

Spring MVC

Antonio Espín Herranz

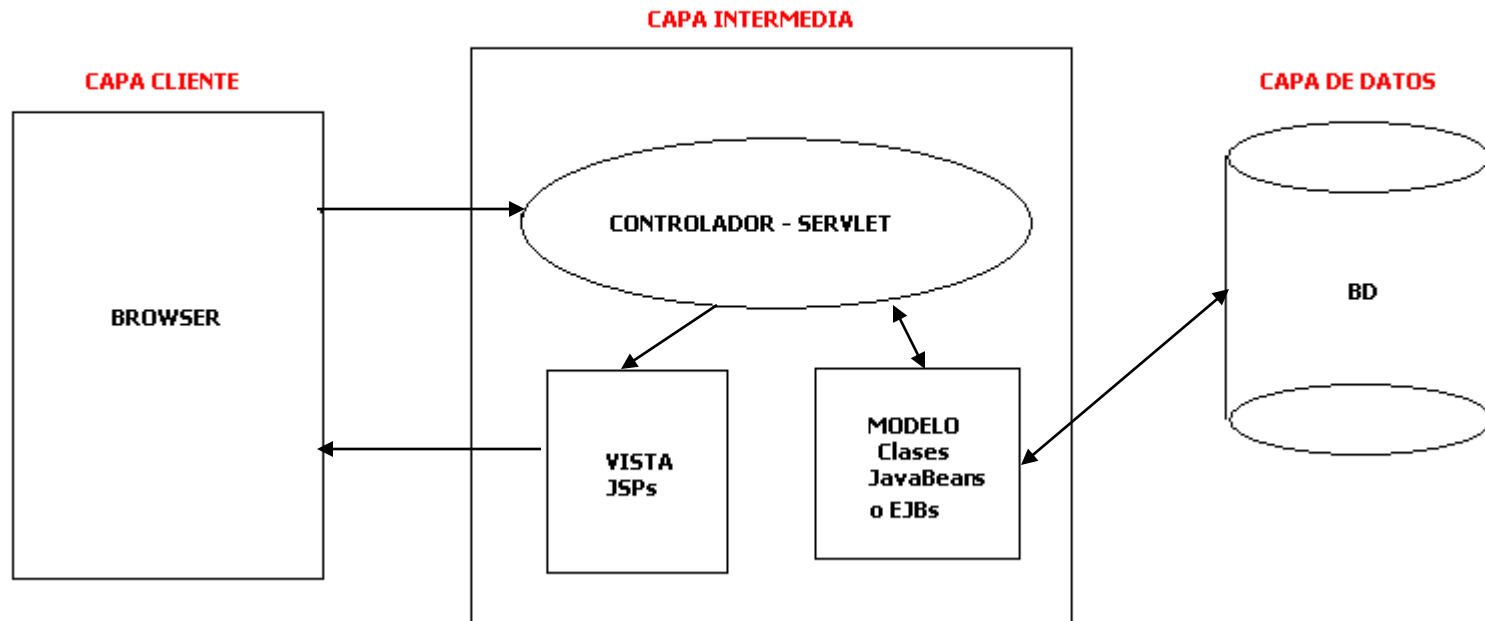
Spring MVC

- Capa Web (componentes MVC).
- Funcionamiento.
- DispatcherServlet.
- Configuración MVC.
- Anotaciones.
- Ficheros de configuración.
- Controladores.
- Modelo.
- Vistas – etiquetas.
- Objetos de respaldo.
- Validadores.
- i18n.
- Tiles.
- FileUpload.

Capa Web

- Dentro de Spring tenemos soporte para aplicaciones Web, disponemos de dos frameWork:
 - Web MVC frameWork: Basado en Servlets.
 - Porlet MVC frameWork: Basado en Portlets.
- Además Spring se puede integrar con otros frameworks que no pertenecen al proyecto de Spring, como Struts y JSF (ambos basados en el patrón MVC).

Esquema



El Controlador

- Es el “cerebro” de la Aplicación.
 - En él se centran todas las peticiones por parte de la capa Cliente.
 - Sabe quien tiene que hacer cada cosa.
 - Se encarga de redirigir el flujo de las peticiones.
 - Si tiene que mostrar datos hará uso de la Vista (las páginas JSP).
 - Si tiene que extraer datos de la BD, llamará a una clase del modelo.

El Controlador

- Si la capa cliente solicita unos determinados datos:
 - El Controlador solicita los datos necesarios al modelo (es el encargado de interactuar con la BD).
 - Una vez que tiene los datos, se los envía a la vista para que se encargue de mostrárselos al Cliente.

¿Por qué un Servlet?

- Es una clase Java, es mas útil para gestionar el flujo de la programación, tiene estructura en el código, y es incómodo para mostrar datos.
- La vista utilizaremos páginas JSP mas parecidas a una página de HTML en la que es mas fácil dar formato a los datos que queremos mostrar.
- Se pueden separar mas los perfiles, programadores pueden desarrollar el Servlet y el modelo.
- Diseñadores se pueden centrar mas en la Vista.

El Controlador

- La centralización del flujo de peticiones en el Servlet, proporciona una serie de ventajas:
 - Desarrollo mas limpio y sencillo.
 - Facilita el posterior mantenimiento de la aplicación, es mas escalable.
 - Facilita la detección de errores.
- Puede darse el caso de que el Servlet se apoye en otros Servlets auxiliares, el Servlet principal recibe las peticiones y las redirige a otros Servlet auxiliares.

La Vista

- Se encarga de generar las respuestas, generalmente serán código xHTML que serán enviadas al Navegador.
- Si la respuesta a generar no es fija y procede de los datos obtenidos del Controlador, está se tendrá que generar de forma dinámica, aquí es donde entran las **páginas JSPs**.
- Si la información a devolver fuera estática se podría implementar directamente con páginas XHTML.

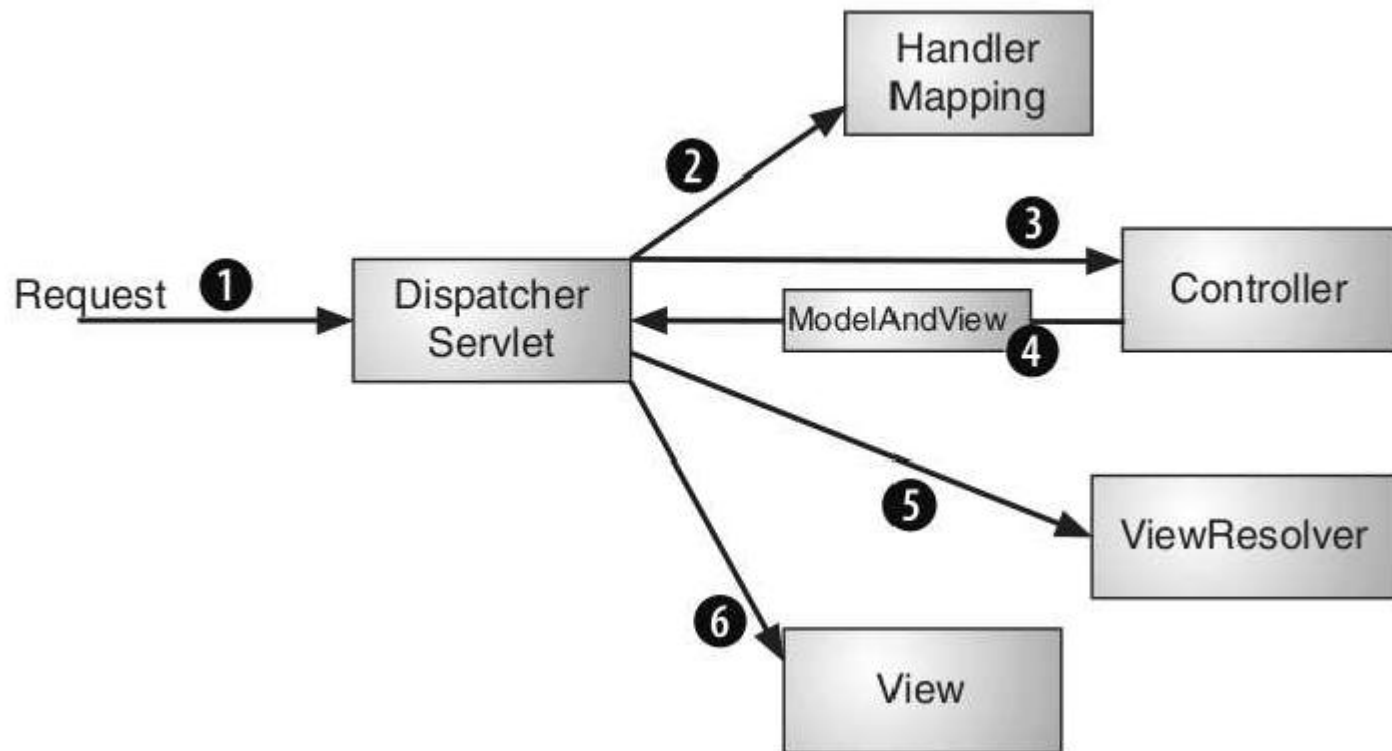
El Modelo

- Representa la lógica de Negocio de nuestra aplicación.
- Incluye el acceso a datos y su manipulación.
- El Modelo está representado por una serie de componentes independientes del Controlador y de la Vista, que permite la reutilización y desacoplamiento de las capas.
- Serán clases ayudante (JavaBeans) o EJBs o un patrón DAO de Spring.

Spring MVC

- Spring implementa su propio MVC.
- Para pintar la vista se apoya en JSLT (definición de etiquetas) y además proporciona una librería de etiquetas propias.
- Por lo tanto nuestro proyecto tiene que incluir los jar:
 jslt.jar
 standard.jar
- A parte los jar propios de Spring.

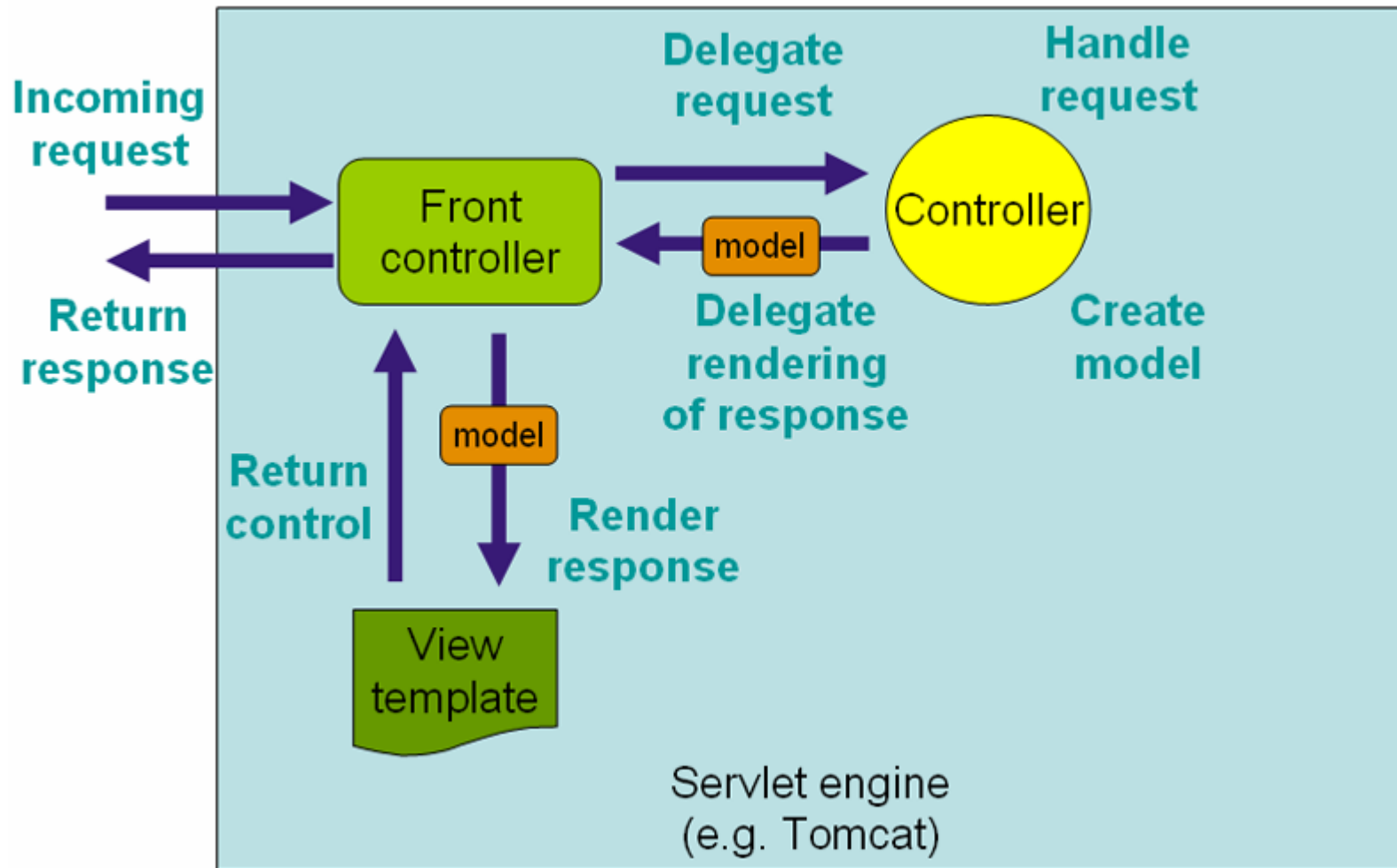
Spring MVC - Funcionamiento



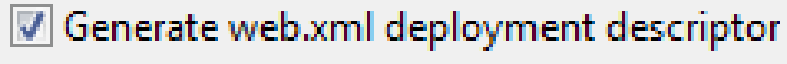
Spring MVC - Funcionamiento

- (1) Las peticiones pasan a través de un **Servlet** que actúa de **Controlador Frontal** representado por la clase de Spring → (*DispatcherServlet*).
- (2) El *DispatcherServlet* **consulta** a un **HandlerMapping** para decidir a qué controlador le pasa la petición.
 - Usa la URL de la solicitud para decidir.
- (3 y 4) El controlador procesa la petición, accede a la lógica de negocio y potencialmente obtiene resultados (modelo, un *Map*) *además selecciona la vista* para presentar el modelo.
 - Para que el controlador no esté acoplado a la vista, se devuelve un identificador lógico (nombre) de vista.
 - Y también empaqueta los datos del modelo que devuelve al *DispatcherServlet*.
- (5) El *DispatcherServlet* utiliza un *ViewResolver* para resolver el nombre en una vista concreta.
 - Normalmente un JSP, pero se soportan otros como:
 - XSLT, Tiles, ...
- (6) El *DispatcherServlet* utiliza la vista para mostrar el modelo al usuario.

DispatcherServlet



Spring MVC - DispatcherServlet

- Objeto de la clase ***DispatcherServlet*** del paquete ***org.springframework.web.servlet***
- **Recibe** peticiones y las **redirige** al Controlador que tiene que atenderlas (**podemos tener varios controladores**).
- **Configurar** el **/WEB-INF/web.xml** de la aplicación para que el *dispatcher reciba las peticiones*.
 - **OJO!** Si estamos trabajado con la **especificación 3.0 de Servlets** el fichero de configuración **web.xml** no se crea por defecto, hay que decirle a Eclipse que lo tiene que crear:
 - Botón derecho sobre el proyecto → Java EE Tools → Generate Deployment Descriptor Stub.
 - O indicarlo al crear el proyecto: 
- Además necesitamos **un fichero de configuración XML para el DispatcherServlet**, ubicado en:
/WEB-INF/nombre_dispatcher-servlet.xml

Configuración MVC

- Dentro del fichero **web.xml** hacemos referencia al DispatcherServlet:

- En la **declaración** del Servlet.

```
- <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>
```

- En el **mapeo** de servlet.

```
- <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

- Tener en cuenta que el nombre que le demos al servlet determinará el nombre del fichero de configuración XML.
 - En nuestro caso sería: **dispatcher-servlet.xml**
 - **REPRESENTA EL FICHERO DE CONFIGURACION.**

Configuración MVC

- El definir el **mapeo** del **DispatcherServlet** con la `/` indica que **todas las solicitudes** van a ser tratadas por este servlet, **incluyendo las peticiones de contenido estático**.
- Podemos utilizar un elemento nuevo de configuración.
 - El **espacio de nombres mvc**.



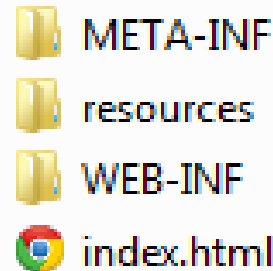
mvc - <http://www.springframework.org/schema/mvc>

- Disponemos del elemento: **<mvc:resources>** para indicar donde se encuentra todo el contenido de estático de la app.
- Este elemento se define dentro del fichero:
 - **dispatcher-servlet.xml**

Configuración MVC

```
<mvc:resources mapping="/resources/**"  
location="/resources/">
```

- Configura un gestor para proporcionar contenido estático.
- De la carpeta de **WebContent** de nuestra aplicación tendríamos:



- **Atributos:**

- **mapping:** indica que todas las rutas deben comenzar por **/resources** y que incluya todas las subrutas.
- **location:** indica la ubicación de los archivos. Esto implica que todas las: imágenes, javascript, css, etc. Irán en una carpeta con este nombre.

Configuración MVC: anotaciones

- Para empezar a definir **controladores** no necesitamos ninguna clase de Spring, tendremos **una clase POJO más unas anotaciones de Spring**.
- Para poder utilizar las anotaciones tenemos que definir dentro del fichero: **dispatcher-servlet.xml**
- El elemento: **<mvc:annotation-driven />**
 - **Compatible** con la **JSR-303** nos va a permitir también definir **validaciones**.

Definir Controladores

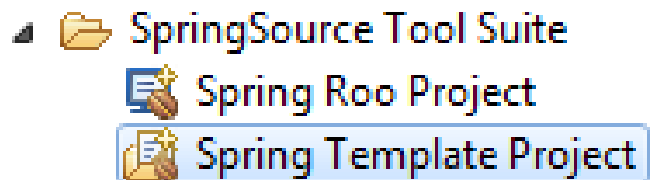
```
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public String helloWorld(Model model) {
        model.addAttribute("message", "Hello World!");
        return "helloWorld";
    }
}
```










- Uso de anotaciones:
 - La clase será anotada con **@Controller**.
 - Y la anotación **@RequestMapping** indica a la URL que responde el controlador.
 - El **método retorna el nombre lógico** de una vista: “helloWorld”, que mediante un **ViewResolver** se obtendrá la ruta completa.

Ejemplo

- Crear un proyecto en STS utilizando la plantilla para MVC.



Templates:

-  Simple Spring Batch Project
-  Simple Spring Hibernate Utility Project
-  Simple Spring JPA Utility Project
-  Simple Spring Utility Project
-  Spring Batch Admin Webapp
-  Spring Integration Project (Standalone) - File
-  Spring Integration Project (Standalone) - Simple
-  Spring Integration Project (War)
-  **Spring MVC Project**

El descriptor de despliegue: **web.xml (1)**

<!-- Declarar un Listener que se encargará de cargar el contexto de Spring -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!--Definir donde se ubica el contenedor de Spring, donde vamos a definir todos los beans-- >

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/root-context.xml</param-value>

</context-param>

El descriptor de despliegue: **web.xml (2)**

<!-- Declaración del DispatcherServlet - ->

<servlet>

<servlet-name>appServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!-- Si queremos asignarle otro nombre al fichero de configuración del Servlet se le indica como un parámetro de inicialización del Servlet, indicando nombre y ubicación -->

<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>

<!-- Mapeo del Servlet atiende a TODAS las peticiones - ->

<servlet-mapping>

<servlet-name>appServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

Fichero de configuración del Servlet: **servlet-context.xml** (1)

```
<?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-3.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">
```

<! - - En este caso se define por defecto el mvc (no lleva prefijo en el xmlns) - - >

Fichero de configuración del Servlet:

servlet-context.xml (2)

<!--Habilitar anotaciones @Controller -->

`<annotation-driven />` **<!-- Este elemento pertenece a <mvc: -->**

<!-- ¿donde ubicamos todo el contenido estático de la aplicación: js, css, img, etc? -->

<!-- Este elemento pertenece a <mvc: -->

`<resources mapping="/resources/**" location="/resources/" />`

<!-- Definir un ViewResolver: a cada vista que indiquemos mediante un nombre lógico la concatena el prefijo y el sufijo que aquí aparecen: -->

`<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>`

<!--Le indicamos un paquete para que busque todos los controladores anotados y creará automáticamente los beans -->

`<context:component-scan base-package="com.ejemplo.mi_mvc" />`

Fichero de configuración de Spring: **root-context.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"  
  xsi:schemaLocation="http://www.springframework.org/schema/beans  
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">  
  
  <!-- define recursos compartidos visibles a otros componentes de web -->  
  
</beans>
```

El controlador: HomeController.java

@Controller

public class HomeController {

// Si queremos volcar información a la consola del Server

private static final Logger logger = LoggerFactory.getLogger(HomeController.class);

@RequestMapping(value = "/", method = RequestMethod.GET)

public String home(Locale locale, Model model) {

logger.info("Welcome home! the client locale is "+ locale.toString());

Date date = new Date();

DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.LONG, DateFormat.LONG, locale);

String formattedDate = dateFormat.format(date);

// Carga en el modelo los parámetros que queremos enviar a la VISTA.

model.addAttribute("serverTime", formattedDate);

// Devuelve una cadena que identifica a la VISTA que queremos pintar.

return "home";

}

}

La vista: home.jsp

<!-- Referencia a jslt (etiquetas para pintar en la vista) - ->

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

```
<%@ page session="false" %>
```

```
<html>
```

```
<head>
```

```
<title>Home</title>
```

```
</head>
```

```
<body>
```

```
<h1>
```

```
Hello world!
```

```
</h1>
```

<!-- Recuperar del modelo, este atributo lo cargamos en el Controlador -- >

```
<P> The time on the server is ${serverTime}. </P>
```

```
</body>
```

```
</html>
```

Anotaciones en los Controladores

- En los controladores podemos encontrar las siguientes anotaciones:
- **@Controller**: La clase es un controlador. Se coloca por encima de la declaración de la clase.
- **@RequestMapping**: Por encima de la declaración del método o a nivel de clase.
 - Atributos:
 - **value**: La url a la que responde el método.
 - **method**: Puede tomar los valores:
 - **RequestMethod.GET** o **RequestMethod.POST**.

Anotaciones en los Controladores

- Ejemplos de **@RequestMapping**
 - Establecer la ruta base (cuando se declara a nivel de clase):

```
@Controller
```

```
@RequestMapping("/spitter") → establece una ruta base.
```

```
public class SpitterController {
```

```
    // Peticiones relativas a partir de /spitter
```

```
    // Atiende peticiones a: /spitter/spittles
```

```
    @RequestMapping(value="/spittles", method=RequestMethod.GET)
```

```
    public String metodo( .. ){}
```

Anotaciones en los Controladores

- Ejemplos de **@RequestMapping**

- También se pueden definir métodos que salten por GET / POST.

- Parámetros de la petición:

@Controller

@RequestMapping("/spitter") → **establece una ruta base.**

```
public class SpitterController {
```

```
    @RequestMapping(value="/spittles", method=RequestMethod.GET,  
        params="new")
```

```
    public String metodo() {...}
```

- Atenderá a las siguientes peticiones:

- <http://localhost:8080/app/spitter/spittles?new>

Anotaciones en los Controladores

- **@RequestMapping**(value = {"/", "/home"}, ...)
 - Si necesitamos que el método atienda a más de una URL.
- **@RequestParam**: Se coloca delante del parámetro del método. Para capturar parámetros que vengan de un formulario.
 - `public String metodo(@RequestParam("nombre") String nombre)`
 - **nombre** será el nombre de una caja de texto del html.

Anotaciones en los Controladores

- En el caso de **@RequestParam** podemos recoger parámetros de otros tipos y hace la conversión automática.
- Por ejemplo:
- ```
public String calcular(Model model,
 @RequestParam("a") double a,
 @RequestParam("b") double b,
 @RequestParam("c") double c) { ... }
```

# Acciones en los Controladores

- En el mismo controlador podemos procesar peticiones de tipo POST y tipo GET.
- Por ejemplo:

```
@Controller
@RequestMapping("/")
public class CustomerController{
 @RequestMapping(method = RequestMethod.POST)
 public String processSubmit(...) {
 // Procesa la info de un formulario ...
 }
 @RequestMapping(method = RequestMethod.GET)
 public String initForm(...){
 // Puede rellenar datos del formulario... Cargar colecciones para mostrar
 controles
 }
}
```

# Parámetros de los controladores

- Los métodos de los controladores no están ligados a ningún interface.
- Sólo se lanzan cuando coincide la cadena url.
- Por tanto pueden recibir o devolver una serie de parámetros variado.

# Parámetros de entrada en un método del Controller

- Parámetros nativos:
  - ServletRequest o HttpServletRequest.
  - ServletResponse o HttpServletResponse.
  - HttpSession.
- WebDataBinder.
- Locale.
- InputStream / OutputStream.
- @RequestParam.
- Map / Model.
- Errors / BindingResult.
- @Validated MiClase miBean
- @ModelAttribute("miBean") MiClase miBean

# Parámetros de salida en un método del Controller

- Con respecto al modelo:
  - ModelAndView.
  - Model.
  - Map
- Con respecto a la vista:
  - String
- Un objeto de una clase.
- void

# El modelo

- Los métodos que se definen dentro de los controladores pueden recibir los parámetros:
  - Model miModelo.
    - En este objeto vamos a cargar los resultados para que la vista los utilice.
    - Disponemos de estos métodos:

```
● addAllAttributes(Collection<?> arg0) : Model - Model
● addAllAttributes(Map<String,?> arg0) : Model - Model
● addAttribute(Object arg0) : Model - Model
● addAttribute(String arg0, Object arg1) : Model - Model
```

# El modelo

- Los datos que cargan los controladores en el modelo se pasan a la vista de forma automática y podemos acceder a ellos.
- Normalmente vamos a cargar un objeto, o un objeto etiquetado con una clave de tipo String.

# Vistas

- En las vistas basadas en JSP normalmente podemos encontrar referencias a las etiquetas de JSLT y específicas de Spring.
- `<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>`
- Por ejemplo, para mostrar un dato cargado en el modelo cuya clave es mensaje.
- `<c:out value="${mensaje}"></c:out>`



# Etiquetas Form de Spring

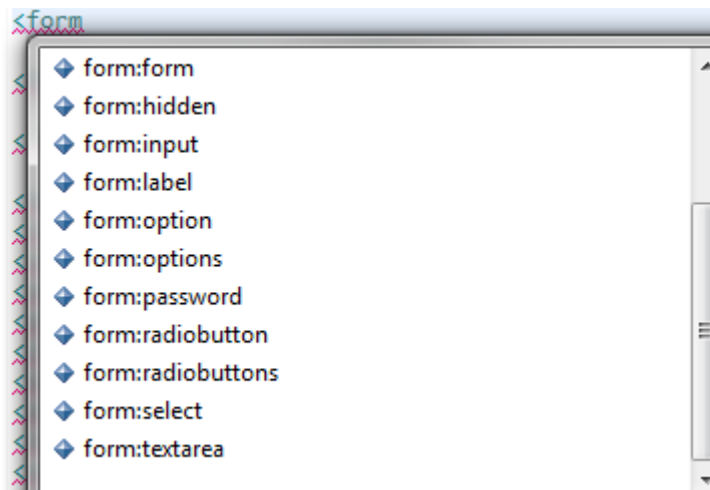
- Spring dispone de su propia librería de etiquetas para la gestión de formularios:

`<%@ taglib prefix="form"`

`uri="http://www.springframework.org/tags/form"%>`

- Para utilizarlas dentro de la vista en las JSP:

– `<form:`



# <form:form

- Representa la etiqueta form.
- Los atributos más importantes:
  - **method**: POST / GET.
  - **commandName**: Hace referencia al nombre del bean que se va a ligar con el form.
  - **action**: Destino, tiene que enganchar con algún RequestMapping de algún método del controller.
- Disponemos de las etiquetas habituales en los form de HTML.
- Con el atributo path hacemos referencia a un objeto de respaldo.
- Ejemplo: `<form:form method="POST" commandName="customer">`
  - **customer**: tiene que ser un bean de respaldo que el controlador lo ha tenido que cargar con anterioridad.

# Etiquetas I

- El atributo path hace referencia a una propiedad del bean de respaldo.
- **Importante: EL BEAN DE RESPALDO TIENE QUE CUMPLIR LAS CONVECIONES DE LOS JAVABEAN. CONSTRUCTOR POR DEFECTO / GET / SET.**
- Caja de Texto
  - `<form:input path="userName" />`
- Text Area:
  - `<form:textarea path="address" />`
- Password:
  - `<form:password path="password" />`
- CheckBox:
  - `<form:checkbox path="receiveNewsletter" />`

# Etiquetas II

- Varios CheckBox cuyos valores son una colección del objeto de respaldo:
  - `<form:checkboxes items="{webFrameworkList}" path="favFramework" />`
- Radio Buttons:
  - `<form:radiobutton path="sex" value="M"/>Male`
  - `<form:radiobutton path="sex" value="F"/>Female`
- Radio Buttons con una colección:
  - `<form:radiobuttons path="favNumber" items="{numberList}" />`

# Etiquetas III

- Desplegable:
  - `<form:select path="country">`
    - `<form:option value="NONE" label="--- Select ---"/>`
    - `<form:options items="${countryList}" />`
  - `</form:select>`
- Lista de selección con una colección:
  - `<form:select path="javaSkills" items="${javaSkillsList}" multiple="true" />`
- Campos ocultos:
  - `<form:hidden path="secretValue" />`
- Botón de Submit:
  - `<input type="submit" />`
  - `<form:button value="Enviar" name="enviar">Enviar</form:button>`

# Objetos de Respaldo

- Mediante el uso de anotaciones se pueden rellenar objetos a partir de las etiquetas de la vista.
- No es necesario recoger cada campo por separado. Se realiza una ligadura entre el bean y las etiquetas de la vista.
- Se declara en el controlador.

# Objetos de Respaldo

- Las **colecciones** que podemos utilizar para rellenar controles de formulario son:
  - **Map**: Se suelen utilizar para controles que tengan value / descripción. Por ejemplo, select.
  - **List**: Cuando no queramos rellenar propiedades value.
  - Estas colecciones se referencian en el atributo **items** de la etiqueta de la vista.

# Mas Anotaciones en el Controlador

- **@ModelAttribute("countryList")**
  - Esta anotación se utiliza para rellenar colecciones que luego van a ser utilizadas en checkbox, radiobuttons, select, etc.
  - En este caso, en la vista podríamos referenciar el nombre: countryList.

```
<form:select path="country">
```

```
 <form:option value="NONE" label="--- Select ---"/>
```

```
 <form:options items="{countryList}" />
```

```
</form:select>
```



# Mas Anotaciones en el Controlador

- **@ModelAttribute("nombreBean")**
  - También se puede utilizar para inicializar Beans que luego queremos utilizar en la Vista.
  - El nombre que damos en la anotación coincide con el atributo: `commandName` de la etiqueta `<form:form commandName="nombreBean">`
  - El método puede ser:

```
@ModelAttribute("nombreBean")
public ClaseBean populateForm(){
 return new ClaseBean();
}
```

# Mas Anotaciones en el Controlador

- **@initBinder**

- Esta anotación se puede utilizar para anotar métodos en el controller.
- Por parámetro el método recibirá un objeto de tipo: `WebDataBinder`.
- Dentro de este método lo que haremos es registrar validadores que queramos utilizar.
- Ejemplo:

```
@InitBinder
```

```
protected void initBinder(WebDataBinder binder) {
 binder.setValidator(new ValidadorDatosCoche());
}
```

# Mas Anotaciones en el Controlador

- **@sessionAttributes**
  - Se utilizan para anotar en la clase del controlador e indicar variables que queremos almacenar a nivel de sesión.

```
@Controller
@RequestMapping("/editPet.do")
@SessionAttributes("pet")
public class EditPetForm {
 // ...
}
```

# Implementación de Validadores

- Una clase que implemente el interface Validator.
- Dentro del controlador registrar el validador dentro de un método anotado con `@InitBinder`.
- El método del controller que procesa el formulario recibe dos parámetros:
  - **@Validated ClaseBean objetoBean:**
    - Recibe el bean anotado. La validación se lanza automática cumpliendo los pasos anteriores.
  - **BindingResult result:**
    - Podemos detectar si hay errores o no.
      - `if (result.hasErrors()) // Volvemos al mismo form`
      - `else // A otra vista ...`
- En la vista utilizar la etiqueta `<form:errors path="nombre_prop" cssClass="una clase">`
- El form debe tener la propiedad `commandName="nombre_bean"`

# Validadores

- Los validadores tienen que implementar la interface: **Validator**.

```
public boolean supports(Class<?> clazz) {
 // clase del bean al que da soporte este validador
 return DatosCoche.class.equals(clazz);
}

public void validate(Object target, Errors errors){
 // target representa un objeto de la clase que
 // estamos validando. Errors son los errores asociados.
}
```

# Validadores

- Dentro del método **validate**:
  - Hace un **casting** al tipo de objeto que vamos a validar.
  - Hacer las comprobaciones:
    - Campos requeridos.
    - Valores dentro de un rango.
    - Formatos adecuados.
    - Lo que queremos validar ...

# Validadores

- Para campos requeridos podemos utilizar:  
**`ValidationUtils.rejectIfEmptyOrWhitespace(errors, "modelo", "field.modelo.required", "El modelo es obligatorio");`**
  - errors: los errores asociados.
  - “modelo”: El nombre de la propiedad a validar.
  - “field.modelo.required”: La clave del error de validación.
  - El último campo es una descripción.

# Validadores: métodos de errors

- Para indicar que los valores no son correctos:  
**errors.rejectValue("anho", "field.anho.invalid", "El anho es incorrecto");**

- “anho”: nombre de la propiedad.
- “field.anho.invalid”: La clave del error de validación.
- El último: la descripción del error.

**errors.hasFieldErrors(“una\_propiedad”):**

- Podemos comprobar si una propiedad tiene o no algún error.

– Mas info:

- <http://static.springsource.org/spring/docs/3.1.x/javadoc-api/org/springframework/validation/Errors.html>



# Etiquetas de Error

- Mostrando todos los errores:
  - **<form:errors** *path="\*" cssClass="errorblock" element="div"/>*
    - El \* indica todas las propiedades.
    - cssClass: La clase del estilo.
    - element: El id del elemento HTML donde queremos mostrarlas. Normalmente una capa (DIV).
- Ligadas a una propiedad y clase de CSS.
  - **<form:errors** *path="confirmPassword" cssClass="error" />*

# Externalizar los mensajes de error

- Los mensajes de error se pueden externalizar a un fichero `.properties`.
- El fichero se coloca en:
  - `src/main/resources/mensajes_es.properties`
  - En el nombre tiene que aparecer el locale.
  - El fichero se compone de claves (las que queramos) y el valor (el mensaje a mostrar).
  - En el fichero de configuración del Servlet hacemos referencia a la ubicación del fichero.

# Externalizar los mensajes de error

```
<beans:bean id="messageSource"
 class="org.springframework.context.support.ResourceBundleMessageSource">
 <beans:property name="basename">
 <beans:value>mensajes</beans:value>
 </beans:property>
</beans:bean>
```

Clave del fichero

- En el fichero tendremos algo similar a esto:  
*matricula.vacia=La matricula es un campo obligatorio*
- En el Validador cuando agregamos un error ...
  - `ValidationUtils.rejectIfEmptyOrWhitespace(errors, "matricula", "matricula.vacia");`
- Se puede utilizar con un mensaje por defecto. En caso de que no encuentre la clave en el fichero, se muestra dicho mensaje:
  - `ValidationUtils.rejectIfEmptyOrWhitespace(errors, "matricula", "matricula.vacia","otro mensaje por defecto");`

# i18n

- Para internacionalizar una app en Spring creamos un fichero properties por cada idioma.
- El nombre de los ficheros coincide salvo las siglas del país que representan.
  - **messages\_en.properties**
  - **messages\_de.properties**
- Estos ficheros representan contenido estático de la app. Se colocan en la carpeta de **resources**.

# i18n – Ficheros de properties

- Los ficheros definen una serie de claves (que tiene que ser la misma en todos los ficheros) / valor.
- `message_es.properties`  
    `label.nombre=nombre`  
    `label.ape=apellido`
- `message_en.properties`  
    `label.nombre=name`  
    `label.ape=surname`

# i18n – En la vista

- En la vista necesitamos referenciar las dos librerías de spring:
- [<%@taglib uri="http://www.springframework.org/tags" prefix="\*\*spring\*\*"%>](http://www.springframework.org/tags)
- [<%@taglib uri="http://www.springframework.org/tags/form" prefix="\*\*form\*\*"%>](http://www.springframework.org/tags/form)
- **form**: para las etiquetas de HTML.
- **spring**: para mostrar las etiquetas de los mensajes.

- Ejemplo

```
<form:label path="firstName">
 <spring:message code="label.firstname"/>
</form:label>
```

# i18n – En la vista

- El cambio de idioma desde la vista se puede realizar con un enlace para cada idioma:

en | de

- Con cada enlace mandamos el locale correspondiente:

```
en
```

```
de
```

# i18n – en el controller

- Necesitamos un método que esté mapeado con el parámetro recibido:
  - `@RequestMapping(value="/{lang}")`
  - Método GET.
  - El método puede recibir el locale (por consultarlo), pero no es obligatorio.
  - El cambio del locale quien lo hace es un interceptor: **localeChangeInterceptor**.



# i18n - Configuración

- En el fichero de **configuración del Servlet**:
  - Definir un bean para dar la ubicación de los ficheros de properties.

```
<beans:bean id="messageSource"
 class="org.springframework.context.support.ResourceBundleMessageSource">
 <beans:property name="basename" value="messages" />
</beans:bean>
```

- En la propiedad basename indicar el prefijo de los ficheros.
- Si están ubicados en una carpeta:
  - value="locale/messages"

# i18n - Configuración

- Definir el interceptor (del namespace: **mvc**).

```
<interceptors>
 <beans:bean
 class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
 <beans:property name="paramName" value="lang"></beans:property>
 </beans:bean>
</interceptors>
```

- En la propiedad paramName se indica el nombre del parámetro que indicamos en enlace de la vista → `<a href="?lang=en">`

# i18n - Configuración

- Definir un localeResolver, indicando el idioma por defecto:

```
<beans:bean id="localeResolver"
 class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
 <beans:property name="defaultLocale" value="en" />
</beans:bean>
```

- Dos posibles clases:
  - CookieLocaleResolver
  - SessionLocaleResolver

# Tiles

- Spring MVC soporta la integración con Tiles para dividir las vistas en varias partes y poder reutilizarlas.
- Para trabajar con Tiles necesitamos indicar unas dependencias en Maven:

<dependency>

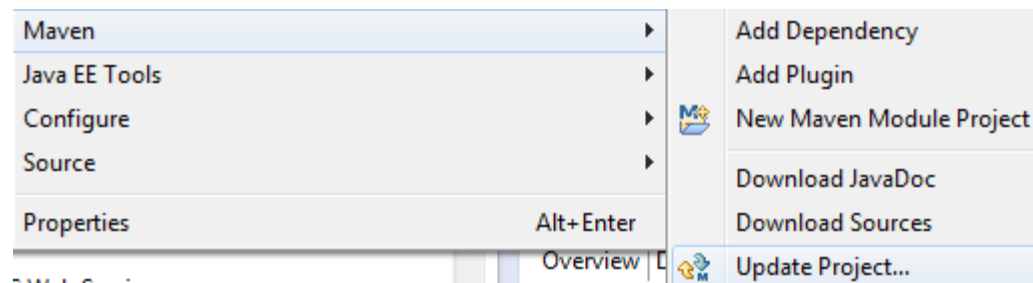
<groupId>org.apache.tiles</groupId>

<artifactId>tiles-extras</artifactId>

<version>2.2.2</version>

</dependency>

Agregar las dependencias en el fichero pom.xml y actualizar proyecto.



# Tiles

- En el fichero de configuración del Servlet tenemos que declarar los siguientes beans:
  - Un bean **TilesConfigurer** que indique la ruta donde se encuentra el fichero de configuración **tiles.xml**

```
<beans:bean id="tilesConfigurer"
 class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
 <beans:property name="definitions">
 <beans:list>
 <beans:value>/WEB-INF/tiles.xml</beans:value>
 </beans:list>
 </beans:property>
</beans:bean>
```

# Tiles

- Otro bean que necesitamos es un viewResolver: **TilesView** que trabaje con Tiles.

```
<beans:bean id="viewResolver"
 class="org.springframework.web.servlet.view.UrlBasedViewResolver">
 <beans:property name="viewClass">
 <beans:value>
 org.springframework.web.servlet.view.tiles2.TilesView
 </beans:value>
 </beans:property>
</beans:bean>
```

# Tiles.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
 "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
 "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">
<tiles-definitions>
 <definition name="base.definition"
 template="/WEB-INF/views/layout.jsp">
 <put-attribute name="title" value="" />
 <put-attribute name="header" value="/WEB-INF/views/header.jsp" />
 <put-attribute name="menu" value="/WEB-INF/views/menu.jsp" />
 <put-attribute name="body" value="" />
 <put-attribute name="footer" value="/WEB-INF/views/footer.jsp" />
 </definition>

 <definition name="contact" extends="base.definition">
 <put-attribute name="title" value="Contact Manager" />
 <put-attribute name="body" value="/WEB-INF/views/contact.jsp" />
 </definition>
</tiles-definitions>
```

Define una plantilla base. Todas las partes a sustituir se indican como atributos.  
A partir de esta podemos heredas modificando únicamente lo que cambia.

En la propiedad **value** podemos indicar texto o la ruta a un JSP.

# Las Vistas

- La plantilla:
  - Debe hacer referencia a la siguiente librería:  
`<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles"%>`
- En esta página se define la estructura (con Capas) y las partes modificables de la plantilla se rellenan con:
  - `<tiles:insertAttributes name="nombre_de_att" />`



# En el Controller

```
@RequestMapping(value= "/")
public ModelAndView showContacts() {
 return new ModelAndView("contact", "command", new
 Contact());
}
```

- Podemos devolver la vista y el modelo a la vez:
  - **contact**: es el nombre que hemos indicado en la definición de vistas en el **Tiles.xml**
  - **New Contact()**: creamos un bean de respaldo para la vista y se llama “**command**”. Esta cadena tendrá que aparecer en el atributo del form:
  - `<form:form method="post" action="addContact" commandName="command">`

# File Upload

- A partir de un proyecto para Spring MVC con el plugin de Maven, incluir la siguiente dependencia en el fichero pom.xml. Al grabar se descargan los jar necesarios.

```
<dependency>
```

```
 <groupId>commons-fileupload</groupId>
```

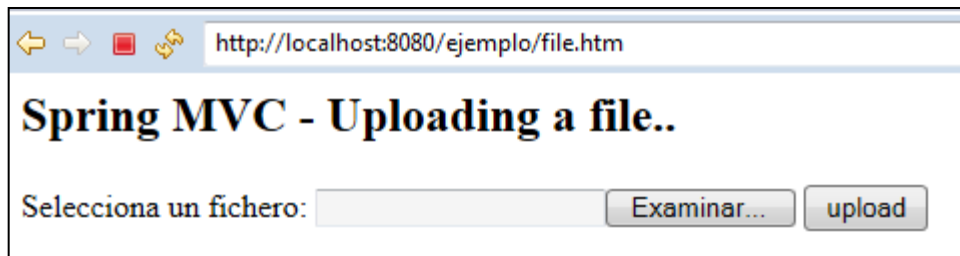
```
 <artifactId>commons-fileupload</artifactId>
```

```
 <version>1.3.1</version>
```

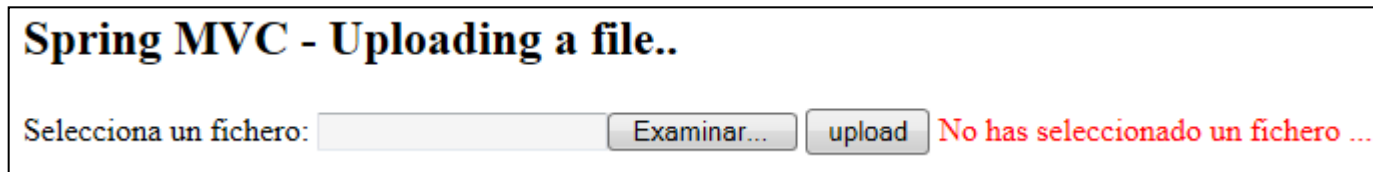
```
</dependency>
```

# FileUpload

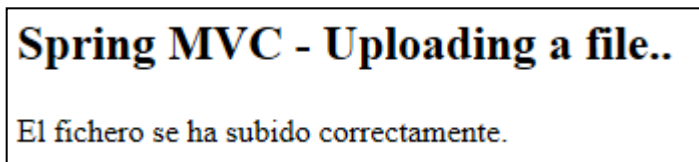
- Vistas que mostrará la aplicación:



A screenshot of a web browser window. The address bar shows 'http://localhost:8080/ejemplo/file.htm'. The page title is 'Spring MVC - Uploading a file..'. Below the title, there is a text input field labeled 'Selecciona un fichero:', followed by an 'Examinar...' button and an 'upload' button.



A screenshot of the web application interface. The title is 'Spring MVC - Uploading a file..'. Below the title, there is a text input field labeled 'Selecciona un fichero:', followed by an 'Examinar...' button and an 'upload' button. To the right of the 'upload' button, there is a red error message: 'No has seleccionado un fichero ...'.



A screenshot of the web application interface. The title is 'Spring MVC - Uploading a file..'. Below the title, there is a success message: 'El fichero se ha subido correctamente.'

# Modelo

- Representamos el fichero que vamos a subir mediante la siguiente clase:

```
public class File {
 private MultipartFile file;

 public MultipartFile getFile() {
 return file;
 }
 public void setFile(MultipartFile file) {
 this.file = file;
 }
}
```

# Modelo II

- Podemos tener un validador para controlar si seleccionan o no el fichero:

```
public class FileValidator implements Validator {
 @Override
 public boolean supports(Class<?> clazz) {
 return File.class.equals(clazz);
 }
 @Override
 public void validate(Object obj, Errors errors) {
 File file = (File) obj;
 if (file.getFile().getSize() == 0)
 errors.rejectValue("file", "valid.file",
 "No has seleccionado un fichero ...");
 }
}
```

# Controller

- En el controlador necesitamos registrar el validador.

```
@Controller
```

```
@RequestMapping("/file.htm")
```

```
public class HomeController {
```

```
@Autowired
```

```
FileValidator validator;
```

```
@InitBinder
```

```
private void initBinder(WebDataBinder binder) { // Registrar el validador ...
```

```
 binder.setValidator(validator);
```

```
}
```

# Controller II

```
@RequestMapping(method = RequestMethod.GET)
public String getForm(Model model) {
 File fileModel = new File();
 model.addAttribute("file", fileModel);
 return "file";
}
```

// Retorna la vista que hay que mostrar (viewResolver montará el nombre físico del fichero) y crear el objeto del modelo.

# Controller III

```
@RequestMapping(method = RequestMethod.POST)
public String fileUploaded(Model model, @Validated File file, BindingResult result) {
 String returnVal = "successFile";
 if (result.hasErrors()) {
 returnVal = "file";
 } else {
 MultipartFile multipartFile = file.getFile();
 try {
 // Aquí tenemos el fichero ... lo podemos grabar ..
 System.out.println(multipartFile.getBytes().length);
 } catch (IOException e) {
 // TODO Auto-generated catch block
 e.printStackTrace();
 }
 }
 return returnVal;
}
```



# Vista

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
<body>
<h2>Spring MVC - Uploading a file.. </h2>
 <form:form method="POST" commandName="file" enctype="multipart/form-data">
 Selecciona un fichero:
 <input type="file" name="file" />
 <input type="submit" value="upload" />
 <form:errors path="file" cssStyle="color: #ff0000;" />
 </form:form>
</body>
</html>
```

# Root-context.xml

- En el contexto **definir el validador** y además para poder subir el fichero necesitamos otro resolver:
- `<bean id="multipartResolver"  
class="org.springframework.web.multipart.commons.CommonsMultipartResolver" />`