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

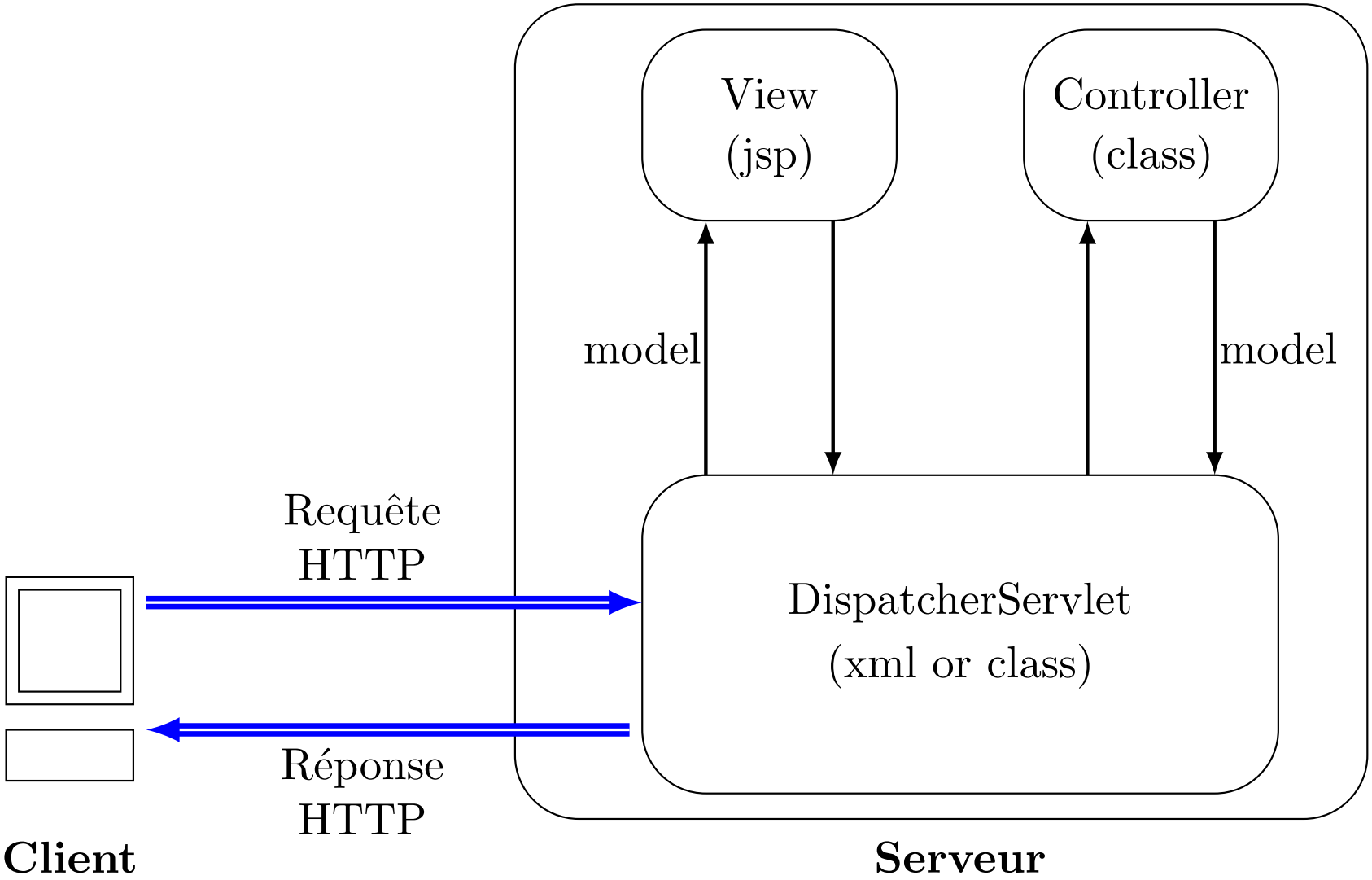
Plan

Spring MVC : Introduction

Spring MVC

- un des frameworks Spring
- permettant de simplifier la création d'applications web
- masquant l'utilisation de l'API Servlet de JEE grâce aux annotations `@RequestParam`...
- implémentant le patron de conception MVC 2
- basé sur le concept d'injection de dépendances
- offrant la possibilité de gérer le service REST

Architecture d'une application Spring MVC



Architecture d'une application Spring MVC

Explication

- Le contrôleur frontal intercepte la requête HTTP du client pour la rediriger vers le contrôleur approprié (selon la route)
- Le contrôleur frontal est implémenté soit par une classe Java (Projet configurable par classe ou par annotation) soit par un fichier XML (projet configurable par XML)
- Le contrôleur traite la requête, prépare un modèle et retourne le nom d'une vue au contrôleur frontal
- Le contrôleur frontal transmet le modèle à la vue afin de préparer la réponse puis retourne cette dernière au client

Spring MVC : les fondamentaux

Créons un projet Spring MVC avec dépendances gérées par `Maven`

- Aller dans `File > New > Other`
- Chercher `Spring`
- Choisir `Spring Legacy Project` et cliquer sur `Next`
- Saisir `FirstSpringMvc` dans `Project name` puis sélectionner `Spring MVC Project` et cliquer sur `Next`
- Saisir `org.eclipse.FirstSpringMvc` et valider

Spring MVC : les fondamentaux

Créons un projet Spring MVC avec dépendances gérées par `Maven`

- Aller dans `File > New > Other`
- Chercher `Spring`
- Choisir `Spring Legacy Project` et cliquer sur `Next`
- Saisir `FirstSpringMvc` dans `Project name` puis sélectionner `Spring MVC Project` et cliquer sur `Next`
- Saisir `org.eclipse.FirstSpringMvc` et valider

Exécuter le projet

`Hello World!` est affiché en allant à `localhost:8080/FirstSpringMvc/`

Spring MVC : les fondamentaux

Le fichier `web.xml`

- Une application Spring MVC utilise les mécanismes de la spécification Java EE
- La configuration d'un projet Java EE, et aussi Spring MVC, passe par un fichier `web.xml` situé dans `WEB-INF`
- Dans ce fichier, et au sein d'une application Spring MVC, on trouve la déclaration du contrôleur frontal (`DispatcherServlet`)

Contenu du `web.xml`

```
<web-app ...>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>
  <listener><listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class></listener>
  <servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</
       servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</
         param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

Le contrôleur frontal : `dispatcher`

Explication

- Les attributs de la balise `web-app` sont les namespaces à utiliser dans le projet.
- Dans `<context-param>`, on indique les éléments Spring qui sont partagés par tous les contrôleurs.
- Dans `<listener>`, on charge l'écouteur de Spring.
- Ensuite, on déclare le contrôleur frontal `DispatcherServlet` et on lui redirige toutes les requêtes (avec `/`).
- Le contrôleur frontal charge les paramètres de l'application à partir d'un fichier indiqué dans la balise `<param-value>` de `<init-param>`

Le contrôleur frontal : `dispatcher`

Allons voir `servlet-context.xml` (première partie)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">
```

Explication

- Spring commence par charger et affecter le contenu de certaines librairies indispensables pour la validation de ce fichier de configuration
- Ces librairies seront chargées dans les trois `namespace` suivant : `mvc`, `beans` et `context`
- Ces trois `namespace` permettront de configurer une application Spring MVC

Le contrôleur frontal : `dispatcher`

`servlet-context.xml` **(deuxième partie)**

```
  <!-- Enables the Spring MVC @Controller programming model -->
  <annotation-driven />

  <!-- Handles HTTP GET requests for /resources/** by efficiently
     serving up static resources in the ${webappRoot}/resources
     directory -->
  <resources mapping="/resources/**" location="/resources/" />

  <!-- Resolves views selected for rendering by @Controllers to .jsp
     resources in the /WEB-INF/views directory -->
  <beans:bean class="org.springframework.web.servlet.view.
     InternalResourceViewResolver">
    <beans:property name="prefix" value="/WEB-INF/views/" />
    <beans:property name="suffix" value=".jsp" />
  </beans:bean>

  <context:component-scan base-package="org.eclipse.FirstSpringMvc" />
</beans:beans>
```

Le contrôleur frontal : `dispatcher`

Explication

- La balise `<annotation-driven />` active les annotations permettant d'utiliser un contrôleur Spring MVC telles que `@Controller`...
- La balise `<resources mapping="/resources/**" location="/resources/" />` indique l'emplacement de fichiers statiques (`js`, `css`...). Ce répertoire `resources` est situé dans `webapp`
- Le bean permet d'indiquer que nos vues seront dans le répertoire `views` (situé dans `WEB-INF`) et qui auront comme extension `.jsp`
- Le dernier permet de préciser l'emplacement des contrôleurs de l'application : le package `org.eclipse.FirstSpringMvc`

Le contrôleur frontal : `dispatcher`

Restructuration de packages

- Pour rester cohérent, renommons le package contenant les contrôleurs `org.eclipse.FirstSpringMvc.controller`
- Pour cela, clic droit sur le package et aller dans `Refactor > Rename`
- Saisir le nouveau nom `org.eclipse.FirstSpringMvc.controller` et valider

Dans `servlet-context.xml`, vérifions que l'emplacement de contrôleurs a bien été mis-à-jour.

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

Le contrôleur frontal : `dispatcher`

Restructuration de packages

- Pour rester cohérent, renommons le package contenant les contrôleurs `org.eclipse.FirstSpringMvc.controller`
- Pour cela, clic droit sur le package et aller dans `Refactor > Rename`
- Saisir le nouveau nom `org.eclipse.FirstSpringMvc.controller` et valider

Dans `servlet-context.xml`, vérifions que l'emplacement de contrôleurs a bien été mis-à-jour.

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

Relancer le projet

Le contrôleur frontal : `dispatcher`

Dans `pom.xml`, mettre à jour la version de Spring utilisé

```
<properties>
  ...
  <org.springframework-version>5.1.1.RELEASE</org.
    springframework-version>
  ...
</properties>
```

Le contrôleur frontal : `dispatcher`

Supprimer la dépendance suivante

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>${org.springframework-version}</version>
  <exclusions>
    <!-- Exclude Commons Logging in favor of SLF4j -->
    <exclusion>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

La dépendance `spring-webmvc 5` est suffisante pour télécharger `spring-context`.

Le contrôleur frontal : `dispatcher`

Supprimer la dépendance suivante

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>${org.springframework-version}</version>
  <exclusions>
    <!-- Exclude Commons Logging in favor of SLF4j -->
    <exclusion>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

La dépendance `spring-webmvc 5` est suffisante pour télécharger `spring-context`.

Relancer le projet

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

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="/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 `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 (`/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 org.eclipse.FirstSpringMvc.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/FirstSpring/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 {
   ...
}
```

Les paramètres de requête

Les paramètres de requête

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

Les paramètres de requête

Comment le contrôleur récupère les paramètres de requête ?

```
package org.eclipse.FirstSpringMvc.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)
  public void sayHello(@RequestParam(value = "nom") String nom)
      {

    System.out.println("hello " + nom);
  }
}
```

Les paramètres de requête

Pour tester, il faut aller sur l'URL

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

Explication

- 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 `nom` de la méthode.

Les paramètres de requête

Pour tester, il faut aller sur l'URL

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

Explication

- 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 `nom` de la méthode.

Remarque

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

Les paramètres de requête

Peut-on accéder à `localhost:8080/FirstSpringMvc/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 void sayHello(@RequestParam(value="nom",
     required=false) String nom) {

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

  }
}
```

Les paramètres de requête

Il est possible aussi de préciser une valeur par défaut

```
@Controller
public class HomeController {

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

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

  }
}
```

Les variables de chemin

Les variables de chemin

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

Les variables de chemin

Comment le contrôleur récupère une variable de chemin ?

```
package org.eclipse.FirstSpringMvc.controller;

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

@Controller
public class HomeController {

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

    System.out.println("hello " + nom);
  }
}
```

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

Les vues

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 les différents composants
- Afficher la réponse est le rôle d'une 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 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.FirstSpringMvc.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";
  }
}
```

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

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 org.eclipse.FirstSpringMvc.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 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 ${ nom }
  </body>
</html>
```

Exactement comme dans la plateforme JEE

Les vues

Une deuxième façon en utilisant `ModelMap`

```
package org.eclipse.FirstSpringMvc.controller;

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

@Controller
public class HomeController {

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

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

Les vues

Une troisième façon en utilisant `ModelAndView`

```
package org.eclipse.FirstSpringMvc.controller;

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

@Controller
public class HomeController {

  @GetMapping(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 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`)
- ...

Les vues

Appliquer un style dans une application Spring MVC

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

Les vues

Le fichier `file.css`

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

Les vues

Pour tester, référençons le fichier `file.css` et utilisons la classe CSS `first` dans la vue

```
<%@ page language="java" contentType="text/html;
   charset=UTF-8"  pageEncoding="UTF-8"%>
<html>
  <head>
    <link rel="stylesheet" href="${pageContext.
       request.contextPath}/resources/css/file.css"
       />
    <title>first jsp called from controller</title>
  </head>
  <body>
    <h1>first jsp called from controller</h1>
    <p class=first> Je m'appelle ${ nom } </p>
  </body>
</html>
```

Le modèle

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

- Aller sur le lien `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)

Le modèle

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

- Aller sur le lien `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)

Créer une base de données `MyBase`

Le modèle

Modèle : accès et traitement de données

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

Le modèle

Étapes à suivre

- Tout d'abord, ajouter dans `pom.xml` les dépendances nécessaires relatives à `Mysql`, `JPA` et `Hibernate`
- 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

Le modèle

Les dépendances à ajouter dans `pom.xml`

```
<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.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.3.6.Final</version>
</dependency>
```

Le modèle

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>
```

Le modèle

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.jdbc.Driver"
     />
  <beans:property name="url" value="jdbc:mysql://localhost:3306/myBase"
     />
  <beans:property name="username" value="root" />
  <beans:property name="password" value="root" />
</beans:bean>
```

Le modèle

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.
         MySQL5Dialect</beans:prop>
    </beans:props>
  </beans:property>
</beans:bean>
```

Le modèle

Et un `bean` pour les transactions

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

Le modèle

Et un `bean` pour les transactions

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

Indiquons maintenant l'emplacement de nos repositories

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

Le modèle

Et un `bean` pour les transactions

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

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
```

L'entité `Personne`

```
package org.eclipse.FirstSpringMvc
   .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 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) 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="addPerson" 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="/showAll")
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` **:**

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ page import="java.util.List" %>
<%@ page import="org.eclipse.FirstSpringMvc.model.Personne"%>
<html>
  <head>
    <title>Result page</title>
  </head>
  <body>
    <%
      List <Personne> al = (List <Personne>) request.
        getAttribute("tab");
      for(Personne p: al){
        out.print("<br> " + p.getNom() + " " + p.getPrenom());
      }
    %>
  </body>
</html>
```

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

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`
- ...

Pour plus de détails : `https://docs.spring.io/spring-data/jpa/docs/current/reference/html/`

Le modèle

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

```
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> {
  @Query("select p from Personne p where p.nom = ?1"
     )
  Personne findByNom(String nom);
}
```

Configuration avec les classes

Idée

- 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`
- Supprimer le répertoire `spring` situé dans `WEB-INF`
- supprimer le contenu du `web.xml` (garder seulement les balise `web-app`)

Créons la classe `ApplicationConfig` dans `org.eclipse.FirstSpringMVC.configuration`

```
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
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;
@Configuration
@ComponentScan("org.eclipse.FirstSpringMvc.controller")
@EnableJpaRepositories("org.eclipse.FirstSpringMvc.dao")
@EnableTransactionManagement
public class ApplicationConfig {
  @Bean
  public InternalResourceViewResolver setup() {
    InternalResourceViewResolver viewResolver = new
       InternalResourceViewResolver();
    viewResolver.setPrefix("/WEB-INF/views/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
  }
```

La classe `ApplicationConfig` (suite)

```
@Bean
public DataSource dataSource() {
  DriverManagerDataSource dataSource = new DriverManagerDataSource();
  dataSource.setDriverClassName("com.mysql.jdbc.Driver");
  dataSource.setUrl("jdbc:mysql://localhost:3306/myBase");
  dataSource.setUsername("root");
  dataSource.setPassword("root");
  return dataSource;
}

@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory()
   {
  HibernateJpaVendorAdapter vendorAdapter = new
     HibernateJpaVendorAdapter();
  vendorAdapter.setGenerateDdl(true);
  vendorAdapter.setShowSql(true);
  LocalContainerEntityManagerFactoryBean factory = new
     LocalContainerEntityManagerFactoryBean();
  factory.setJpaVendorAdapter(vendorAdapter);
  factory.setPackagesToScan("org.eclipse.FirstSpringMvc.model");
  factory.setDataSource(dataSource());
  return factory;
}
```

Configuration avec les classes

La classe `ApplicationConfig` (suite)

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

  @Override
  public void addResourceHandlers(ResourceHandlerRegistry
     registry) {
    registry.addResourceHandler("/resources/**")
            .addResourceLocations("/resources/");
  }
}
```

Configuration avec les classes

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 avec les classes

La classe `MyWebInitializer`

```
package org.eclipse.FirstSpringMvc.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 avec les classes

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>
```

N'oublions pas de supprimer le répertoire `spring` situé dans `WEB-INF`

Configuration avec les classes

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>
```

N'oublions pas de supprimer le répertoire `spring` situé dans `WEB-INF`

Relancer le projet

Restructuration des classes de configuration

La classe `ApplicationConfig` contient beaucoup de code, solution ?

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

Restructuration des classes de configuration

La classe `MvcConfig`

```
package org.eclipse.FirstSpringMvc.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;
  }
}
```

Restructuration des classes de configuration

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{

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

Restructuration des classes de configuration

La classe `ApplicationConfig` devient ainsi :

```
package org.eclipse.FirstSpringMvc.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.transaction.annotation.
   EnableTransactionManagement;

@Configuration
@ComponentScan(basePackages = "org.eclipse.FirstSpringMvc.controller")
@EnableJpaRepositories("org.eclipse.FirstSpringMvc.dao")
@EnableTransactionManagement
public class ApplicationConfig {
  @Bean
  public DataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/myBase");
    dataSource.setUsername("root");
    dataSource.setPassword("root");
    return dataSource;
  }
```

Restructuration des classes de configuration

La classe `ApplicationConfig` (suite)

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

Restructuration des classes de configuration

La classe `MyWebInitializer`

```
package org.eclipse.FirstSpringMvc.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 [] {"/"};
  }
}
```