

Spring MVC

Cedric Dumoulin

Plan

- Bibliographie
- Qu'est-ce que Spring et Spring MVC ?
- Spring container
- Spring MVC

Bibliographie

- Spring Framework
 - <http://docs.spring.io/spring/docs/3.2.5.RELEASE/spring-framework-reference/htmlsingle/#overview>
- **Designing and Implementing a Web Application with Spring**
 - <http://spring.io/guides/tutorials/web/>

Bibliographie

- **Spring IO**
 - <http://spring.io/>
- **Developing a Spring Framework MVC application step-by-step (2.5)**
 - <http://docs.spring.io/docs/Spring-MVC-step-by-step/>
- **Spring MVC Framework Tutorial**
 - http://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm
- **Wikipedia**
 - http://en.wikipedia.org/wiki/Spring_Framework
- **Quick start**
 - <http://projects.spring.io/spring-framework/#quick-start>

Bibliographie

- Spring 3.x tutorials
 - <http://www.roseindia.net/spring/spring3/index.shtml>
 - <http://yannart.developpez.com/java/spring/tutoriel/>
 - <http://www.theserverside.com/tutorial/Spring-30-Tutorial-Setting-Up-Configuring-The-Environment>

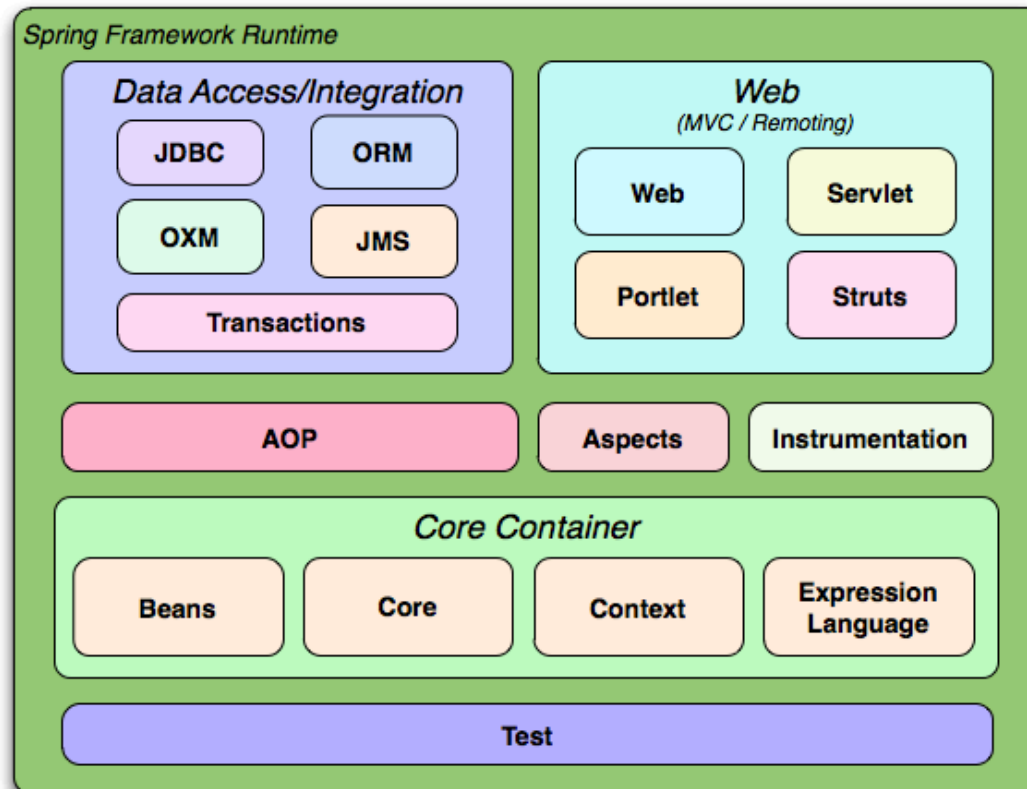
Guides

- Accessing Data with JPA
 - <http://spring.io/guides/gs/accessing-data-jpa/>
- **Designing and Implementing a Web Application with Spring**
 - <http://spring.io/guides/tutorials/web/>

Qu'est-ce que Spring
et Spring MVC ?

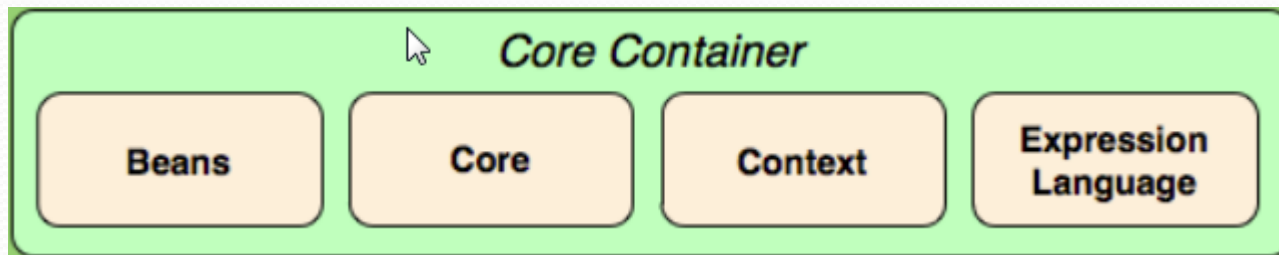
Qu'est-ce que Spring

- Un Framework
 - Constitué de caractéristiques organisées en 20 modules



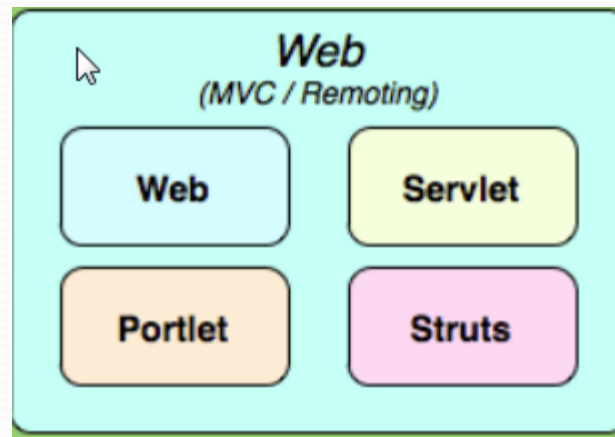
Core Container

- Fournit les mécanismes :
 - d'inversion de contrôle
 - D'injection
 - De fabrique d'objets (BeanFactory)



Web

- Web
 - Fournit les fonctionnalité web de base
- Servlet
 - Fournit Spring MVC
 - Permet de séparer le model Domaine et les pages web
- Struts
 - Integration Struts 1. Deprecated (utiliser Struts-Spring integration)
- Portlet
 - Pour un environnement avec des Portlet

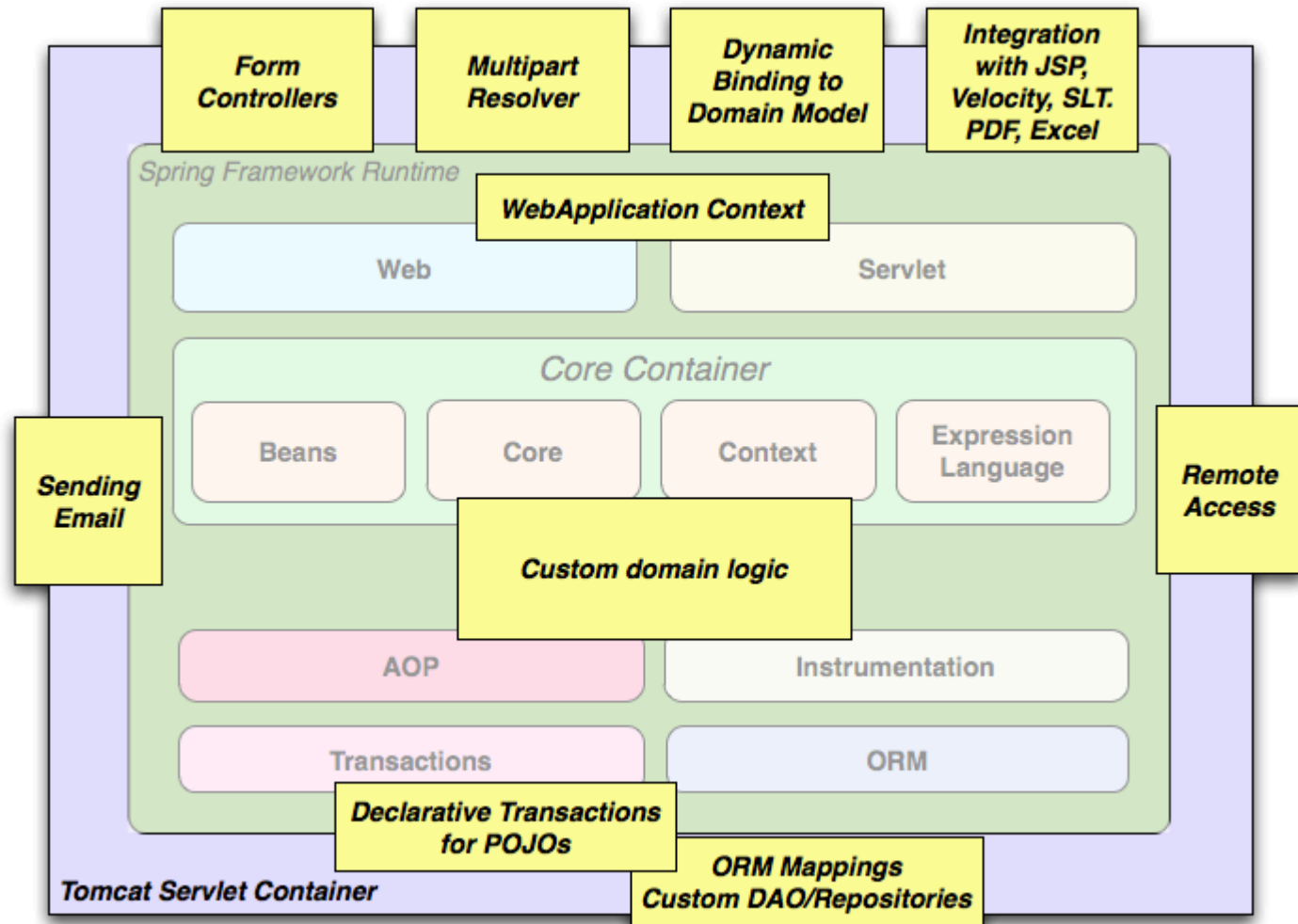


Test

- Pour les test avec Junit et TestNG

Test

Scénarios d'utilisation



Scénarios d'utilisation

- Custom business logic
 - can be implemented with simple POJOs
 - and managed by Spring's IoC container
- Transactions
 - Managed by container (like in EJB container)
- Additional services
 - email and
 - validation
 - ORM support
 - integrated with JPA, Hibernate, JDO and iBatis
 - Form controllers

Obtenir les librairies

- Maven
- SPS - Spring Tool Suite
 - Permet de créer un projet pré-initialisé

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>3.0.0.RELEASE</version>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

```
<repositories>
  <repository>
    <id>com.springsource.repository.maven.release</id>
    <url>http://repo.springsource.org/release/</url>
    <snapshots><enabled>false</enabled></snapshots>
  </repository>
</repositories>
```

Spring container



Le cœur de l'environnement Spring est un « conteneur léger »

Un conteneur léger sert à contenir un ensemble d'objets instanciés et initialisés, formant un contexte initial (ou une hiérarchie de contextes) pour une application.

Ce contexte initial est souvent construit à partir d'une description externe (xml) décrivant les objets à créer, les valeurs initiales et les dépendances entre objets.

Les dépendances (liens) entre objets sont automatiquement créées à partir de la description (on parle d'injection de dépendances) et non par les objets eux-mêmes par programmation.

C'est le Design Pattern de l'Inversion du Contrôle : **IoC**

Exemple simplifié:

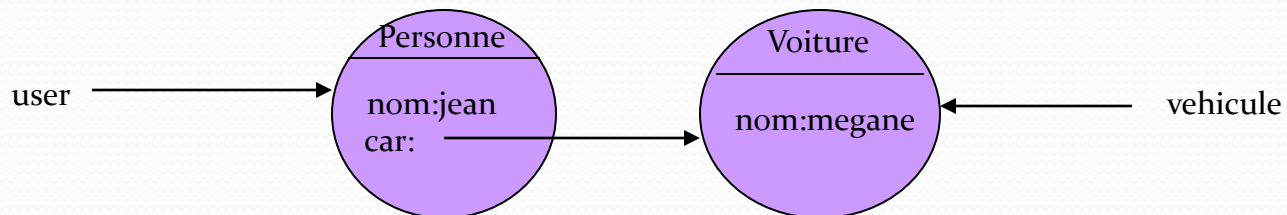
Avec les classes:

```
class Personne { String nom; Voiture car; }  
class Voiture {String nom;}
```

et la description de contexte Spring:

```
<beans>  
  <bean id=" user " class=" Personne ">  
    <property name=" nom " value=" jean "/>  
    <property name=" car " ref=" vehicule "/>  
  </bean>  
  <bean id=" vehicule " class=" Voiture ">  
    <property name=" nom " value=" megane "/>  
  </bean>  
</beans>
```

Le contexte initial de l'application dans le conteneur SPRING sera:



SpringMVC est un framework de présentation, pour application WEB, suivant le modèle MVC, et fondé sur le conteneur léger de SPRING

Dans le cas de SpringMVC le conteneur va servir à créer:

- Le contexte de l'application Web
- Les objets traitant les requêtes (Controller)
- Les objets créant les pages HTML (View)
- Les objets données des formulaires (Command)
- Les liens avec les couches métiers et BD
- Et pleins d'autres
 - Le mapping des URL vers les contrôleurs
 - Le mapping des vues , etc.

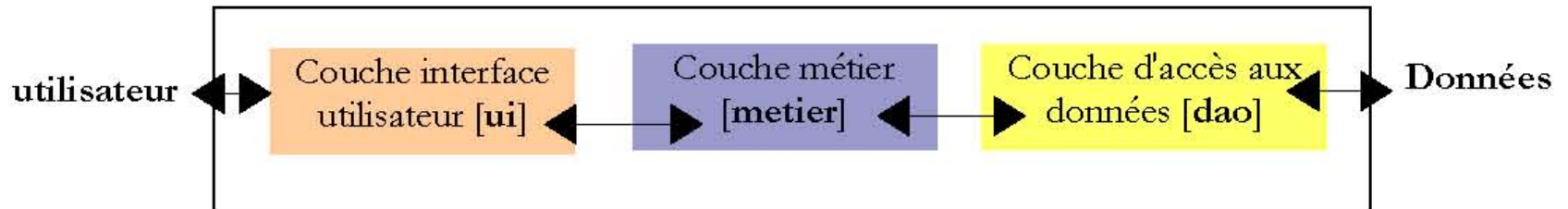
L'inversion du contrôle permet ensuite de changer le comportement de l'application, en modifiant la description xml du conteneur, sans changer les éléments programmés!

Web MVC Framework

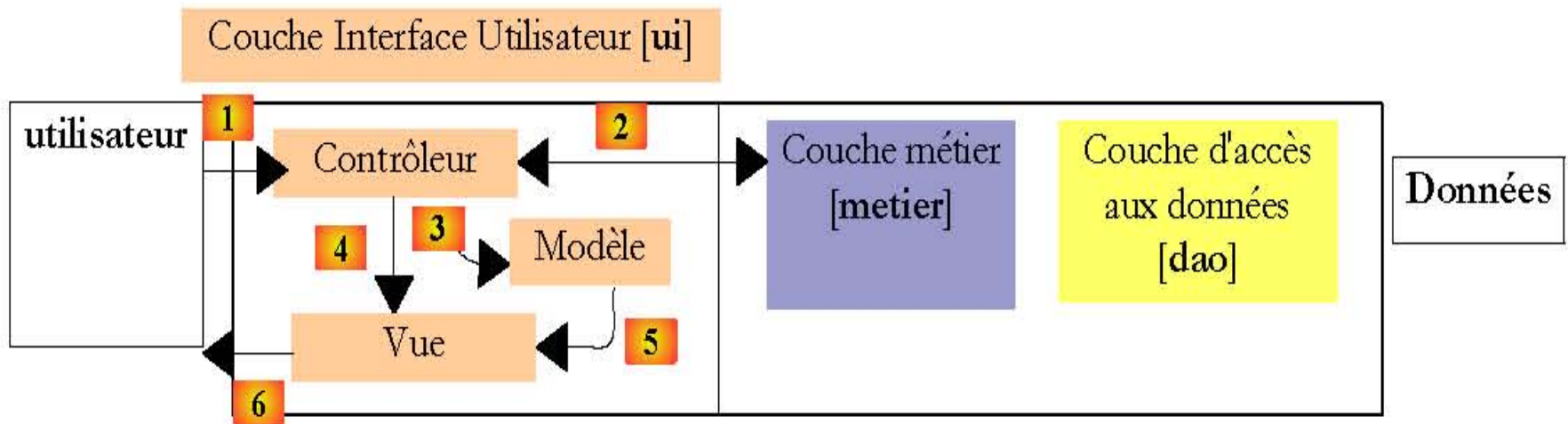
Principes de base

Retour sur le modèle MVC

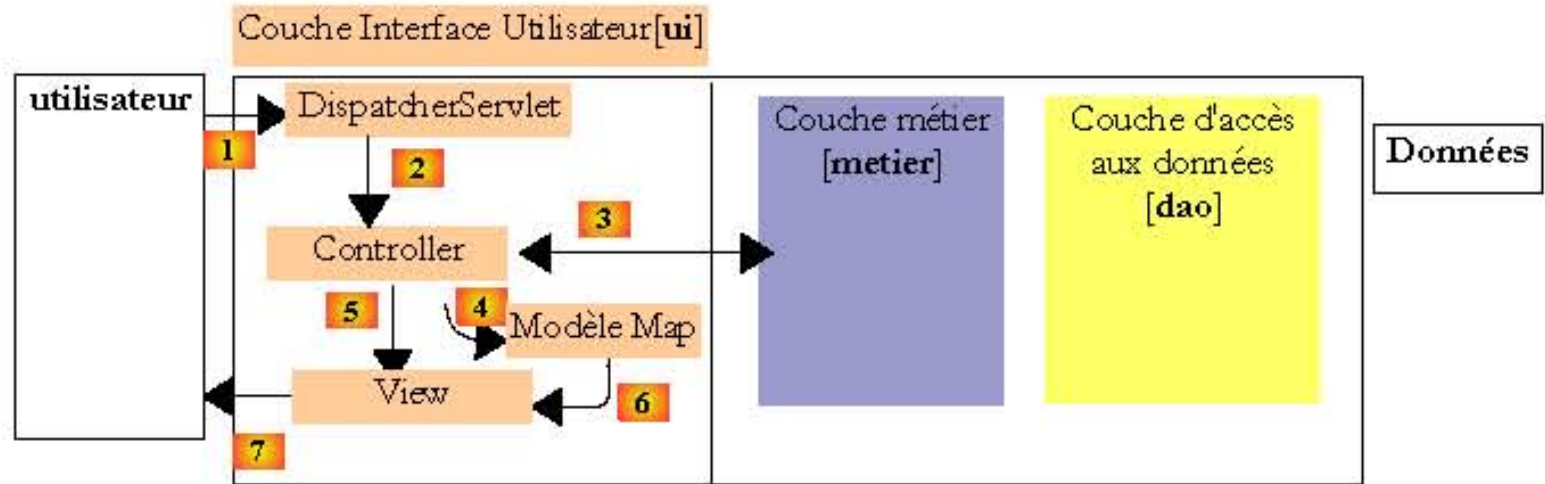
Une application 3tier classique:



Une application 3tier avec MVC:



La vision de SpringMVC



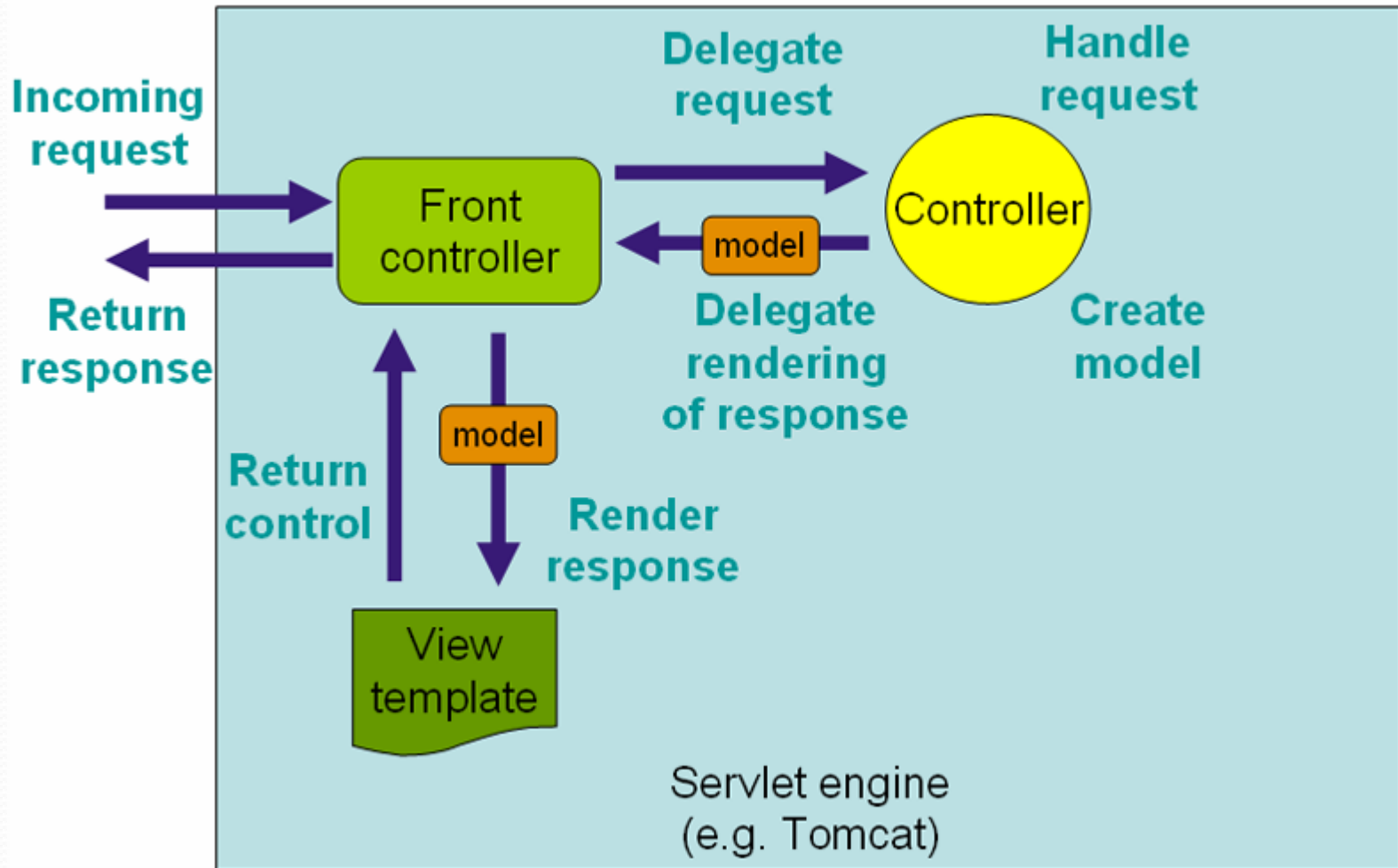
La `org.springframework.web.servlet.DispatcherServlet` est le point d'entrée générique qui délègue les requêtes à des **Controller**

Un `org.springframework.web.servlet.mvc.Controller` prend en charge une requête, et utilise la couche métier pour y répondre.

Un **Controller** fabrique un modèle sous la forme d'une `java.util.Map` contenant les éléments de la réponse.

Un **Controller** choisit une `org.springframework.web.servlet.View` qui sera paramétrée par la **Map** pour donner la page qui sera affichée.

Traitement type d'une requête



Pour faire fonctionner Spring MVC, il faut :

- Front Controller
 - ➔ servlet **DispatcherServlet**
 - Déclaration dans web.xml
- Controllers
 - Ce sont des POJO annotés @Controller
- Des vues
 - Choix possible de la technologie
 - Ex: jsp
- Un mapping request ➔ Controller
- Un mapping nom_logique_de_vue => implementation_de_la_vue
- Des objets métiers
 - Objet Spring
 - Ou Objets JEE

Web MVC Framework Configuration de base

Le « Front Controller »

- Il est chargé de traiter les requêtes de l'extérieur
 - Il analyse l'url, et appelle le contrôleur correspondant
 - Le contrôleur renvoie un modèle et le nom logique de la page servant à construire la page
 - Le front contrôleur appelle la page demandée
 - La page construit la réponse
 - Le front controlleur renvoie la réponse

Implémentation du Front Controller

- C'est un servlet
- Déclaré dans META-INF/web.xml

```
<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>
```

Le servlet

Le fichier de
configuration de
Spring

Le nom et les urls
traitées par le servlet

La configuration de Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans
  <!-- DispatcherServlet Context:
    defines this servlet's request-processing infrastructure -->

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

  <!-- 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="ipint13.spring.tuto" />
</beans:beans>
```

Indique
d'interpréter les
annotations

Spécifie le mapping
nom logique =>
vues

Chemin des classes
annotées à scanner

- Fichier [servletName]-context.xml

Le contrôleur

Controller simple

@Controller

public class GreetingController {

@RequestMapping("/greeting")

public String greeting(

@RequestParam(value="name", required=false, defaultValue="World") String name,
Model model) {

model.addAttribute("name", name);

return "greeting";

}

}

Déclare un contrôleur

L'url associée a ce ctrl

Mapping paramètre =>
argument

Le model qui sera passé à la
page

Un attribut qui sera passé à la
page

Nom logique de la page

Controller

```
/**
 * Simply selects the home view to render by returning its name.
 */
```

Spécifie le type de
requete

```
@RequestMapping(value = "/", method = RequestMethod.GET)
public String home(Locale locale, Model model) {
    logger.info("Welcome home! The client locale is {}.", locale);
```

Demande l'objet Locale

```
    Date date = new Date();
    DateFormat dateFormat =
        DateFormat.getDateInstance(DateFormat.LONG, DateFormat.LONG, locale);

    String formattedDate = dateFormat.format(date);

    model.addAttribute("serverTime", formattedDate );

    return "home";
}
```

Controller

- Peut contenir plusieurs méthodes avec différentes URL
- Les paramètres des méthodes sont « libre »
 - C'est le type ou l'annotation qui est détecté
- L'annotation `@RequestMapping` peut se faire sur la classe
 - Toutes les URL de méthodes sont alors relatives à l'URL de la classe.

Les vues

Technologies supportées

- Spring permet différente technologie pour les vues:
 - Struts, Tiles
 - Tapestry
 - JSP

Spring et JSP

- JSF sert de template à la réponse renvoyé par le front controlleur
- Les données viennent des modèles renvoyé par le controlleur

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
<head>
    <title>Getting Started: Serving Web Content</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    <p/>
    Hello again  ${name} !
</body>
</html>
```

Remplacé par la valeur
trouvé dans le modèle

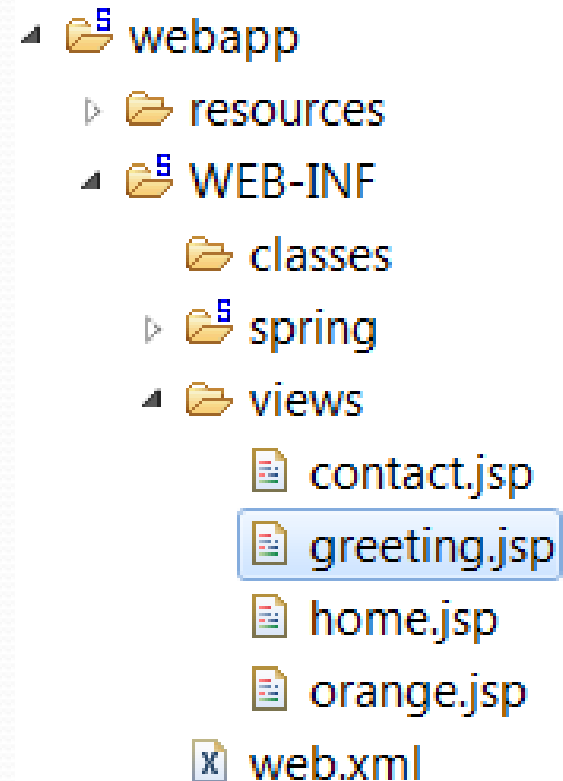
Mapping nom logique => vues

- Les vues sont séparées des contrôleurs
- Un controleur specifie la vue qui doit être utilisé
 - Il peut envoyer une vue differente en fonction du resultat (ok, cancel, ...)
- Un mapper se charge de mapper les noms logique vers les noms physique
 - Déclaré dans le fichier de config [nom]_config.xml

```
<!-- 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>
```

Où placer les vues ?

- Dans un répertoire sous WEB-INF



Les formulaires

L'entête

- views/contact.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring 3 MVC Series - Contact Manager</title>
</head>
<body>
<h2>Contact Manager</h2>
...
```

Le form

L'url où la requête est envoyée

```
<form:form method="post" action="addContact.html">

  <table>
  <tr>
    <td><form:label path="firstname">First Name</form:Label></td>
    <td><form:input path="firstname" /></td>
  </tr>
  <tr>
    <td><form:label path="lastname">Last Name</form:Label></td>
    <td><form:input path="lastname" /></td>
  </tr>
  <tr>
    <td><form:label path="lastname">Email</form:Label></td>
    <td><form:input path="email" /></td>
  </tr>
  <tr>
    <td><form:label path="lastname">Telephone</form:Label></td>
    <td><form:input path="telephone" /></td>
  </tr>
  <tr>
    <td colspan="2">
      <input type="submit" value="Add Contact"/>
    </td>
  </tr>
</table>

</form:form>
</body>
</html>
```

Le nom d'un attribut dans l'objet 'command'

Le contrôleur

```
@Controller
@SessionAttributes
public class ContactController {

    @RequestMapping(value = "/addContact.html", method = RequestMethod.POST)
    public String addContact(@ModelAttribute("contact")
        Contact contact, BindingResult result, Model model) {

        System.out.println("First Name:" + contact.getFirstname() +
            "Last Name:" + contact.getLastname());
        contact.setEmail(contact.getFirstname() + "." + contact.getLastname());
        model.addAttribute("command", contact);

        return "contact";
    }

    // return "redirect:contacts.html";

    @RequestMapping("/contacts.html")
    public ModelAndView showContacts() {

        return new ModelAndView("contact", "command", new Contact("name", "lastname"));
    }
}
```

L'url où la requête est
envoyée

L'url où la requête est
envoyée

Pour connaître les
erreurs lors du Bind

On peut demander des
redirection vers une URL

L'objet passé entre le ctrl et le form

- Classe POJO

```
public class Contact {  
    private String firstname;  
    private String lastname;  
    private String email;  
    private String telephone;  
  
    // Getter and setters ...  
}
```

@ModelAttribute sur les paramètres

- An @ModelAttribute on a method argument indicates the argument should be retrieved from the model.
- If not present in the model, the argument should be instantiated first and then added to the model.
- Once present in the model, the argument's fields should be populated from all request parameters that have matching names.
- This is known as data binding in Spring MVC, a very useful mechanism that saves you from having to parse each form field individually.

Comment initialiser un objet dans le model

- Utiliser @ModelAttribute sur la méthode

```
/**
 * Initialize attribute in the model.
 * Use default name 'contact' (name of the returned type).
 * @return
 */
@ModelAttribute
public Contact initContact() {
    Contact contact = new Contact();
    contact.setFirstname("set firstname");
    contact.setLastname("set lastname");

    return contact;
}
```

EJB integration

- 
- Doc chap 22

Accéder à un objet JNDI ou EJB

- L'objet doit exister dans un autre container
- On injecte le bean
- ex: injecter un bean dans Spring ☺

membres-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-2.5.xsd">

  <bean id="catalogController" class="ipint.mysite.CatalogController">
    <property name="catalog" ref="catalogId"/>
  </bean>

  <jee:jndi-lookup id="catalogId" jndi-name="CatalogBean/remote"
    cache="true" />
</beans>
```

injection

recherche du bean – lien nom logique <-> nom JNDI

Accéder à un objet JNDI ou EJB

- `<jee:jndi-lookup>`
 - Accès par JNDI
- `<jee:local-slsb>`
 - Accès à un bean local
- `<jee:remote-slsb>`
 - Accès à un bean distant

```
<jee:local-slsb id="myComponent" jndi-name="ejb/myBean"
business-interface="com.mycom.MyComponent"/>
<bean id="myController" class="com.mycom.myController">
<property name="myComponent" ref="myComponent"/>
</bean>
```

Accéder à un objet JNDI ou EJB

Autre methode

- Permet d'utiliser le nom jndi directement dans les annotations
- A tester

```
<bean  
class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor">  
<property name="alwaysUseJndiLookup" value="true" />  
</bean>
```


Injection d'objet

- Spring fournit la notion de composant
- On peut injecter des composants dans d'autre composant
- Même principe que JEE
- @Autowired, @Inject, @Resource, and @Value

```
public class MovieRecommender {  
  
    @Autowired  
    private ApplicationContext context;  
  
    public MovieRecommender() {  
    }  
  
    // ...  
}
```

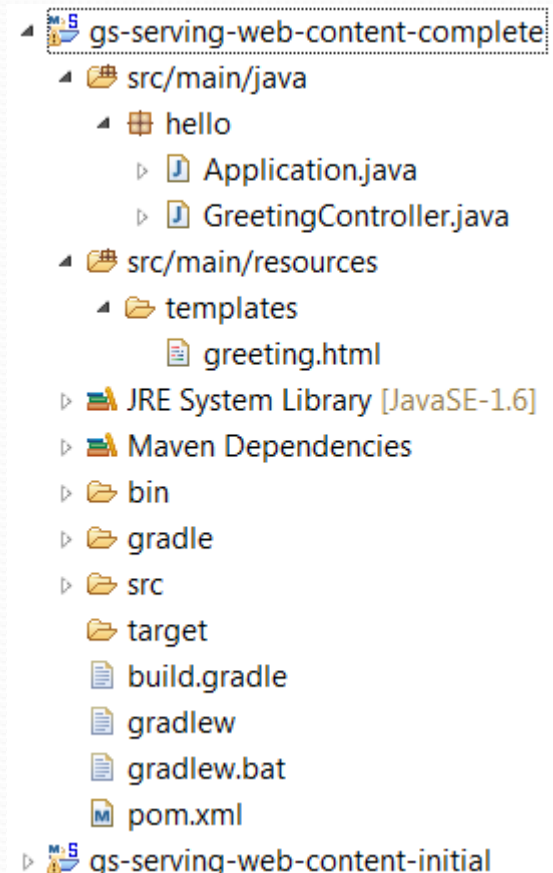
Spring Tool Suite™ Downloads

SPS - Spring Tool Suite

- Plugins Eclipse pour le développement avec Spring
- <http://spring.io/tools/sts/all>
 - all-in-one distribution
 - Update zip
 - Update site (pour ajouter dans une application existante).

Installer un exemple avec STS

- File -> New -> Import Spring Getting Started Content
- Installe aussi les jars necessaire à la compilation et l'exécution
- Ce sert de Maven
 - ➔ les jar sont dans .m2
- <http://spring.io/guides/gs/sts/>



Exécuter un exemple

- Construire le jar et le lancer :

- `mvn clean package && java -jar target/gs-serving-web-content-o.1.o.jar`

- Notes:

- Ce n'est pas un war ☹
 - Le jar contient un serveur embarqué
 - Projet 'boot'
 - Certain fichiers de base manquent (web.xml, ...)
 - Ils sont déduits des classes et annotations
 - Approche non recommandé pour la production (et même pour le développement).

Convertir un projet 'boot' en war

- <http://spring.io/guides/gs/convert-jar-to-war/>
- Modifier le build.gradle
 - Ajouter `apply plugin: 'war'`
 - Modifier jar en war
- Ajouter la classe
 - HelloWorldXml extends SpringBootServletInitializer
- `./gradlew clean build`
 - Vous obtenez un war
- Déployer le war dans votre serveur

```
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'spring-boot'
apply plugin: 'war'

war {
    baseName =
        'gs-serving-web-content'
    version = '0.1.0'
}
```

hello/HelloWebXml.java

```
package hello;

import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.SpringBootServletInitializer;

public class HelloWebXml extends SpringBootServletInitializer {

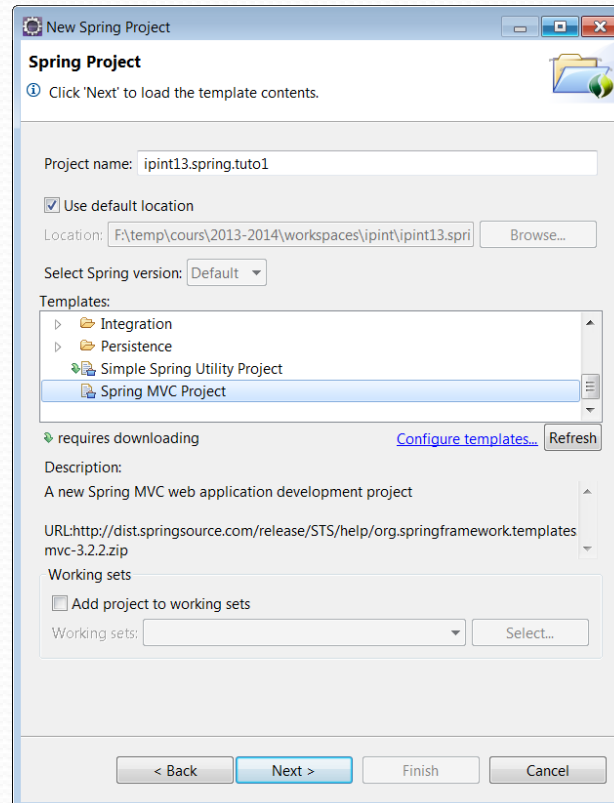
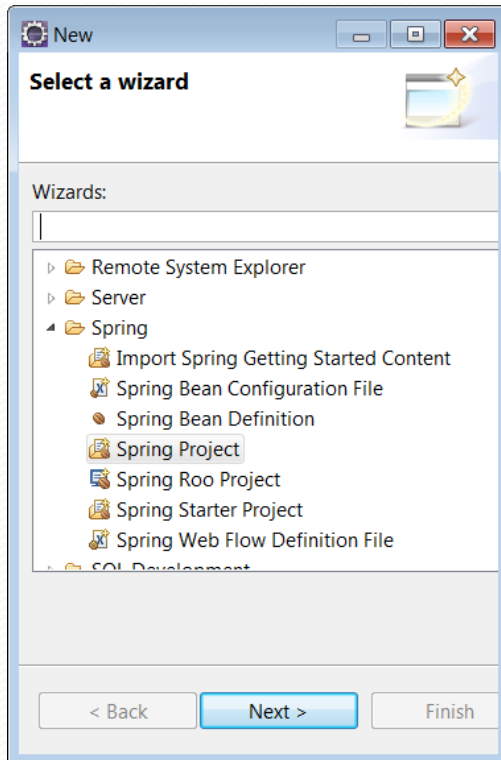
    @Override
    protected SpringApplicationBuilder
        configure(SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }

}
```


Créer un projet avec Spring et SPS

Créer un nouveau projet

- New ->Other->Spring->Spring Project
- Choisir Spring MVC Project
- Utiliser le même nom pour le nom de projet et le 'top level package'



Nouveau projet

- Ce projet peut être compilé sous Eclipse.
- Il peut être déployé dans Glassfish

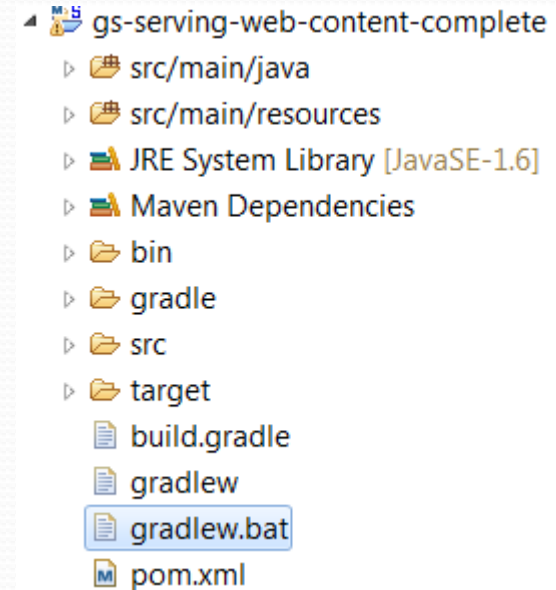


Spring MVC

Gradle

Gradle

- Outils similaire a Maven
- Chaque exemple de Spring vient avec un bin permettant de lancer gradle
 - ➔ ouvrir une console
 - `./gradlew clean build`
 - Charge les jar gradle
 - Execute `build.gradle`



Validation