode.

Les paramètres de requête et les variables de chemin

Peut-on accéder à `localhost:8080/hello` sans préciser le paramètre `nom` ?

- non, une erreur sera affichée.

Mais, il est possible de rendre ce paramètre facultatif

```
@Controller
public class HomeController {

  @GetMapping(value="/hello")
  public String sayHello(@RequestParam(value="nom",
     required=false) String nom, Model model) {

    model.addAttribute("nom", nom);
    return "hello";
  }
}
```

Les paramètres de requête et les variables de chemin

Il est possible aussi de préciser une valeur par défaut

```
@Controller
public class HomeController {

  @GetMapping(value="/hello")
  public String sayHello(@RequestParam(value="nom",
     required=false, defaultValue="wick") String nom,
     Model model) {

    model.addAttribute("nom", nom);
    return "hello";
  }
}
```

Les paramètres de requête et les variables de chemin

Comment le contrôleur récupère une variable de chemin ?

```
package com.spring.demo.controller;

import org.springframework.web.bind.annotation.PathVariable;

@Controller
public class HomeController {

  @GetMapping(value = "/hello/{nom}")
  public void sayHello(@PathVariable String nom, Model model) {

    model.addAttribute("nom", nom);
    return "hello";
  }
}
```

Pour tester, il faut aller sur l'URL `localhost:8080/hello/wick`

Les vues

Comment une vue peut faire appel à une méthode d'un contrôleur ?

- Soit en utilisant les formulaires et on précise la route dans l'attribut `action` et la méthode dans l'attribut `method`
- Soit en utilisant un lien hypertexte (dans ce cas la méthode est `get`)
- ...

Le modèle

Modèle : accès et traitement de données

- Utilisation de JPA et Hibernate
- Précision de données de connexion dans `application.properties`
- Utilisation des annotations (`@Repository`, `@Service`... et `@Autowired` pour l'injection de dépendance)

Organisation du projet

- Créons un premier répertoire `com.spring.demo.model` dans `src/main/java` où nous placerons les entités JPA
- Créons un deuxième répertoire `com.spring.demo.dao` dans `src/main/java` où nous placerons les classes DAO (ou ce qu'on appelle `Repository` dans Spring)

Le modèle

On commence par ajouter les dépendances suivantes

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

Le modèle

On commence par ajouter les dépendances suivantes

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

Et si on veut intégrer le modèle dès la création du projet

- Dans ce cas, il faut cocher les cases respectives de `MySQL` et `JPA` et `Web`

Le modèle

Dans `application.properties`, on ajoute les données concernant la connexion à la base de données et la configuration de `Hibernate`

```
spring.datasource.url = jdbc:mysql://localhost:3306/myBase?useUnicode=
  true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&
  serverTimezone=UTC
spring.datasource.username =root
spring.datasource.password =root
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql = true
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.
  MySQL5Dialect
```

Cette partie (`?useUnicode=true&useJDBCCompliantTimezoneShift=true &useLegacyDatetimeCode=false&serverTimezone=UTC`) dans la chaîne de connexion est ajoutée pour éviter un bug du connecteur `MySQL` concernant l'heure système.

L'ajout de la propriété `spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl` permet de forcer **Hibernate** à utiliser les mêmes noms pour les tables et les colonnes que les entités et les attributs.

L'entité `Personne`

```
package com.spring.demo.model;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.
   GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "personnes")
public class Personne implements
   Serializable {
  @Id @GeneratedValue
  private Long num;
  private String nom;
  private String prenom;
  private static final long
     serialVersionUID = 1L;
  public Personne() {    }
  public Personne(String nom,
     String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }
  public Long getNum() {
    return num;
  }
  public void setNum(Long num) {
    this.num = num;
  }
  public String getNom() {
    return nom;
  }
  public void setNom(String nom) {
    this.nom = nom;
  }
  public String getPrenom() {
    return prenom;
  }
  public void setPrenom(String
     prenom) {
    this.prenom = prenom;
  }
}
```

Le modèle

Pour obtenir le DAO, il faut créer une interface qui étend

- soit `CrudRepository` : fournit les méthodes principales pour faire le CRUD
- soit `PagingAndSortingRepository` : hérite de `CrudRepository` et fournit en plus des méthodes de pagination et de tri sur les enregistrements
- soit `JpaRepository` : hérite de `PagingAndSortingRepository` en plus de certaines autres méthodes JPA.

Le modèle

Le repository

```
package com.spring.demo.dao;

import java.util.List;

import org.springframework.data.jpa.repository.
   JpaRepository;

import com.spring.demo.model.Personne;

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

`Long` est le type de la clé primaire (`Id`) de la table (entité) personnes

Le modèle

Préparons le formulaire dans `addPerson.jsp`

```
<%@ page language="java" contentType="text/html;␣
   charset=UTF-8"  pageEncoding="UTF-8"%>
<html>
  <head>
    <title>Index page</title>
  </head>
  <body>
    <h2>To add new person</h2>
    <form action="add" method="post">
      Nom :  <input  type="text" name="nom">
      Prenom :  <input  type="text" name="prenom">
      <button type="submit">Envoyer</button>
    </form>
  </body>
</html>
```

Le modèle

Préparons le contrôleur

```
@Controller
public class PersonneController {
  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping(value="/addPerson")
  public  String addPerson() {
    return "addPerson";
  }

  @PostMapping(value="/addPerson")
  public  ModelAndView addPerson(@RequestParam(value = "nom") String
     nom, @RequestParam(value = "prenom") String prenom) {
    Personne p1 = new Personne(nom,prenom);
    personneRepository.save(p1);
    ModelAndView mv = new ModelAndView();
    mv.setViewName("confirm");
    mv.addObject("nom", nom);
    mv.addObject("prenom", prenom);
    return mv;
  }
}
```

Le modèle

Préparons la vue `confirm.jsp`

```
<%@ page language="java" contentType="text/html;␣
   charset=UTF-8"  pageEncoding="UTF-8"%>
<html>
  <head>
    <title>Confirm page</title>
  </head>
  <body>
    <h1>Welcome</h1>
    Person named ${ nom } ${ prenom } has been
       successfully added in our database.
  </body>
</html>
```

Le modèle

Et si on veut récupérer la liste de toutes les personnes ?

```
@RequestMapping(value="/show")
public  ModelAndView showAll() {
  ArrayList <Personne> al =(ArrayList<Personne>)
     personneRepository.findAll();
  ModelAndView mv = new ModelAndView();
  mv.setViewName("result");
  mv.addObject("tab", al);
  return mv;
}
```

Le modèle

Et la vue `result.jsp` **:**

```
<%
  ArrayList <Personne> al = (ArrayList <Personne>)
     request.getAttribute("tab");
  for(Personne p: al){
    out.print("Hello␣" + p.getNom() + "␣" + p.
       getPrenom());
  }
%>
```

Le modèle

Autres méthodes du repository

- `findById()` : recherche selon la valeur de la clé primaire
- `findAllById()` : recherche selon un tableau de clé primaire
- `deleteById()` : Supprimer selon la valeur de la clé primaire
- `deleteAll()` : supprimer tout
- `flush()` : modifier
- `count()`, `exists()`, `existsById()`...

Le modèle

On peut aussi récupérer la liste de personnes par page

```
@RequestMapping(value="/showAllByPage/{i}/{j}")
public  ModelAndView showAllByPage(@PathVariable int i,
   @PathVariable int j) {
  Page<Personne> personnes = personneRepository.findAll(
     PageRequest.of(i, j));
  ModelAndView mv = new ModelAndView();
  mv.setViewName("result");
  mv.addObject("tab", personnes.getContent());
  return mv;
}
```

Les variables de chemin `i` et `j`

- `i` : le numéro de la page (première page d'indice 0)
- `j` : le nombre de personnes par page

Le modèle

On peut aussi récupérer une liste de personnes triée

```
@RequestMapping(value="/showAllSorted")
public  ModelAndView showAllSorted() {
  List<Personne> personnes = personneRepository.findAll(
     Sort.by("nom").descending());
  ModelAndView mv = new ModelAndView();
  mv.setViewName("result");
  mv.addObject("tab", personnes);
  return mv;
}
```

Explication

- Ici on trie le résultat selon la colonne `nom` dans l'ordre décroissant

Le modèle

Les méthodes personnalisées du repository

- On peut aussi définir nos propres méthodes personnalisées dans le repository et sans les implémenter.

Le modèle

Le repository

```
package org.eclipse.FirstSpringMvc.dao;

import java.util.List;
import org.springframework.data.jpa.repository.
   JpaRepository;

import org.eclipse.FirstSpringMvc.model.Personne;

public interface PersonneRepository extends
   JpaRepository <Personne, Long> {
  List<Personne> findByNomAndPrenom(String nom,
     String prenom);
}
```

`nom` et `prenom` : des attributs qui doivent exister dans l'entité `Personne`.

Il faut respecter le `CamelCase`

Le modèle

Le contrôleur

```
@RequestMapping(value="/showSome")
public  ModelAndView showSome(@RequestParam(value =
   "nom") String nom, @RequestParam(value = "prenom"
   ) String prenom) {
  ArrayList <Personne> al =(ArrayList<Personne>)
     personneRepository.findByNomAndPrenom(nom,
     prenom);
  ModelAndView mv = new ModelAndView();
  mv.setViewName("result");
  mv.addObject("tab", al);
  return mv;
}
```

Le modèle

Dans la méthode précédente on a utilisé l'opérateur logique `And`

Mais, on peut aussi utiliser

- `Or`, `Between`, `Like`, `IsNull`...
- `StartingWith`, `EndingWith`, `Containing`, `IgnoreCase`
- `After`, `Before` pour les dates
- `OrderBy`, `Not`, `In`, `NotIn`
- ...

Le modèle

On peut également écrire des requêtes HQL (Hiberenate Query Language) avec l'annotation `Query`

```
package com.spring.demo.dao;

import java.util.List;
import org.springframework.data.jpa.repository.
   JpaRepository;

import com.spring.demo.model.Personne;

public interface PersonneRepository extends
   JpaRepository <Personne, Long> {
  @Query("select p from Personne p where p.nom = ?1"
     )
  Personne findByNom(String nom);
}
```

Thymeleaf

Dépendance Maven à ajouter

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

Thymeleaf

Gestion de vues

- Créer deux répertoires : `jsp` et `thymeleaf` dans le répertoire `views` de `webapp`
- Déplacer et placer toutes les pages `JSP` dans `jsp`
- Placer les vues `Thymeleaf` dans `thymeleaf`

Thymeleaf

Gestion de vues

- Créer deux répertoires : `jsp` et `thymeleaf` dans le répertoire `views` de `webapp`
- Déplacer et placer toutes les pages `JSP` dans `jsp`
- Placer les vues `Thymeleaf` dans `thymeleaf`

Configurons `application.properties`

```
spring.view.view-names=jsp/*
spring.thymeleaf.prefix=/views/
spring.thymeleaf.suffix=.html
spring.thymeleaf.view-names=thymeleaf/*
```

JSP et Thymeleaf

Dans les contrôleurs, remplacer chaque appel d'une page JSP

```
return "nomVue";
```

JSP et Thymeleaf

Dans les contrôleurs, remplacer chaque appel d'une page JSP

```
return "nomVue";
```

Par

```
return "jsp/nomVue";
```

JSP et Thymeleaf

Dans les contrôleurs, remplacer chaque appel d'une page JSP

```
return "nomVue";
```

Par

```
return "jsp/nomVue";
```

Pour appeler une page Thymeleaf

```
return "thymeleaf/nomVue";
```

Thymeleaf

Pour tester, créer un contrôleur `ThymeleafController`

```
@Controller
public class ThymeleafController {

  @GetMapping(value="/thymeleaf")
  public String displayMessage(Model model) {

    model.addAttribute("message", "Hello World!");
    return "thymeleaf/index";
  }
}
```

Thymeleaf

La vue `index.html`

```
<!DOCTYPE html>
<html xmlns:th="www.thymeleaf.org">
  <head>
    <meta charset="ISO-8859-1">
    <title>First Thymeleaf Page</title>
  </head>
  <body>
    <p th:text = "${ message }"></p>
  </body>
</html>
```

Thymeleaf

La vue `index.html`

```
<!DOCTYPE html>
<html xmlns:th="www.thymeleaf.org">
  <head>
    <meta charset="ISO-8859-1">
    <title>First Thymeleaf Page</title>
  </head>
  <body>
    <p th:text = "${ message }"></p>
  </body>
</html>
```

En allant , `Hello World!` **est affiché**

L'internationalisation (i18n)

Préciser les sources et l'encodage de messages dans `application.properties`

```
spring.messages.encoding=UTF-8
spring.messages.basename=messages
```

L'internationalisation (i18n)

Préciser les sources et l'encodage de messages dans `application.properties`

```
spring.messages.encoding=UTF-8
spring.messages.basename=messages
```

Contenu de `messages.properties` **(dans** `src/main/resources`**)**

```
welcome.text=Bonjour tout le monde
```

L'internationalisation (i18n)

Préciser les sources et l'encodage de messages dans `application.properties`

```
spring.messages.encoding=UTF-8
spring.messages.basename=messages
```

Contenu de `messages.properties` **(dans** `src/main/resources`**)**

```
welcome.text=Bonjour tout le monde
```

Contenu de `messages_en.properties` **(dans** `src/main/resources`**)**

```
welcome.text=Hello world
```

L'internationalisation (i18n)

Préciser les sources et l'encodage de messages dans `application.properties`

```
spring.messages.encoding=UTF-8
spring.messages.basename=messages
```

Contenu de `messages.properties` **(dans** `src/main/resources`**)**

```
welcome.text=Bonjour tout le monde
```

Contenu de `messages_en.properties` **(dans** `src/main/resources`**)**

```
welcome.text=Hello world
```

Dans une vue (Thymeleaf), ajouter

```
<h1 th:text = "#{ welcome.text }"></h1>
```

Créons la classe de configuration `MvcConfig` dans `com.spring.demo.configuration`

```
@Configuration
public class MvcConfig implements WebMvcConfigurer{
  @Bean
  public LocaleResolver localeResolver() {
    SessionLocaleResolver sessionLocaleResolver = new
       SessionLocaleResolver();
    sessionLocaleResolver.setDefaultLocale(Locale.FRANCE);
    return sessionLocaleResolver;
  }

  @Bean
  public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor localeChangeInterceptor = new
       LocaleChangeInterceptor();
    localeChangeInterceptor.setParamName("language");
    return localeChangeInterceptor;
  }

  @Override
  public void addInterceptors(InterceptorRegistry registry) {
     registry.addInterceptor(localeChangeInterceptor());
  }
}
```

L'internationalisation (i18n)

En allant sur `http://localhost:8080/thymeleaf?language=en` **, le résultat est :**

```
Hello world
```

L'internationalisation (i18n)

En allant sur `http://localhost:8080/thymeleaf?language=en` **, le résultat est :**

```
Hello world
```

En allant sur `http://localhost:8080/thymeleaf?language=fr` **, le résultat est :**

```
Bonjour tout le monde
```

L'internationalisation (i18n)

En allant sur `http://localhost:8080/thymeleaf?language=en` **, le résultat est :**

```
Hello world
```

En allant sur `http://localhost:8080/thymeleaf?language=fr` **, le résultat est :**

```
Bonjour tout le monde
```

En allant sur `http://localhost:8080/thymeleaf?language=it` **, le résultat est toujours le même :**

```
Bonjour tout le monde
```

Spring Boot & REST

Considérons le contrôleur `PersonneRestController`

```
@RestController
public class PersonneRestController {

  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping("/personnes")
  public List<Personne> getPersonnes() {
    return personneRepository.findAll();
  }

  @GetMapping("/personnes/{id}")
  public Personne getPersonne(@PathVariable("id") int id) {
    return personneRepository.findById(id).orElse(null);
  }

  @PostMapping("/personnes")
  public Personne addPersonne(@RequestBody Personne personne) {
    System.out.println(personne);
    return personneRepository.save(personne);
  }
}
```

Spring Boot & REST

Pour tester

- Aller sur `localhost:8080/personnes`
- Ou sur `localhost:8080/personnes/1`

Spring Boot & REST

Pour tester

- Aller sur `localhost:8080/personnes`
- Ou sur `localhost:8080/personnes/1`

Pour ajouter une personne

- utiliser Postman en précisant la méthode `POST` et l'url `localhost:8080/personne`
  - dans `Headers`, préciser la clé `Accept` et la valeur `application/json`
  - dans `Body`, cocher `raw` et sélectionner `JSON(application/json)`