Spring MVC : 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

Spring : les fondamentaux

Comment créer un projet Spring

- Aller dans le menu `File > New > Other`
- Saisir `Maven` et choisir `Maven Project`
- Cliquer sur `Next` puis
- Sélectionner `maven-archetype-webapp`
- Remplir les deux champs `Group Id` (*org.eclipse* par exemple) et `Artifact Id` (*FirstSpring* par exemple) et Valider

Spring : les fondamentaux

Le projet est signalé en rouge ⇒ Choisir un serveur Apache

- Un clic droit sur le nom du projet et choisir `Properties`
- Sélectionner un serveur de la liste du menu `Targeted Runtimes` (Si la liste est vide, ajouter un en cliquant sur `New`)
- Ensuite valider en cliquant sur `Apply and Close`

Spring : les fondamentaux

Le projet est signalé en rouge ⇒ Choisir un serveur Apache

- Un clic droit sur le nom du projet et choisir `Properties`
- Sélectionner un serveur de la liste du menu `Targeted Runtimes` (Si la liste est vide, ajouter un en cliquant sur `New`)
- Ensuite valider en cliquant sur `Apply and Close`

Pour exécuter

- Un clic droit sur le nom du projet et dans le menu `Run As` choisir `Run on Server`
- Le fameux `Hello World!` est affiché dans le navigateur

Configuration du projet

Si le `src/main/java` n'existe pas

- Un clic droit sur le nom du projet et choisir `Properties`
- Aller dans le menu `Java Build Path` et cliquer sur la rubrique `Source`
- Ensuite supprimer les deux éléments de la liste marqués par `(missing)`
- Puis cliquer sur `Add Folder`
- Par la suite, sélectionner `main` et cliquer sur le bouton `Create New Folder` et saisir `java`
- Enfin, tout valider.

Configuration du projet

Ajouter les dépendances nécessaires pour la réalisation du projet

- Ouvrir le fichier de configuration de Maven `pom.xml`
- Aller dans l'onglet `Dependencies`
- Cliquer sur le bouton `Add`
- Dans la zone de `Enter groupId, artifactId...`, saisir les dépendances nécessaires, une par une (Ici, on va ajouter : `servlet-api`, `jstl`, `spring-webmvc` et `spring-context`.
- Enfin, enregistrer pour démarrer le téléchargement (il faut une connexion internet).
- Pour vérifier l'importation de ces modules, aller dans `Maven Dependencies` de votre projet.

Configuration du projet

La liste de dépendance ajoutée

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet</
      groupId>
    <artifactId>servlet-api</
      artifactId>
    <version>2.5</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</
      groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</
      groupId>
    <artifactId>spring-context</
      artifactId>
    <version>5.0.5.RELEASE</
      version>
  </dependency>
  <dependency>
    <groupId>org.springframework</
      groupId>
    <artifactId>spring-webmvc</
      artifactId>
    <version>5.0.5.RELEASE</
      version>
  </dependency>
</dependencies>
```

Le dispatcher et les contrôleurs

Déclarer le dispatcher **dans le** web.xml

```
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
␣␣␣␣␣␣␣␣http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/dispatcher-servlet.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.
      ContextLoaderListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</
      servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

Le dispatcher et les contrôleurs

Explication

- On commence par préciser les namespaces à utiliser dans le projet.
- Dans <context-param>, on indique les éléments Spring qui seront partagés par tous les contrôleurs.
- Dans <listener>, on charge l'écouteur de Spring.
- Ensuite, on déclare le dispatcher (le contrôleur frontal de Spring, qui est une servlet, qu'on peut lui attribuer n'importe quel nom) et on lui redirige toutes requêtes /. L'emplacement de ce dispatcher est indiqué dans la balise <param-value> de <context-param>

Le dispatcher et les contrôleurs

Allons voir le dispatcher-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
␣␣http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
␣␣http://www.springframework.org/schema/context
␣␣http://www.springframework.org/schema/context/spring-context-4.0.xsd
␣␣http://www.springframework.org/schema/mvc
␣␣http://www.springframework.org/schema/mvc/spring-mvc.xsd
␣␣http://www.springframework.org/schema/tx␣http://www.springframework.
   org/schema/tx/spring-tx-3.0.xsd">
  <mvc:annotation-driven />
  <context:annotation-config></context:annotation-config>
  <bean class="org.springframework.web.servlet.view.
     InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
  </bean>
  <context:component-scan base-package="org.eclipse.controller"/>
</beans>
```

Le dispatcher et les contrôleurs

Explication

- On commence par préciser les namespaces à utiliser dans le projet.
- Les balises `<mvc:annotation-driven />` et `<context:annotation-config>` permettent d'utiliser des annotations telles que `@Controller`...
- Le bean permet d'indiquer que nos vues seront dans le répertoire `views` (que nous devons créer) et qui auront comme extension `.jsp` (donc pas besoin d'indiquer ces informations qu'on fait appel à une vue)
- Le dernier permet de préciser le package de nos contrôleurs qui est ici `org.eclipse.controller` (d'où l'intérêt de le créer dans `src/main/jave`.

Le dispatcher et les contrôleurs

Créons un premier contrôleur (classe Java) que nous appelons `HomeController`

```
package org.eclipse.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 `dispatcher` et les contrôleurs

Explication

- La première ligne indique que notre contrôleur se trouve dans le package `org.eclipse.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 `dispatcher` et les contrôleurs

Testons tout cela

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

Le `dispatcher` et les contrôleurs

Testons tout cela

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

Constats

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

Les vues

Rôle

- 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 des moteurs de templates comme `Thymeleaf`...

Les vues

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

```
<%@ 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 org.eclipse.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 String sayHello() {
                return "hello";
        }
}
```

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

Les vues

Deux questions

- Comment passer des paramètres 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 org.eclipse.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")
  public String sayHello(Model model) {
    System.out.println("I␣am␣here");
    model.addAttribute("nom", "Wick");
    return "hello";
  }
}
```

Dans le `return`, 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 <%= request.getAttribute("nom") %>
</body>
</html>
```

Exactement comme dans la plateforme JEE

Les vues

Comment le contrôleur récupère les paramètres d'une requête ?

```
package org.eclipse.controller;

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

@Controller
public class HomeController {

  @RequestMapping(value="/hello", method = RequestMethod.GET,
     params = {"nom"})
  public String sayHello(@RequestParam(value = "nom") String
     nom, Model model) {
    model.addAttribute("name", nom);
    return "hello";
  }
}
```

Les vues

Pour tester, il faut aller sur l'URL

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

Explication

- `params = {"nom"}` présente la liste des paramètres à récupérer de la requête (c'est facultatif)
- La méthode `sayHello` prend deux paramètres dont le premier est annoté par `@RequestParam(value = "nom") String nom` : cette annotation permet de récupérer la valeur du paramètre de la requête HTTP est de l'affecter au paramètre de la méthode `nom`.
- Ensuite nous passons cette valeur à la vue pour qu'elle l'affiche

Les vues

Comment le contrôleur récupère un chemin variable ?

```
package org.eclipse.controller;

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

@Controller
public class HomeController {

  @RequestMapping(value = "/hello/{nom}", method =
     RequestMethod.GET)
  public String hello2(@PathVariable String nom,Model model) {

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

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

Les vues

Remarque

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

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

Les vues

Une deuxième façon de faire en utilisant `ModelAndView`

```
package org.eclipse.controller;

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

@Controller
public class HomeController {

  @RequestMapping(value="/hello", method = RequestMethod.GET,
     params = {"nom"})
  public ModelAndView sayHello(@RequestParam(value = "nom")
     String nom) {
    ModelAndView mv = new ModelAndView();
    mv.setViewName("hello");
    mv.addObject("name", nom);
    return mv;
  }
}
```

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`)
- ...

Configuration du projet avec les annotations

Rôle

- Remplacer le contrôleur frontal (`dispatcher-servlet`) par une nouvelle classe (que nous appellerons, par exemple, `ApplicationConfig`) et qui utilisera les annotations pour faire le travail du contrôleur frontal
- Remplacer le contenu du `web.xml` par une classe qui étend la classe `AbstractAnnotationConfigDispatcherServletInitializer`

Configuration du projet avec les annotations

La classe `ApplicationConfig`

```
package org.eclipse.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.view.
   InternalResourceViewResolver;

@Configuration
@ComponentScan({"org.eclipse.controller"})
public class ApplicationConfig {
  @Bean
  public InternalResourceViewResolver setup() {
    InternalResourceViewResolver viewResolver = new
       InternalResourceViewResolver();
    viewResolver.setPrefix("/WEB-INF/views/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
  }
}
```

Les vues

Remarque

- Pour que notre nouvelle classe (que nous appellerons `MyWebInitializer`) hérite de la classe `AbstractAnnotationConfigDispatcherServletInitializer`, on peut le préciser au moment de la création en cliquant sur `Browse` du champ `Superclass:`

Configuration du projet avec les annotations

La classe `MyWebInitializer`

```
package org.eclipse.configuration;

import org.springframework.web.servlet.support.
   AbstractAnnotationConfigDispatcherServletInitializer;

public class MyWebInitializer extends
   AbstractAnnotationConfigDispatcherServletInitializer {
        @Override
        protected Class<?>[] getRootConfigClasses() {
                return null;
        }

        @Override
        protected Class<?>[] getServletConfigClasses() {
                return new Class [] {ApplicationConfig.class};
        }

        @Override
        protected String[] getServletMappings() {
                return new String [] {"/"};
        }
}
```

Configuration du projet avec les annotations

Le nouveau contenu du fichier `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
     instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/
     javaee http://java.sun.com/xml/ns/javaee/web-
     app_2_5.xsd">
</web-app>
```

Le fichier `dispatcher-servlet.xml` est à supprimer

Le modèle

Modèle : accès et traitement de données

- Utilisation de JPA et Hibernate
- Possible avec les fichiers xml, yml ou avec les annotations (`Repository`, `Service`... et `Autowired` pour l'injection de dépendance)

Organisation du projet

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

Le modèle

Étapes à suivre

- Tout d'abord, modifier notre classe de configuration par l'ajout des données relatives à la base de données (connexion, driver...)
- Ensuite ajouter dans `pom.xml` les dépendances nécessaires relatives à `Mysql`, `JPA` et `Hibernate`
- Puis créer nos entités `JPA`
- Après créer des `repository` correspondants à nos entités
- Utiliser enfin les `repository` dans le contrôleur pour persister les données

Le modèle

La classe `ApplicationConfig`

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.
   LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import javax.persistence.EntityManagerFactory;
@Configuration
@ComponentScan("org.eclipse.controllers")
@EnableJpaRepositories("org.eclipse.dao")
@EntityScan("org.eclipse.entities")
public class ApplicationConfig {
  @Bean
  public InternalResourceViewResolver setup() {
    InternalResourceViewResolver viewResolver = new
       InternalResourceViewResolver();
    viewResolver.setPrefix("/WEB-INF/views/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
  }
```

Le modèle

La classe `ApplicationConfig` (suite)

```
@Bean
public DataSource dataSource() {
  System.out.println("in datasoure");
  DriverManagerDataSource dataSource = new DriverManagerDataSource();
  dataSource.setDriverClassName("com.mysql.jdbc.Driver");
  dataSource.setUrl("jdbc:mysql://localhost:3306/myBase");
  dataSource.setUsername("root");
  dataSource.setPassword("");
  return dataSource;
}
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory()
   {
  HibernateJpaVendorAdapter vendorAdapter = new
    HibernateJpaVendorAdapter();
         vendorAdapter.setGenerateDdl(true);
  LocalContainerEntityManagerFactoryBean factory = new
    LocalContainerEntityManagerFactoryBean();
     factory.setJpaVendorAdapter(vendorAdapter);
     factory.setPackagesToScan("org.eclipse.entities");
     factory.setDataSource(dataSource());
     return factory;
}
```

Le modèle

La classe `ApplicationConfig` (suite)

```
  @Bean
  public PlatformTransactionManager
     transactionManager(EntityManagerFactory emf){
        JpaTransactionManager transactionManager =
           new JpaTransactionManager();
        transactionManager.setEntityManagerFactory(
           emf);
        return transactionManager;
  }
}
```

Le modèle

`pom.xml` **(contenu précédent)**

```
<dependencies>
  <dependency>
    <groupId>javax.servlet</
      groupId>
    <artifactId>servlet-api</
      artifactId>
    <version>2.5</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</
      groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</
      groupId>
    <artifactId>spring-context</
      artifactId>
    <version>5.0.5.RELEASE</
      version>
  </dependency>
  <dependency>
    <groupId>org.springframework</
      groupId>
    <artifactId>spring-webmvc</
      artifactId>
    <version>5.0.5.RELEASE</
      version>
  </dependency>
```

Le modèle

`pom.xml` **(ce qu'il faut ajouter)**

```
<dependency>
  <groupId>org.springframework.
    data</groupId>
  <artifactId>spring-data-jpa</
    artifactId>
  <version>2.0.6.RELEASE</
    version>
</dependency>
<dependency>
  <groupId>org.hibernate.javax.
    persistence</groupId>
  <artifactId>hibernate-jpa-2.1-
    api</artifactId>
  <version>1.0.0.Final</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-
    java</artifactId>
  <version>5.1.46</version>
</dependency>
<dependency>
  <groupId>org.springframework.
    boot</groupId>
  <artifactId>spring-boot-
    starter-data-jpa</artifactId
    >
  <version>2.0.1.RELEASE</
    version>
</dependency>
<dependency>
  <groupId>org.hibernate.common<
    /groupId>
  <artifactId>hibernate-commons-
    annotations</artifactId>
  <version>5.0.1.Final</version>
</dependency>
</dependencies>
```

Le modèle

L'entité `Personne`

```
package org.eclipse.entities;

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

Le repository

```
package org.eclipse.dao;

import java.util.List;

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

import org.eclipse.entities.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 `index.jsp`

```
<html>
<body>
<h2>Hello World!</h2>
<form action="add">
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

```
package org.eclipse.controller;
 //les imports
import org.eclipse.dao.PersonneRepository;
import org.eclipse.entities.Personne;
@Controller
public class PersonneController {
  @Autowired
  private PersonneRepository personneRepository;
  @RequestMapping(value="/add", method = RequestMethod.GET, params = {"
     nom","prenom"})
  public  ModelAndView sayHello(@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 <%= request.getAttribute("name") %> <
     %= request.getAttribute("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
- `findAll()` : supprimer tout
- `flush()` : modifier
- `count()`, `exists()`, `existsById()`...

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.dao;

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

import org.eclipse.entities.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 utiliser l'annotation `Query`

```
package org.eclipse.dao;

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

import org.eclipse.entities.Personne;

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

Le modèle

La classe `ApplicationConfig` devient trop chargée, solution ?

- Mettre tout ce qui concerne les vues et les contrôleurs dans un fichier `MvcConfig` (voir les slides suivants)
- Annoter les deux classes de configuration par `@EnableWebMvc`
- Mettre à jour le fichier `MyWebInitializer`

Le modèle

La classe `MvcConfig`

```
package org.eclipse.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.
   ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.
   WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.
   InternalResourceViewResolver;
@Configuration
public class MvcConfig extends WebMvcConfigurerAdapter{
  @Bean
  public InternalResourceViewResolver setup() {
    InternalResourceViewResolver viewResolver = new
       InternalResourceViewResolver();
    viewResolver.setPrefix("/WEB-INF/views/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
  }
}
```

Depuis Java 8, `WebMvcConfigurerAdapter` est dépréciée

On peut donc la remplacer par

```
package com.example.configuration;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.
   ViewResolverRegistry;
import org.springframework.web.servlet.config.annotation.
   WebMvcConfigurer;
@EnableWebMvc
@Configuration
public class MvcConfig implements WebMvcConfigurer{

  @Override
  public void configureViewResolvers(ViewResolverRegistry registry) {
    registry.jsp("/WEB-INF/views/", ".jsp");
  }
}
```

Le modèle

La classe `ApplicationConfig` devient ainsi :

```
package org.eclipse.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.view.
   InternalResourceViewResolver;
@EnableWebMvc
@Configuration
@ComponentScan(basePackages = "org.eclipse")
@EnableJpaRepositories("org.eclipse.dao")
@EntityScan("org.eclipse.entities")
public class ApplicationConfig {
  @Bean
  public DataSource dataSource() {
    System.out.println("in datasoure");
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/myBase");
    dataSource.setUsername("root");
    dataSource.setPassword("");
    return dataSource;
  }
```

Le modèle

La classe `ApplicationConfig` (suite)

```
  @Bean
  public LocalContainerEntityManagerFactoryBean entityManagerFactory()
     {
    HibernateJpaVendorAdapter vendorAdapter = new
       HibernateJpaVendorAdapter();
            vendorAdapter.setGenerateDdl(true);
    LocalContainerEntityManagerFactoryBean factory = new
       LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(vendorAdapter);
        factory.setPackagesToScan("org.eclipse.entities");
        factory.setDataSource(dataSource());
        return factory;
  }
  @Bean
  public PlatformTransactionManager transactionManager(
     EntityManagerFactory emf){
        JpaTransactionManager transactionManager = new
           JpaTransactionManager();
        transactionManager.setEntityManagerFactory(emf);
        return transactionManager;
  }
}
```

Configuration du projet avec les annotations

La classe `MyWebInitializer`

```
package org.eclipse.configuration;

import org.springframework.web.servlet.support.
   AbstractAnnotationConfigDispatcherServletInitializer;

public class MyWebInitializer extends
   AbstractAnnotationConfigDispatcherServletInitializer {
        @Override
        protected Class<?>[] getRootConfigClasses() {
                return null;
        }

        @Override
        protected Class<?>[] getServletConfigClasses() {
                return new Class [] {ApplicationConfig.class, MvcConfig
                   .class};
        }

        @Override
        protected String[] getServletMappings() {
                return new String [] {"/"};
        }
}
```