Spring MVC : 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

Spring Boot

Rappel

- **Spring** : framework **Java** composé de plusieurs projets comme
  - `Spring Core`,
  - `Spring Data`,
  - `Spring Web` (incluant le framework **Spring MVC**).
  - ...
- **Spring MVC** : framework permettant de créer des applications web implémentant le modèle **MVC** (**M**odel-**V**iew-**C**ontroller).

Spring Boot

Rappel

- **Spring** : framework **Java** composé de plusieurs projets comme
  - `Spring Core`,
  - `Spring Data`,
  - `Spring Web` (incluant le framework **Spring MVC**).
  - ...
- **Spring MVC** : framework permettant de créer des applications web implémentant le modèle **MVC** (**M**odel-**V**iew-**C**ontroller).

Objectif de ce chapitre

Découvrir comment **Spring MVC** implémente le modèle **MVC** .

Spring MVC 5

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

Spring MVC 5

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

```
package org.eclipse.firstspringmvc.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 = "/", method = RequestMethod.GET)
  public void home() {

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

  }
}
```

Spring MVC 5

Explication

- La première ligne indique que notre contrôleur se trouve dans le package `org.eclipse.firstspringmvc.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 (`/` ici) et la méthode indique le verbe **HTTP** (`GET` ici : méthode par défaut).

Spring MVC 5

Attributs de `@RequestMapping`

- `path` : accepte une chaîne de caractères correspondant à la route
- `value` : alias de `path`
- `name` : permet d'attribuer un nom à la route
- `method` : verbe ou méthode **HTTP**
- `params` : contient le tableau de paramètre accepté par la méthode
- `headers` : spécifie les éléments de l'entête
- `consumes` : indique le format de données accepté par la méthode
- `produces` : indique le format de données retourné par la méthode

Spring MVC 5

Depuis Spring 4, `@RequestMapping(value = "/", method = RequestMethod.GET)` **peut être remplacé par** `@GetMapping(value = "/")`

```
package org.eclipse.firstspringmvc.controller;

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

@Controller
public class HomeController {

  @GetMapping(value = "/")
  public void home() {

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

  }
}
```

Spring MVC 5

Ou aussi sans préciser le nom d'attribut dans l'annotation `@GetMapping`

```
package org.eclipse.firstspringmvc.controller;

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

@Controller
public class HomeController {

  @GetMapping("/")
  public void home() {

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

  }
}
```

Spring MVC 5

Pour tester

- Démarrer le serveur **Apache Tomcat**,
- Aller à l'URL `http://localhost:8080/firstspringmvc/` et vérifier qu'un `Hello World!` s'affiche dans la console d'**Eclipse**.

Spring MVC 5

Remarque

Le contrôleur peut aussi être annoté par `@RequestMapping`

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

Spring MVC 5

La méthode d'un contrôleur peut être associée à plusieurs routes

```
package org.eclipse.firstspringmvc.controller;

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

@Controller
public class HomeController {

        @GetMapping({ "/home", "/" })
        public void home() {
                System.out.println("Hello World!");
        }
}
```

Spring MVC 5

La méthode d'un contrôleur peut être associée à plusieurs routes

```
package org.eclipse.firstspringmvc.controller;

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

@Controller
public class HomeController {

        @GetMapping({ "/home", "/" })
        public void home() {
                System.out.println("Hello World!");
        }
}
```

La méthode `home` est accessible via les deux routes suivantes

- `localhost:8080/firstspringmvc/`
- `localhost:8080/firstspringmvc/hello`

Spring MVC 5

Paramètres de requête : paramètres ayant la forme

```
/chemin?param1=value1&param2=value2
```

Spring MVC 5

Pour récupérer les paramètres de requête, on utilise l'annotation `@RequestParam` (code à ajouter dans `HomeController`)

```
@GetMapping("/hello")
public void sayHello(@RequestParam(value = "nom") String s) {

    System.out.println("Hello " + s);

}
```

Spring MVC 5

Pour récupérer les paramètres de requête, on utilise l'annotation `@RequestParam` (code à ajouter dans `HomeController`)

```
@GetMapping("/hello")
public void sayHello(@RequestParam(value = "nom") String s) {

    System.out.println("Hello " + s);

}
```


URL pour tester

`localhost:8080/firstspringmvc/hello?nom=wick`

Spring MVC 5

Explication

L'annotation `@RequestParam(value = "nom") String s` permet de récupérer la valeur du paramètre de la requête **HTTP** est de l'affecter au paramètre `s` de la méthode `sayHello()`.

Spring MVC 5

Explication

L'annotation `@RequestParam(value = "nom") String s` permet de récupérer la valeur du paramètre de la requête **HTTP** est de l'affecter au paramètre `s` de la méthode `sayHello()`.

Remarque

On peut aussi ajouter `params = {"nom"}` dans `@RequestMapping` pour préciser la liste des paramètres à récupérer de la requête (facultatif)

Spring MVC 5

Question

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

Spring MVC 5

Question

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

Réponse

Non, une erreur sera affichée : `Error 400: Required String parameter 'nom' is not present`

Spring MVC 5

Mais, il est possible de rendre ce paramètre facultatif

```
@GetMapping("/hello")
public void sayHello(@RequestParam(value = "nom", required =
   false) String s) {

    System.out.println("Hello " + s);

}
```

Spring MVC 5

Mais, il est possible de rendre ce paramètre facultatif

```
@GetMapping("/hello")
public void sayHello(@RequestParam(value = "nom", required =
   false) String s) {

    System.out.println("Hello " + s);

}
```


URL pour tester

```
localhost:8080/firstspringmvc/hello?nom=
```

Spring MVC 5

Il est aussi possible de préciser une valeur par défaut

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

    System.out.println("Hello " + s);

}
```

Spring MVC 5

Il est aussi possible de préciser une valeur par défaut

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

    System.out.println("Hello " + s);

}
```


URL pour tester

```
localhost:8080/firstspringmvc/hello?nom=
```

Spring MVC 5

Si le paramètre de la requête HTTP et l'argument de la méthode portent le même nom, l'écriture peut être simplifiée

```
@GetMapping("/hello")
public void sayHello(@RequestParam String nom) {

    System.out.println("Hello " + nom);

}
```

Spring MVC 5

Si le paramètre de la requête HTTP et l'argument de la méthode portent le même nom, l'écriture peut être simplifiée

```
@GetMapping("/hello")
public void sayHello(@RequestParam String nom) {

    System.out.println("Hello " + nom);

}
```


URL pour tester

```
localhost:8080/firstspringmvc/hello?nom=wick
```

Spring MVC 5

Variables de chemin : paramètres ayant la forme

```
/chemin/value1/value2
```

Spring MVC 5

Pour récupérer une variable de chemin, on utilise l'annotation `@PathVariable` (code à ajouter dans `HomeController`)

```
@GetMapping("/hello/{nom}")
public void sayHelloTo(@PathVariable(name="nom") String s) {

    System.out.println("Hello " + s);

}
```

Spring MVC 5

Pour récupérer une variable de chemin, on utilise l'annotation `@PathVariable` (code à ajouter dans `HomeController`)

```
@GetMapping("/hello/{nom}")
public void sayHelloTo(@PathVariable(name="nom") String s) {

    System.out.println("Hello " + s);

}
```


URL pour tester

`localhost:8080/firstspringmvc/hello/wick`

Spring MVC 5

Ou en plus simple

```
@GetMapping("/hello/{nom}")
public void sayHelloTo(@PathVariable String nom) {

    System.out.println("Hello " + nom);

}
```

Spring MVC 5

Ou en plus simple

```
@GetMapping("/hello/{nom}")
public void sayHelloTo(@PathVariable String nom) {

    System.out.println("Hello " + nom);

}
```


URL pour tester

`localhost:8080/firstspringmvc/hello/wick`

Spring MVC 5

Deux solutions pour la redirection

- Utiliser la classe `RedirectView`,
- Ajouter le préfixe `redirect` dans la valeur retournée par l'action du contrôleur.

En allant à l'URL `localhost:8080/firstspringmvc/bonjour`, on est redirigé vers `localhost:8080/firstspringmvc/home`

```
@GetMapping("/bonjour")
public RedirectView accueil () {
    RedirectView rv = new RedirectView();
    rv.setUrl("home");
    return rv;
}
```

En allant à l'URL `localhost:8080/firstspringmvc/bonjour`, on est redirigé vers `localhost:8080/firstspringmvc/home`

```
@GetMapping("/bonjour")
public RedirectView accueil () {
    RedirectView rv = new RedirectView();
    rv.setUrl("home");
    return rv;
}
```

Pour rediriger vers une route avec paramètre

```
@GetMapping("/bonjour")
public RedirectView accueil () {
    RedirectView rv = new RedirectView("hello");
    rv.addStaticAttribute("nom", "dalton");
    return rv;
}
```

Spring MVC 5

Solution avec le préfixe `redirect`

```
@GetMapping("/bonjour")
public String accueil (Model model) {
    return "redirect:hello?nom=dalton";
}
```

Spring MVC 5

Constats

- Dans une application **Spring MVC**, le rôle du contrôleur n'est pas d'afficher dans la console.
- C'est plutôt de communiquer avec les différents composants afin de préparer les différents éléments pour répondre à la requête utilisateur.
- Construire la réponse est le rôle d'une vue.

Spring MVC 5

Les vues avec **Spring**

- Permettent d'afficher des données
- Récupèrent les données envoyées par le contrôleur
- Doivent être créées dans le répertoire `WEB-INF` de `views`
- Peuvent être créées avec un simple code **HTML**, **JSP**, **JSTL** ou en utilisant un moteur de templates comme **Thymeleaf**, **Mustache**...

Spring MVC 5

Par défaut, **Spring** cherche une vue

- qui porte le même nom que le chemin demandé si le type de retour de la méthode est `void`.
- dont le nom est retourné par la méthode sinon.

Spring MVC 5

Modifions la vue `home.jsp`

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
   prefix="c" %>
<html>
<head>
        <title>Home</title>
</head>
<body>
<h1>
        Hello world!
</h1>
</body>
</html>
```

Spring MVC 5

Appelons `home.jsp` **depuis la méthode** `home()` **de** `HomeController`

```
@GetMapping({ "/home", "/" })
public void home() {

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

}
```

Spring MVC 5

Appelons `home.jsp` **depuis la méthode** `home()` **de** `HomeController`

```
@GetMapping({ "/home", "/" })
public void home() {

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

}
```


Vérifier qu'en allant à

- `localhost:8080/firstspringmvc/home` la vue `home.jsp` est affichée
- `localhost:8080/firstspringmvc/`, aucune vue n'est affichée

Spring MVC 5

Pour résoudre le problème précédent, la méthode `home()` doit retourner une chaîne de caractères contenant le nom de la vue

```
@GetMapping({ "/home", "/" })
public String home() {

    return "home";

}
```

Spring MVC 5

Pour résoudre le problème précédent, la méthode `home()` doit retourner une chaîne de caractères contenant le nom de la vue

```
@GetMapping({ "/home", "/" })
public String home() {

    return "home";

}
```


Vérifier que la vue `home.jsp` s'affiche pour

- `localhost:8080/firstspringmvc/home`
- `localhost:8080/firstspringmvc/`

Spring MVC 5

Question

Le contrôleur peut-il envoyer des données à la vue ?

Spring MVC 5

Question

Le contrôleur peut-il envoyer des données à la vue ?

Réponse

Oui, et pour le faire, **Spring** propose plusieurs solutions.

Spring MVC 5

3 Solutions proposées par **Spring** pour l'envoi de données

- `Model` (disponible depuis **Spring 3.x**)
- `ModelMap` (disponible depuis **Spring 2.x**)
- `ModelAndView` (disponible depuis **Spring 2.x**)

Spring MVC 5

Première solution avec l'interface `Model`

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

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

Spring MVC 5

Première solution avec l'interface `Model`

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

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

Explication

On injecte l'interface `Model` comme paramètre de la méthode pour envoyer les attributs à la vue.

Spring MVC 5

Comme en JEE, on utilise EL pour récupérer les données dans la vue

```
<%@ page language="java" contentType="text/html; charset=
   UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>hello.jsp</title>
</head>
<body>
        <h1>hello.jsp</h1>
        <p>Hello ${ prenom } ${ nom }</p>
</body>
</html>
```

Exactement comme en **JEE**

Spring MVC 5

Deuxième solution avec la classe `ModelMap`

```
@GetMapping("/hello")
public String sayHello(
                @RequestParam String nom,
                @RequestParam String prenom,
                ModelMap model) {

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

Spring MVC 5

Deuxième solution avec la classe `ModelMap`

```
@GetMapping("/hello")
public String sayHello(
                @RequestParam String nom,
                @RequestParam String prenom,
                ModelMap model) {

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

Explication

On injecte la classe `ModelMap` comme paramètre de la méthode pour envoyer les attributs à la vue.

Spring MVC 5

Troisième solution avec la classe `ModelAndView`

```
@GetMapping("/hello")
public ModelAndView sayHello(
                @RequestParam String nom,
                @RequestParam String prenom) {

        ModelAndView mv = new ModelAndView();
        mv.addObject("nom", nom);
        mv.addObject("prenom", prenom);
        mv.setViewName("hello");
        return mv;
}
```

Spring MVC 5

Troisième solution avec la classe `ModelAndView`

```
@GetMapping("/hello")
public ModelAndView sayHello(
                @RequestParam String nom,
                @RequestParam String prenom) {

        ModelAndView mv = new ModelAndView();
        mv.addObject("nom", nom);
        mv.addObject("prenom", prenom);
        mv.setViewName("hello");
        return mv;
}
```

Explication

On instancie ou on injecte `ModelAndView` et on l'utilise pour retourner une seule valeur contenant les attributs et le nom de la vue.

Spring MVC 5

On peut spécifier le nom de la vue à l'instanciation de `ModelAndView`

```
@GetMapping("/hello")
public ModelAndView sayHello(
                @RequestParam String nom,
                @RequestParam String prenom) {

        ModelAndView mv = new ModelAndView("hello");
        mv.addObject("nom", nom);
        mv.addObject("prenom", prenom);
        return mv;
}
```

Spring MVC 5

`Model` vs `ModelMap` vs `ModelAndView`

- `Model` (**Spring 3.x**) : interface permettant d'ajouter des attributs et les passer à la vue.
- `ModelMap` (**Spring 2.x**) : classe implémentant l'interface `Map` et permettant d'ajouter des attributs sous forme de `key - value` et les passer à la vue. On peut donc chercher un élément selon la valeur de la clé ou de la valeur.
- `ModelAndView` (**Spring 2.x**) : classe contenant à la fois un `ModelMap` pour les attributs et un `View Object`. Le contrôleur pourra ainsi retourner une seule valeur.

Spring MVC 5

Question

La vue peut envoyer de données à une méthode du contrôleur ?

Spring MVC 5

Question

La vue peut envoyer de données à une méthode du contrôleur ?

Réponse : oui

- Soit depuis un formulaire
- Soit avec un lien hypertexte

Exemple de construction de lien avec paramètre dans `home.jsp`

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
        <title>Home</title>
</head>
<body>
<h1>
        Hello world!
</h1>
        <a href="${pageContext.request.contextPath}/hello/wick">Lien
          absolu</a><br>
        <a href="/hello/wick">Lien relatif</a>
</body>
</html>
```

Exemple de construction de lien avec paramètre dans `home.jsp`

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
        <title>Home</title>
</head>
<body>
<h1>
        Hello world!
</h1>
        <a href="${pageContext.request.contextPath}/hello/wick">Lien
          absolu</a><br>
        <a href="/hello/wick">Lien relatif</a>
</body>
</html>
```

URLs de test

- `localhost:8080/firstspringmvc/home`
- `localhost:8080/firstspringmvc/`

Spring MVC 5

Pour voir la différence, ouvrez le code source de la page

```
<html>
<head>
        <title>Home</title>
</head>
<body>
<h1>
        Hello world!
</h1>
        <a href="/firstspringmvc/hello/wick">Lien absolu</a><br>
        <a href="/hello/wick">Lien relatif</a>
</body>
</html>
```

Spring MVC 5

Ressource statique

- Fichier de style **CSS**
- Fichier de script **JS**
- Image
- ...

Spring MVC 5

Appliquer un style dans une application **Spring MVC**

- Créer un fichier `style.css` dans `webapp/resources/css/` (le dossier `css` est à créer)
- Référencer cette feuille de style dans les vues

Spring MVC 5

Appliquer un style dans une application **Spring MVC**

- Créer un fichier `style.css` dans `webapp/resources/css/` (le dossier `css` est à créer)
- Référencer cette feuille de style dans les vues

Contenu de `style.css`

```
.first {
        color: blue;
}
```

Spring MVC 5

Pour tester, référençons le fichier `style.css` et utilisons la classe CSS `first` dans `hello.jsp`

```
<%@ page language="java" contentType="text/html; charset=
   UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>hello.jsp</title>
    <link rel="stylesheet" href="${pageContext.request.
      contextPath}/resources/css/style.css">
</head>
<body>
        <h1>hello.jsp</h1>
        <p class="first">Hello ${ prenom } ${ nom }</p>
</body>
</html>
```

Spring MVC 5

Exercice

- Créer un contrôleur `PersonneController` avec deux méthodes annotées respectivement par `@GetMapping("/addPersonne")` et `@PostMapping("/addPersonne")`.
- La méthode annotée par `@GetMapping("/addPersonne")` affiche la vue `addPersonne.jsp` contenant un formulaire composé de deux inputs (un pour le nom et un pour le prénom) et un bouton pour la soumission.
- La méthode annotée par `@PostMapping("/addPersonne")` récupère les données du formulaire et retourne la vue `confirm` qui affiche les données du formulaire.

Spring MVC 5

Correction : `PersonneController`

```
package org.eclipse.firstspringmvc.controller;

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

@Controller
public class PersonneController {

        @GetMapping("addPersonne")
        public String addPersonne() {
                return "addPersonne";
        }

        @PostMapping("addPersonne")
        public String addPersonne(@RequestParam String nom,
                                  @RequestParam String prenom,
                                  Model model) {
                model.addAttribute("nom", nom);
                model.addAttribute("prenom", prenom);
                return "confirm";
        }
}
```

Spring MVC 5

Correction : `addPersonne.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ajouter une nouvelle personne</title>
  </head>
<body>
  <h2>Ajouter une nouvelle personne</h2>
  <form method="POST" action="addPersonne">
        <div>
                Nom : <input type="text" name="nom">
        </div>
        <div>
                Prénom : <input type="text" name="prenom">
        </div>
        <button>Ajouter</button>
  </form>
</body>
</html>
```

Spring MVC 5

Correction : `confirm.jsp`

```
<%@ page language="java" contentType="text/html; charset=
   UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Page de confirmation</title>
</head>
<body>
  <p>${ prenom } ${ nom } a été ajoutée avec succès.</p>
</body>
</html>
```

Spring MVC 5

Avant de commencer, installer **MySQL** (s'il n'est pas installé)

- Aller à `https://dev.mysql.com/downloads/mysql/` et choisir la version à télécharger selon le système d'exploitation
- Lancer l'installation du fichier MSI sous Windows (fichier pkg de l'archive DMG sous MAC)

Spring MVC 5

Avant de commencer, voici le script SQL qui permet de créer la base de données utilisée dans ce cours

```
CREATE DATABASE mybase;

USE mybase;

CREATE TABLE personne(
num INT PRIMARY KEY AUTO_INCREMENT,
nom VARCHAR(30),
prenom VARCHAR(30)
)ENGINE=InnoDB;

SHOW TABLES;

INSERT INTO personne (nom, prenom) VALUES ("Wick", "John"),
 ("Dalton", "Jack");

SELECT * FROM personne;
```

Spring MVC 5

Plusieurs solutions avec **Spring**

- **Spring JDBC**
- **Spring Data JPA**
- ...

Spring MVC 5

Accès et traitement de données

- Utilisation de
  - **MySQL** pour le stockage de données
  - **Spring JDBC** pour la connexion et l'exécution de requête **SQL**
- Configuration avec
  - des fichiers **XML**
  - des classes **Java**

Étapes à suivre

- Tout d'abord, ajouter dans `pom.xml` les dépendances suivantes
  - une pour le driver **MySQL** (connecteur)
  - une pour **Spring JDBC**
- Ensuite ajouter 2 `beans` dans le conteneur
  - un bean `DataSource` pour spécifier les données concernant la connexion
  - un bean `JdbcTemplate` pour l'exécution de requête **SQL**
- Puis créer nos classes modèles
- Après créer une classe **DAO** pour chaque classe modèle
- Injecter le `JdbcTemplate` dans le **DAO**
- Injecter le **DAO** dans le contrôleur pour persister les données

Spring MVC 5

Les dépendances à ajouter dans `pom.xml`

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.17</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${org.springframework-version}</version>
</dependency>
```

Spring MVC 5

Dans `servlet-context.xml`**, ajoutons un** `bean` **contenant les données de la connexion (**`dataSource`**)**

```
<beans:bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <beans:property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
  <beans:property name="url" value="jdbc:mysql://localhost:3306/mybase?serverTimezone=UTC"/>
  <beans:property name="username" value="root" />
  <beans:property name="password" value="root" />
</beans:bean>
```

Spring MVC 5

Dans `servlet-context.xml`**, ajoutons un** `bean` **contenant les données de la connexion (**`dataSource`**)**

```
<beans:bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <beans:property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
  <beans:property name="url" value="jdbc:mysql://localhost:3306/mybase?serverTimezone=UTC"/>
  <beans:property name="username" value="root" />
  <beans:property name="password" value="root" />
</beans:bean>
```

Et un `bean` **pour** `jdbcTemplate`

```
<beans:bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <beans:property name="dataSource" ref="dataSource"/>
</beans:bean>
```

Spring MVC 5

Dans `servlet-context.xml`**, ajoutons un** `bean` **contenant les données de la connexion (**`dataSource`**)**

```
<beans:bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <beans:property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
  <beans:property name="url" value="jdbc:mysql://localhost:3306/mybase?serverTimezone=UTC"/>
  <beans:property name="username" value="root" />
  <beans:property name="password" value="root" />
</beans:bean>
```

Et un `bean` **pour** `jdbcTemplate`

```
<beans:bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <beans:property name="dataSource" ref="dataSource"/>
</beans:bean>
```

Sans oublier de scanner le package `org.eclipse.firstspringmvc.dao`

```
<context:component-scan
base-package="org.eclipse.firstspringmvc.controller, org.eclipse.firstspringmvc.dao" />
```

Créons la classe `Personne`

```
package org.eclipse.firstspringmvc.model;

public class Personne {

        private Long num;
        private String nom;
        private String prenom;

        public Personne() {
        }
        public Personne(String nom, String prenom) {
                this.nom = nom;
                this.prenom = prenom;
        }
        public Personne(Long num, String nom, String prenom) {
                this.num = num;
                this.nom = nom;
                this.prenom = prenom;
        }

  // + getters, setters et toString
}
```

Spring MVC 5

Et la classe `PersonneDao`

```
package org.eclipse.firstspringmvc.dao;

import org.eclipse.firstspringmvc.model.Personne;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

@Repository
public class PersonneDao {
        @Autowired
        JdbcTemplate jdbcTemplate;

        public int save(Personne personne) {
            return jdbcTemplate.update(
              "INSERT INTO Personne VALUES (?, ?, ?)", personne.getNum
                (), personne.getNom(), personne.getPrenom());
        }
}
```

Dans `PersonneController`, on injecte `PersonneDao` (il faut scanner le package `org.eclipse.firstspringmvc.dao`) et utilise ses méthodes

```
package org.eclipse.firstspringmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.eclipse.firstspringmvc.model;

@Controller
public class PersonneController {
        @Autowired
        private PersonneDao personneDao;

        @GetMapping("addPersonne")
        public String addPersonne() {
                return "addPersonne";
        }

        @PostMapping("addPersonne")
        public String addPersonne(@RequestParam String nom,
                                  @RequestParam String prenom,
                                  Model model) {
                Personne personne = new Personne(nom, prenom);
                System.out.println(personneDao.save(personne));
                model.addAttribute("nom", nom);
                model.addAttribute("prenom", prenom);
                return "confirm";
        }
}
```

Spring MVC 5

Contenu de la vue `addPersonne.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ajouter une nouvelle personne</title>
  </head>
<body>
  <h2>Ajouter une nouvelle personne</h2>
  <form method="POST" action="addPersonne">
        <div>
                Nom : <input type="text" name="nom">
        </div>
        <div>
                Prénom : <input type="text" name="prenom">
        </div>
        <button>Ajouter</button>
  </form>
</body>
</html>
```

Spring MVC 5

Contenu de la vue `confirm.jsp`

```
<%@ page language="java" contentType="text/html;
   charset=UTF-8"
     pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Page de confirmation</title>
</head>
<body>
        <p>${ prenom } ${ nom } a été ajoutée avec
           succès.</p>
</body>
</html>
```

Spring MVC 5

Quelques méthodes de `JdbcTemplate`

- `update` pour insérer modifier ou supprimer des données.
- `execute` pour exécuter une requête LDD.
- `queryForObject` pour lire des données.
- ...

Spring MVC 5

Dans `servlet-context.xml`, ajoutons le `bean` suivant pour utiliser les requêtes nommées

```
<beans:bean id="namedParameterJdbcTemplate"
class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
  <beans:constructor-arg ref="dataSource"/>
</beans:bean>
```

Spring MVC 5

Modifions la classe `PersonneDao`

```
package org.eclipse.firstspringmvc.dao;

import org.eclipse.firstspringmvc.model.Personne;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;

@Repository
public class PersonneDao {

    @Autowired
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    public int save(Personne personne) {
        String request = "INSERT INTO Personne VALUES (:num, :nom, :prenom)";
        MapSqlParameterSource params = new MapSqlParameterSource();
        params.addValue("num", personne.getNum());
        params.addValue("nom", personne.getNom());
        params.addValue("prenom", personne.getPrenom());
        return namedParameterJdbcTemplate.update(request, params);
    }
}
```

Rien à modifier dans `PersonneController`

```
package org.eclipse.firstspringmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.eclipse.firstspringmvc.model;

@Controller
public class PersonneController {
        @Autowired
        private PersonneDao personneDao;

        @GetMapping("addPersonne")
        public String addPersonne() {
                return "addPersonne";
        }

        @PostMapping("addPersonne")
        public String addPersonne(@RequestParam String nom,
                                  @RequestParam String prenom,
                                  Model model) {
                Personne personne = new Personne(nom, prenom);
                System.out.println(personneDao.save(personne));
                model.addAttribute("nom", nom);
                model.addAttribute("prenom", prenom);
                return "confirm";
        }
}
```

Spring MVC 5

On peut simplifier l'affectation de paramètres en associant un JavaBean à nos paramètres

```
package org.eclipse.firstspringmvc.dao;

import org.eclipse.firstspringmvc.model.Personne;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;

@Repository
public class PersonneDao {
    @Autowired
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    public int save(Personne personne) {
        String request = "INSERT INTO Personne VALUES (:num, :nom, :prenom)";
        SqlParameterSource params =  new BeanPropertySqlParameterSource(personne);
        return namedParameterJdbcTemplate.update(request, params);
    }
}
```

Spring MVC 5

Pour récupérer la valeur attribuée à la clé primaire

```
package org.eclipse.firstspringmvc.dao;

import org.eclipse.firstspringmvc.model.Personne;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;

@Repository
public class PersonneDao {

    @Autowired
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    public long save(Personne personne) {
        KeyHolder keyHolder = new GeneratedKeyHolder();
        String request = "INSERT INTO Personne VALUES (:num, :nom, :prenom)";
        SqlParameterSource params =  new BeanPropertySqlParameterSource(personne);
        namedParameterJdbcTemplate.update(request, params, keyHolder);
        return keyHolder.getKey().longValue();
    }
}
```

Pour mapper le résultat d'une requête en un objet `Personne`, on utilise l'interface `RowMapper`

```
package org.eclipse.firstspringmvc.mapper;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.eclipse.firstspringmvc.model.Personne;
import org.springframework.jdbc.core.RowMapper;

public class PersonneRowMapper implements RowMapper<Personne> {

    @Override
    public Personne mapRow(ResultSet rs, int rowNum) throws
       SQLException {
        Personne personne = new Personne();
        personne.setNum(rs.getLong("num"));
        personne.setNom(rs.getString("nom"));
        personne.setPrenom(rs.getString("prenom"));
        return personne;
    }
}
```

Ajoutons une méthode `findById()` dans `PersonneDao` qui utilise `PersonneRowMapper`

```
package org.eclipse.firstspringmvc.dao;

import org.eclipse.firstspringmvc.mapper.PersonneRowMapper;
import org.eclipse.firstspringmvc.model.Personne;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;

@Repository
public class PersonneDao {
        @Autowired
        NamedParameterJdbcTemplate namedParameterJdbcTemplate;

        public long save(Personne personne) {
                String request = "INSERT INTO Personne VALUES (:num, :nom, :prenom)";
                KeyHolder keyHolder = new GeneratedKeyHolder();
                SqlParameterSource params = new BeanPropertySqlParameterSource(personne);
                namedParameterJdbcTemplate.update(request, params, keyHolder);
                return keyHolder.getKey().longValue();
        }
        public Personne findById(long id) {
                String request = "SELECT * FROM Personne WHERE num = :num";
                MapSqlParameterSource params = new MapSqlParameterSource();
                params.addValue("num", id);
                return namedParameterJdbcTemplate.queryForObject(request, params, new
                   PersonneRowMapper());
        }
}
```

Spring MVC 5

Pour tester dans `PersonneController`

```
@GetMapping("/showPersonne/{num}")
public String showPersonne(@PathVariable Long num,
                           Model model) {

    model.addAttribute("personne", personneDao.findById(num));
    return "showPersonne";
}
```

Spring MVC 5

La vue `showPersonne.jsp`

```
<%@ page language="java" contentType="text/html; charset=
   UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Résultat de la recherche</title>
</head>
<body>
  <h2>Résultat de la recherche</h2>
  <p>Personne recherchée : ${ personne.prenom } ${
     personne.nom }</p>
</body>
</html>
```

Spring MVC 5

Rappel

- **JPA** : spécification (comme une interface en POO)
- **Hibernate**, **EclipseLink** ... : implémentation de **JPA**

Spring MVC 5

Spring Data

- Projet **Spring Framework**
- Simplifiant l'interaction avec différents systèmes de stockage de données
- Composé de plusieurs sous-projets
  - **Spring Data JPA**
  - **Spring Data REST**
  - **Spring Data MongoDB**
  - ...

Spring MVC 5

Spring Data JPA

- Sous-projet **Spring Data**
- **Spring Data JPA** = **Spring Data** + **JPA**
- Offrant plus d'abstraction d'accès aux données
- Fonctionnant avec un fournisseur implémentant **JPA**
  - **Hibernate**
  - **EclipseLink**
  - ...

Spring MVC 5

Accès et traitement de données

- Utilisation de
  - **MySQL** pour le stockage de données
  - **Hibernate** pour le mapping
  - **Spring-Data-JPA** pour la génération du **DAO**
- Configuration avec
  - des fichiers **XML**
  - des classes **Java**

Spring MVC 5

Étapes à suivre

- Tout d'abord, ajouter dans `pom.xml` les dépendances suivantes
  - une pour le driver `MySQL` (connecteur)
  - une pour `Hibernate`
  - une pour `Spring-Data-JPA`
- Ensuite ajouter 3 `beans` dans le conteneur
  - un bean `DataSource` pour spécifier les données concernant la connexion
  - un bean `EntityManagerFactory` pour la gestion d'entités
  - un bean `TransactionManager` pour la gestion de transactions
- Puis créer nos entités `JPA`
- Après créer des `repository` correspondants à nos entités
- Injecter le `repository` dans le contrôleur pour persister les données

Spring MVC 5

Dans `pom.xml`

- gardons la dépendance **MySQL**
- supprimons la dépendance **Spring-JDBC**

Spring MVC 5

Gardons la dépendance `MySQL`

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.17</version>
</dependency>
```

Spring MVC 5

Gardons la dépendance `MySQL`

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.17</version>
</dependency>
```

Et supprimons celle de `Spring JDBC`

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${org.springframework-version}</version>
</dependency>
```

Spring MVC 5

Et ajoutons les dépendances suivantes

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>2.2.3.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.10.Final</version>
</dependency>
```

Spring MVC 5

Si on utilise une version de JDK $\geq$ 9, on ajoute la dépendance suivante

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
</dependency>
```

Spring MVC 5

Dans `servlet-context.xml`**, gardons le** `bean dataSource`

```
<beans:bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <beans:property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
    <beans:property name="url" value="jdbc:mysql://localhost:3306/mybase?serverTimezone=UTC"/>
    <beans:property name="username" value="root" />
    <beans:property name="password" value="root" />
</beans:bean>
```

Spring MVC 5

Dans `servlet-context.xml`**, gardons le** `bean dataSource`

```
<beans:bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <beans:property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
    <beans:property name="url" value="jdbc:mysql://localhost:3306/mybase?serverTimezone=UTC"/>
    <beans:property name="username" value="root" />
    <beans:property name="password" value="root" />
</beans:bean>
```

Et supprimons les `bean` **pour** `jdbcTemplate` **et** `namedParameterJdbcTemplate`

```
<beans:bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
        <beans:property name="dataSource" ref="dataSource"/>
</beans:bean>

<beans:bean id="namedParameterJdbcTemplate"
class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
        <beans:constructor-arg ref="dataSource"/>
</beans:bean>
```

Spring MVC 5

Encore un `bean` pour l'entity manager

```
<beans:bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <beans:property name="dataSource" ref="dataSource" />
  <beans:property name="packagesToScan" value="org.eclipse.firstspringmvc.model" />
  <beans:property name="jpaVendorAdapter">
    <beans:bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
  </beans:property>
  <beans:property name="jpaProperties">
    <beans:props>
      <beans:prop key="hibernate.show_sql">true</beans:prop>
      <beans:prop key="hibernate.hbm2ddl.auto">update</beans:prop>
      <beans:prop key="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</beans:prop>
    </beans:props>
  </beans:property>
</beans:bean>
```

Spring MVC 5

Encore un `bean` pour l'entity manager

```
<beans:bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <beans:property name="dataSource" ref="dataSource" />
  <beans:property name="packagesToScan" value="org.eclipse.firstspringmvc.model" />
  <beans:property name="jpaVendorAdapter">
    <beans:bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
  </beans:property>
  <beans:property name="jpaProperties">
    <beans:props>
      <beans:prop key="hibernate.show_sql">true</beans:prop>
      <beans:prop key="hibernate.hbm2ddl.auto">update</beans:prop>
      <beans:prop key="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</beans:prop>
    </beans:props>
  </beans:property>
</beans:bean>
```

Et un `bean` pour les transactions

```
<beans:bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
  <beans:property name="entityManagerFactory" ref="entityManagerFactory" />
</beans:bean>
```

Spring MVC 5

Indiquons maintenant l'emplacement de nos repositories

```
<jpa:repositories base-package="org.eclipse.firstspringmvc.dao"/>
```

Spring MVC 5

Indiquons maintenant l'emplacement de nos repositories

```
<jpa:repositories base-package="org.eclipse.firstspringmvc.dao"/>
```

N'oublions pas d'ajouter l'espace de nom associé à JPA

```
xmlns:jpa="http://www.springframework.org/schema/data/jpa"
<!-- dans le schemaLocation -->
http://www.springframework.org/schema/data/jpa
http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
```

Créons une entité `Personne`

```
package org.eclipse.firstspringmvc.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Personne {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long num;
  private String nom;
  private String prenom;

  public Personne() {  }
  public Personne(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }
  // + getters, setters et toString
}
```

Spring MVC 5

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

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

Spring MVC 5

Le repository

```
package org.eclipse.firstspringmvc.dao;

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

import org.eclipse.firstspringmvc.model.Personne;

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

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

Spring MVC 5

Gardons la vue `addPersonne.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ajouter une nouvelle personne</title>
  </head>
<body>
  <h2>Ajouter une nouvelle personne</h2>
  <form method="POST" action="addPersonne">
        <div>
                Nom : <input type="text" name="nom">
        </div>
        <div>
                Prénom : <input type="text" name="prenom">
        </div>
        <button>Ajouter</button>
  </form>
</body>
</html>
```

Mettons à jour le contrôleur `PersonneController`

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

        @GetMapping("addPersonne")
        public String addPersonne() {
                return "addPersonne";
        }

        @PostMapping("addPersonne")
        public String addPersonne(@RequestParam String nom,
                                  @RequestParam String prenom,
                                  Model model) {

                Personne personne = new Personne(nom, prenom);
                System.out.println(personneRepository.save(personne));
                model.addAttribute("nom", nom);
                model.addAttribute("prenom", prenom);
                return "confirm";
        }
        @GetMapping("/showPersonne/{num}")
        public String showPersonne(@PathVariable Long num,
                                   Model model) {

            model.addAttribute("personne", personneRepository.findById(num));
            return "showPersonne";
        }
}
```

Spring MVC 5

Pour récupérer la liste de toutes les personnes

```
@GetMapping("showPersonnes")
public String showPersonnes(Model model) {
    model.addAttribute("personnes", personneRepository.findAll());
    return "showPersonnes";
}
```

Spring MVC 5

La vue `showPersonnes.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Liste de personnes</title>
  </head>
  <body>
    <h1>Liste de personnes</h1>
    <c:forEach items="${ personnes }" var="personne">
      <div>
        <c:out value="${ personne.prenom } ${ personne.nom }"/>
      </div>
    </c:forEach>
  </body>
</html>
```

Spring MVC 5

Exercice

- Dans `showPersonnes.jsp`, ajouter deux liens `deletePersonne/id` et `editPersonne/id` devant chaque personne affichée
- En cliquant sur `deletePersonne/id`, la personne avec l'identifiant `id` sera supprimée et `showPersonnes.jsp` sera rafraichie.
- En cliquant sur `editPersonne/id`, les valeurs (nom et prénom) de la personne avec l'identifiant `id` seront affichées dans des `input` dans la vue `editPersonne.jsp` avec un bouton `Enregistrer`.
- En cliquant sur le bouton `Enregistrer`, les nouvelles valeurs seront enregistrées et la page `showPersonnes.jsp` sera affichée.
- Dans `showPersonnes.jsp`, ajouter un lien qui permet d'afficher la vue `addPersonne.jsp`

Spring MVC 5

On peut aussi récupérer la liste de personnes par page

```
@GetMapping(value = "/showPersonnesByPage/{page}/{size}")
public String showAllByPage(
                        @PathVariable int page,
                        @PathVariable int size,
                        Model model) {

        Page<Personne> personnes = personneRepository.findAll(
           PageRequest.of(page, size));
        model.addAttribute("personnes", personnes.getContent());
        return "showPersonnes";
}
```


Les variables de chemin `page` et `size`

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

Spring MVC 5

Considérons le contenu suivant de la table `Personne`

Personne		
num	nom	prenom
1	Durand	Philippe
2	Leberre	Bernard
3	Benammar	Pierre
4	Hadad	Karim
5	Wick	John

Spring MVC 5

Considérons le contenu suivant de la table `Personne`

Personne		
num	nom	prenom
1	Durand	Philippe
2	Leberre	Bernard
3	Benammar	Pierre
4	Hadad	Karim
5	Wick	John

En allant à l'URL `localhost:8080/firstspringmvc/showAllByPage/1/2`, le résultat est

Personne		
num	nom	prenom
3	Benammar	Pierre
4	Hadad	Karim

Spring MVC 5

On peut aussi récupérer une liste triée de personnes

```
@GetMapping(value = "/showPersonnesSorted")
public String showAllSorted(Model model) {
    List<Personne> personnes = personneRepository.findAll
       (Sort.by("nom").descending());
    model.addAttribute("personnes", personnes);
    return "showPersonnes";
}
```


Explication

Le résultat a été trié selon la colonne `nom` dans l'ordre décroissant

Spring MVC 5

Les méthodes personnalisées

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

Spring MVC 5

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> findByNom(String nom);
}
```

Spring MVC 5

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> findByNom(String nom);
}
```


La méthode de recherche doit

- commencer par `findBy`
- respecter le **Camel Case**

Spring MVC 5

Code à ajouter dans `PersonneController`

```
@GetMapping("/findByNom")
public String findByNom() {
        return "findByNom";
}

@PostMapping("/findByNom")
public String findByNom(@RequestParam String nom, Model model)
   {
        var personnes = personneRepository.findByNom(nom);
        model.addAttribute("personnes", personnes);
        return "showPersonnes";
}
```

Spring MVC 5

La vue `findByNom.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Chercher une personne selon le nom</title>
</head>
<body>
        <h1>Chercher une personne selon le nom</h1>
        <form method="POST" action="findByNom">
                Nom : <input type="text" name=nom>
                <button>Chercher</button>
        </form>
</body>
</html>
```

Spring MVC 5

Ajoutons une méthode qui vérifie si la colonne `nom` contient le motif recherché

```
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> findByNom(String nom);
     List<Personne> findByNomContaining(String nom);
}
```

Spring MVC 5

Code à ajouter dans `PersonneController`

```
@GetMapping("/findByNom")
public String findByNom() {
        return "findByNom";
}

@PostMapping("/findByNom")
public String findByNom(@RequestParam String nom, Model model) {
        var personnes = personneRepository.findByNomContaining(nom);
        model.addAttribute("personnes", personnes);
        return "showPersonnes";
}
```

Spring MVC 5

Code à ajouter dans `PersonneController`

```
@GetMapping("/findByNom")
public String findByNom() {
        return "findByNom";
}

@PostMapping("/findByNom")
public String findByNom(@RequestParam String nom, Model model) {
        var personnes = personneRepository.findByNomContaining(nom);
        model.addAttribute("personnes", personnes);
        return "showPersonnes";
}
```

Aucune modification pour la vue.

Spring MVC 5

Dans la méthode précédente on a utilisé le mot-clé `Containing`

Mais, on peut aussi utiliser

- `Or`, `Between`, `Like`, `IsNull`...
- `StartingWith`, `EndingWith`, `Containing`, `IgnoreCase`
- `After`, `Before` pour les dates
- `OrderBy`, `Not`, `In`, `NotIn`
- Liste complète : `https://docs.spring.io/spring-data/jpa/docs/current/reference/html/`

Spring MVC 5

Exemple 2

```
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> findByNom(String nom);
     List<Personne> findByNomContaining(String nom);
     List<Personne> findByNomContainingAndPrenomContaining(String nom,
        String prenom);
}
```

Spring MVC 5

Code à ajouter dans `PersonneController`

```
@GetMapping("/findByNomAndPrenom")
public String findByNomAndPrenom() {
        return "findByNomAndPrenom";
}

@PostMapping("/findByNomAndPrenom")
public String findByNomAndPrenom(@RequestParam String nom,
   @RequestParam String prenom, Model model) {
        var personnes = personneRepository.
           findByNomContainingAndPrenomContaining(nom, prenom);
        model.addAttribute("personnes", personnes);
        return "showPersonnes";
}
```

Spring MVC 5

La vue `findByNomAndPrenom.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Chercher selon nom et prénom</title>
</head>
<body>
        <h1>Chercher une personne selon nom et prénom</h1>
        <form method="POST" action="findByNomAndPrenom">
                <div>
                        Nom : <input type="text" name="nom">
                </div>
                <div>
                        Prénom : <input type="text" name="prenom">
                </div>
                <button>Chercher</button>
        </form>
</body>
</html>
```

Spring MVC 5

Exercice

- Créer une vue `findPersonne.jsp` contenant une seule zone de saisie (motif) et un bouton de recherche.
- La vue `findPersonne.jsp` permettra à l'utilisateur de saisir une chaîne qu'on doit vérifier sa présence dans les colonnes nom ou prénom dans la base de données.
- Les personnes correspondantes seront affichées dans la vue `showPersonnes.jsp`.
- N'oublions pas de définir une nouvelle méthode de recherche dans `PersonneRepository`.

Spring MVC 5

On peut utiliser l'annotation `Query` pour exécuter des requêtes JPQL (Java Persistence Query Language)

```
package org.eclipse.firstspringmvc.dao;

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

import java.util.List;

import org.eclipse.firstspringmvc.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne,
   Long> {

    @Query("select p from Personne p where p.nom = ?1")
    List<Personne> chercherSelonLeNom(String nom);
    List<Personne> findByNom(String nom);
    List<Personne> findByNomContaining(String nom);
    List<Personne> findByNomContainingAndPrenomContaining(String nom,
       String prenom);
}
```

Spring MVC 5

On peut utiliser l'annotation `Query` pour exécuter des requêtes JPQL (Java Persistence Query Language)

```
package org.eclipse.firstspringmvc.dao;

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

import java.util.List;

import org.eclipse.firstspringmvc.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne,
   Long> {

    @Query("select p from Personne p where p.nom = ?1")
    List<Personne> chercherSelonLeNom(String nom);
    List<Personne> findByNom(String nom);
    List<Personne> findByNomContaining(String nom);
    List<Personne> findByNomContainingAndPrenomContaining(String nom,
       String prenom);
}
```

Spring MVC 5

On peut utiliser l'annotation `@Param` pour nommer les paramètres

```
package org.eclipse.firstspringmvc.dao;

import java.util.List;

import org.eclipse.firstspringmvc.model.Personne;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

public interface PersonneRepository extends JpaRepository<Personne, Long> {
    @Query("select p from Personne p where p.nom = ?1")
    List<Personne> chercherSelonNom(String nom);
    @Query("select p from Personne p where p.prenom = :prenom and p.nom = :nom")
    List<Personne> chercherSelonNomEtPrenom(@Param("nom") String nom, @Param("prenom") String
      prenom);
    List<Personne> findByNom(String nom);
    List<Personne> findByNomContaining(String nom);
    List<Personne> findByNomContainingAndPrenomContaining(String nom, String prenom);
}
```

Spring MVC 5

On peut utiliser l'annotation `@Query` aussi pour exécuter des expressions SpEL (Spring Expression Language)

```
package org.eclipse.firstspringmvc.dao;

import java.util.List;

import org.eclipse.firstspringmvc.model.Personne;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

public interface PersonneRepository extends JpaRepository<Personne, Long> {
    @Query("SELECT p FROM #{#entityName} p WHERE p.nom = ?1")
    List<Personne> chercherSelonNom(String nom);
    @Query("select p from Personne p where p.prenom = :prenom and p.nom = :nom")
    List<Personne> chercherSelonNomEtPrenom(@Param("nom") String nom,@Param("prenom") String
      prenom);
    List<Personne> findByNom(String nom);
    List<Personne> findByNomContaining(String nom);
    List<Personne> findByNomContainingAndPrenomContaining(String nom, String prenom);
}
```