Spring MVC

Víctor Herrero Cazurro

Temario

- Introducción
- Instalación y configuración
- HandlerMapping
- Controller
- ViewResolver
- View
- Formularios
- Validaciones
- Servicios REST
- Gestión de recursos, idiomas y temas

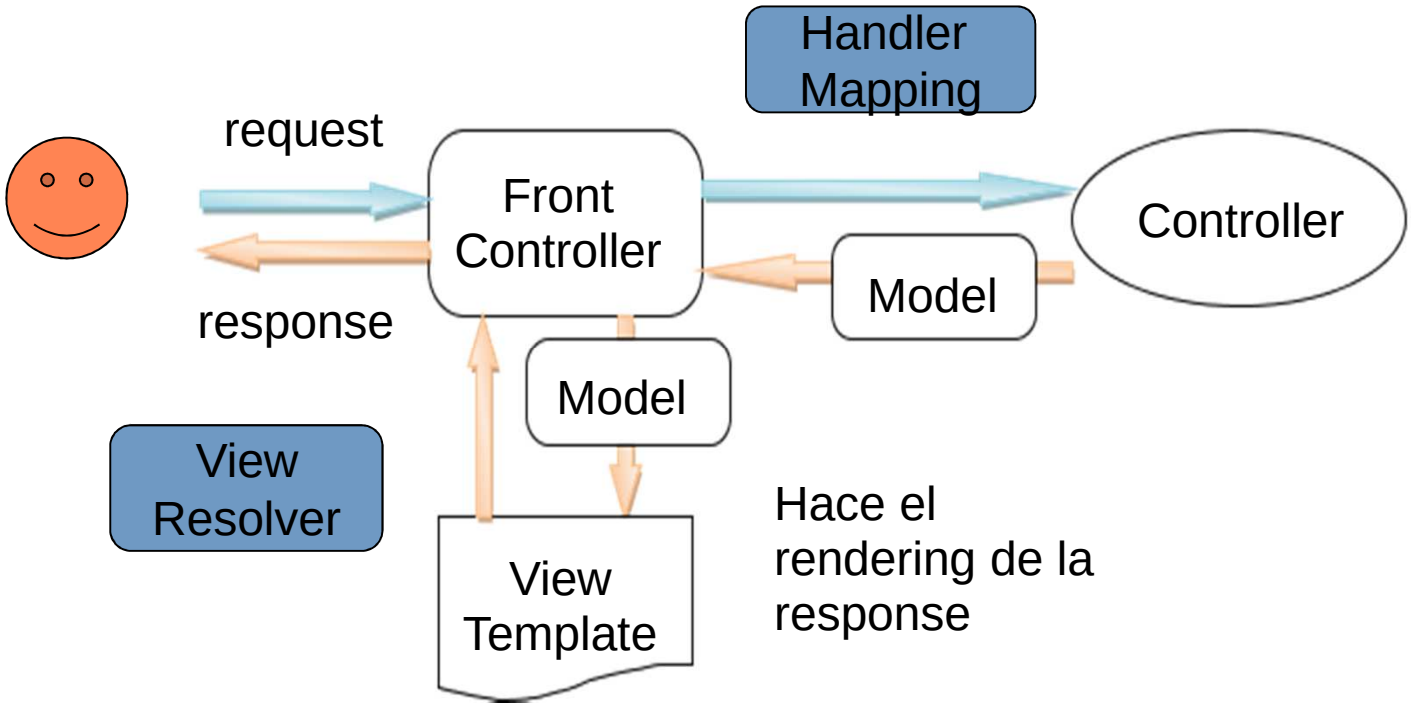
Introducción

- Spring MVC, como su nombre indica es un framework que implementa Modelo-Vista-Controlador, esto quiere decir que proporcionará componentes para que realicen cada una de esas tareas únicamente.

Introducción

- Spring MVC, como la mayoría de frameworks MVC, se basa en un ForntController, en este caso DispatcherServlet.
- Este controlador, realiza las siguientes tareas.
  - Consulta con los HandlerMapping, que controlador ha de resolver la petición.
  - Una vez el HandlerMapping le retorna que controlador ha de invocar, lo invoca para que resuelva la petición.
  - Recoge los datos del Modelo que le envía el Controlador como respuesta y el nombre de la Vista que se empleara para mostrar dichos datos.
  - Selecciona la Vista que corresponde al nombre recibido y le envía los datos del Modelo.

Introducción



Dependencias Maven

- Para obtener los jar del framework, se ha de añadir a un proyecto maven la siguiente dependencia

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>4.2.3.RELEASE</version>
</dependency>
```

DispatcherServlet

- Para configurar Spring MVC, se necesita pues definir el **DispatcherServlet** en el web.xml.

```
<servlet>
        <servlet-name>miApp</servlet-name>
        <servlet-class>
                org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
</servlet>
```

- Sera importante el nombre del servlet, ya que por defecto este buscara en el directorio WEB-INF, el xml de Spring con el nombre

```
<servlet-name>-servlet.xml -> miApp-servlet.xml
```

DispatcherServlet

- El siguiente paso será configurar el mapping

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

- Normalmente se asigna este mappping que recoge todas las peticiones, incluso las del contenido estático (html, css, js, …).

Contexto de Spring

- El siguiente paso será definir el fichero de configuración de Spring, con las siguientes cabeceras

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc">
```

- Si se quiere que algunas URL no sean tratadas por el controlador de Spring, por ejemplo HTML, CSS, JS, JPG, … se añadirá.

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

Contexto de Spring

- Se puede definir un contexto de Spring común para todos los servlet, definiendo el siguiente Listener.

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

Contexto de Spring

- Pudiendo indicarse los ficheros de configuración que lo forman con el siguiente parámetro de contexto web

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
                classpath:xfiles-config.xml,
                /WEB-INF/security.xml
        </param-value>
</context-param>
```

HandlerMapping

- Como **HandlerMapping** por defecto, Spring emplea
  - **BeanNameUrlHandlerMapping**
  - **DefaultAnnotationHandlerMapping**.
- Este ultimo, asigna peticiones, a métodos de controladores anotados con @**RequestMapping**.
- Para que se interpreten estas anotaciones, se ha de incluir en el contexto la siguiente etiqueta

```
<mvc:annotation-driven/>
```

HandlerMapping

- En la anotación @**RequestMapping**, se pueden definir los siguientes parámetros
  - **value**, la url a la que responde el método.
  - **method**, el METHOD HTTP.
  - **params**, el formato de los parámetros de la petición, es decir, que ha de aparecer a continuación de la ? en la petición GET (también sirve para POST, ….)

HandlerMapping

- Esta anotación a nivel de clase, añade un nivel mas a todas las url de la clase **Controller**.
- Dentro de **value**, se pueden incluir variables, siguiendo el formato “{<variable>}”, de tal forma que gracias a la anotación @**PathVariable**, en un parámetro del método, se inyecte esa parte de la url en el parámetro del método.

HandlerMapping

- Ejemplo de uso de **@RequestMapping**

```
@RequestMapping(path="/saludar/{nombre}")
public ModelAndView saludar(
        @PathVariable("nombre") String nombre, Model model){

        model.addAttribute("saludo", "Hola " + nombre + "!!!");
        return new ModelAndView("saludo", model.asMap());
}
```

HandlerMapping

- API de **HandlerMapping**

Nombre	Descripción y Ejemplo
BeanNameUrlHandlerMapping	Usa el nombre del Bean Controlador como mapeo <bean name="/inicio.htm" … >
SimpleUrlHandlerMapping	Mapea mediante propiedades <prop key="/verClientes.htm">
ControllerClassHandlerMapping	Usa el nombre de la clase asumiendo que termina en Controller y sustituyéndolo por .htm
CommonsPathHandlerMapping	Mapea mediante anotaciones en la clase Controller @PathMap("/home.htm")
DefaultAnnotationHandlerMapping	Emplea la propiedad path de la anotación @RequestMapping

HandlerMapping

- Un ejemplo de **BeanNameUrlHandlerMapping**.

```
<bean
        class="org.springframework.web.servlet.handler.BeanNameUrlHan
dlerMapping" />

<bean name="/helloWorld.html"
        class="org.ejemplos.springmvc.EjemploAbstractController" />
```

- Donde se ha de definir un Bean de tipo **controller** cuyo nombre sea la Url con la que está mapeado.

HandlerMapping

- Un ejemplo de **SimpleUrlHandlerMapping**.

```
<bean class=
"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
        <props>
                <prop key="/index.htm">controladorIndex</prop>
                <prop
key="/buscarpedido.htm">controladorConsultas</prop>
        </props>
        </property>
</bean>
```

- Deberán existir dos bean **Controller** llamados **controladorIndex** y **controladorConsultas**.

Controller

- Para definir un **Controller** basta con definir una clase con la anotación @**Controller** y añadir en el xml de Spring la etiqueta.

```
<context:component-scan base-package="<paquete donde están los controladores>"/>
```

- La firma de los métodos del controlador es flexible, puede retornar
  - **String**
  - **View**
  - **ModelAndView**
  - **Objeto (Anotado con @ResponseBody)**

Controller

- Y puede recibir como parámetro
  - **Model**: Datos retornados a la vista.
  - **@PathVariable**: Dato que llega en el path de la Url.

```
//http://……./saludar/Victor
@RequestMapping(path="/saludar/{nombre}")
public ModelAndView saludar(@PathVariable("nombre") String nombre){
```

  - **@RequestParam**: Dato que llega en los parametros de la Url.

```
//http://……./saludar?nombre=Victor
@RequestMapping(path="/saludar")
public ModelAndView saludar(@RequestParam("nombre") String nombre){
```

Controller

- API antigua de **Controller**

Nombre	Descripción
AbstractController	Muy sencillo, cuando se necesita poca funcionalidad (Servlet)
ParametrizableViewController	Muestra una vista estática, sin datos de formularios
AbstractCommandController	Coge parámetros de la request y puede validar
SimpleFormController	Permite mostrar un formulario y procesar sus datos <property name="formView" … />
AbstractWizardFormController	Permite mostrar y procesar los datos de varios formularios

Controller

- Un ejemplo de **Controller** definido con el API antigua

```
public class ControladorIndex implements Controller{
        public ModelAndView handleRequest(HttpServletRequest req,
                                HttpServletResponse res) throws Exception {
                System.out.println("peticion recibida en” +
                                                “ControladorIndex");
                return new ModelAndView("index");
        }
}
```

Controller

- Se puede realizar un mapeo directo de una vista como controlador, es decir, otorgar a una vista directamente una URL.

```
<mvc:view-controller path="/" view-name="welcome" />
```

ViewResolver

- El último componente a definir es el **ViewResolver**.
- Se proporcionan distintas implementaciones para resolver las vistas.
- Uno de los mas habituales es **InternalResourceViewResolver**, permite interpretar el string devuelto por el controlador, como parte de la url de un recurso.

```
<bean class=
"org.springframework.web.servlet.view.InternalResourceViewResolver">
        <propertyname="prefix"value="/WEB-INF/views/"/>
        <propertyname="suffix"value=".jsp"/>
</bean>
```

ViewResolver

- El API de **ViewResolver**:

View resolver	Descripción
BeanNameViewResolver	Finds an implementation of View that's registered as a <bean> whose ID is the same as the logical view name.
ContentNegotiatingViewResolver	Delegates to one or more other view resolvers, the choice of which is based on the content type being requested.
FreeMarkerViewResolver	Finds a FreeMarker-based template whose path is deter- mined by prefixing and suffixing the logical view name.
InternalResourceViewResolver	Finds a view template contained within the web applica- tion's WAR file. The path to the view template is derived by prefixing and suffixing the logical view name.

ViewResolver

- El API de **ViewResolver**:

View resolver	Descripción
JasperReportsViewResolver	Finds a view defined as a Jasper Reports report file whose path is derived by prefixing and suffixing the logical view name.
ResourceBundleViewResolver	Looks up View implementations from a properties file.
TilesViewResolver	Looks up a view that is defined as a Tiles template. The name of the template is the same as the logical view name.
VelocityLayoutViewResolver	This is a subclass of VelocityViewResolver that supports page composition via Spring's VelocityLayoutView (a view implementation that emulates Velocity's VelocityLayoutServlet).
VelocityViewResolver	Resolves a Velocity-based view where the path of a Velocity template is derived by prefixing and suffixing the logical view name.

ViewResolver

- El API de **ViewResolver**:

View resolver	Descripción
XmlViewResolver	Finds an implementation of View that's declared as a <bean> in an XML file (/WEB-INF/views.xml). This view resolver is a lot like BeanNameViewResolver except that the view <bean>s are declared separately from those for the application's Spring context.
XsltViewResolver	Resolves an XSLT-based view where the path of the XSLT stylesheet is derived by prefixing and suffixing the logical view name.
UrlBasedViewResolver	This is the base class for some of the other view resolvers, such as **InternalResourceViewResolver**. It can be used on its own, but it's not as powerful as its sub- classes. For example, UrlBasedViewResolver is unable to resolve views based on the current locale.

ViewResolver

- Un ejemplo de **XmlViewResolver**.

```
<bean class="org.springframework.web.servlet.view.XmlViewResolver">
        <property name="location" value="/WEB-INF/views.xml"/>
        <property name="order" value="0"/>
</bean>
```

- Donde se ha de definir otro fichero XML de context de spring con los bean de las vistas.

```
<bean id="pdf/listadoAfines"
class="com.aplicacion.parejas.presentacion.vistas.ListadoAfinesPdfView"/>
<bean id="excell/listadoAfines"
class="com.aplicacion.parejas.presentacion.vistas.ListadoAfinesExcellView"/>
<bean id="json/listadoAfines"
class="org.springframework.web.servlet.view.json.MappingJacksonJsonView
"/>
```

ViewResolver

- Un ejemplo de **ResourceBundleViewResolver**

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.ResourceBundleViewResolver"
>
        <property name="basename" value="views" />
</bean>
```

- Donde en el classpath existirá el fichero **views.properties** con el siguiente contenido

```
reporteAfines.(class)=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView

reporteAfines.url=/WEB-INF/jasperTemplates/reporteAfines.jasper

reporteAfines.reportDataKey=afinesKey
```

ViewResolver

- Los **ViewResolver** al igual que los **HandlerMapping**, se pueden apilar, es decir definir una batería de objetos, que se irán consultando en orden por parte del **DispatcherServlet**.
- Para ello se han de ordenar

```
<bean >
        <property name="order" value="0"/>
</bean>
```

- Es importante que de emplear el **InternalResourceViewResolver**, este sea el ultimo (Valor mas alto).

View - JasperReport

- En este caso, la vista mapeada, es un vista que se renderiza a partir de una plantilla compilada de JasperReport, debiendo pasar por parámetro
  - **(class)**: siempre será a misma **JasperReportsPdfView**.
  - **url**: El path donde encontrar la plantilla.
  - **reportDataKey**: El parámetro del modelo que servirá para rellenar la plantilla, deberá ser de tipo **JRBeanCollectionDataSource**.
- Todos los parámetros asociados al nombre de la vista que retorna el controlador (**reporteAfines**)

View - Excel

- Para generar una vista en formato de hoja de **Excel**, se necesita una nueva dependencia en el proyecto, con Maven.

```
<dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi</artifactId>
        <version>3.10.1</version>
</dependency>
```

- Esta nueva dependencia aporta las clases necesarias para poder implementar la View con

```
org.springframework.web.servlet.view.document.AbstractExcelView
```

View - Excel

- Algunas de las clases del Api de **Poi** son
  - **HSSFWorkbook**
  - **HSSFSheet**
  - **HSSFRow**
  - **HSSFCell**

View - Pdf

- Para generar una vista en formato **pdf**, se necesita una nueva dependencia en el proyecto, con Maven.

```
<dependency>
        <groupId>com.lowagie</groupId>
        <artifactId>itext</artifactId>
        <version>4.2.1</version>
</dependency>
```

- Esta nueva dependencia aporta las clases necesarias para poder implementar la View con

```
org.springframework.web.servlet.view.document.AbstractPdfView
```

View - Pdf

- Algunas de las clases del Api de **Lowagie** son
  - **Document**
  - **PdfWriter**
  - **Paragraph**
  - **Table**

Expresiones EL

- Los parámetros recibidos en la vista, podrán ser recogidos empleado expresiones EL.
- Por ejemplo para acceder a un parámetro “listado” que este en la request, se podrá hacer.

```
${listado}
```

Formularios

- Para trabajar con formularios, Spring MVC, proporciona una librería de tags TLD, para los jsp.

```
<%@ taglibprefix="sf“ uri="http://www.springframework.org/tags/form"%>
```

Formularios

Tag	Descripción
checkbox	Renders an HTML 'input' tag with type 'checkbox'.
checkboxes	Renders multiple HTML 'input' tags with type 'checkbox'.
errors	Renders field errors in an HTML 'span' tag.
form	Renders an HTML 'form' tag and exposes a binding path to inner tags for binding.
hidden	Renders an HTML 'input' tag with type 'hidden' using the bound value.
input	Renders an HTML 'input' tag with type 'text' using the bound value.
label	Renders a form field label in an HTML 'label' tag.
option	Renders a single HTML 'option'. Sets 'selected' as appropriate based on bound value.
options	Renders a list of HTML 'option' tags. Sets 'selected' as appropriate based on bound value.
password	Renders an HTML 'input' tag with type 'password' using the bound value.
radiobutton	Renders an HTML 'input' tag with type 'radio'.
select	Renders an HTML 'select' element. Supports databinding to the selected option.

Formularios

- Un ejemplo de uso sería

```
<form:form action="altaUsuario" commandName= "user">
        <table>
                <tr>
                        <td>Nombre:</td>
                        <td><form:input path=“nombre" /></td>
                </tr>
                <tr>
                        <td>Apellidos:</td>
                        <td><form:input path=“apellidos" /></td>
                </tr>
                <tr>
                        <td colspan="2">
                        <input type="submit" value=“Guardar info" />
                        </td>
                </tr>
        </table>
</form:form>
```

Formularios

- En el ejemplo anterior, se han definido a nivel del formulario.
  - **action**: Indica la Url del Controlador.
  - **commandName**: Indica la clave con la que se envía el objeto que se representa en el formulario.

Formularios

- El objeto que se representa en el formulario ha de existir al representar el formulario.
- Es típico para los formularios definir dos controladores uno **GET** y otro **POST**.
  - El **GET** inicializara el objeto.
  - El **POST** tratara el envío del formulario.
- Para recuperar en el controlador el objeto enviado, se emplea la anotación **@ModelAttribute**.

Formularios

- Ejemplo de controlador de formulario

```
@RequestMapping(value="altaPersona", method=RequestMethod.GET)
public String inicializacionFormularioAltaPersonas(Model model){
        Persona p = new Persona(null, "", "", null, "Hombre", null);
        model.addAttribute("persona", p);
        model.addAttribute("listadoSexos",
                                        new String[]{"Hombre","Mujer"});
        return "formularioAltaPersona";
}
@RequestMapping(value="altaPersona", method=RequestMethod.POST)
public String procesarFormularioAltaPersonas(
        @ModelAttribute("persona") Persona p,  Model model){
                servicio.altaPersona(p);
                model.addAttribute("estado", "OK");
                model.addAttribute("persona", p);
                model.addAttribute("listadoSexos",
                                        new String[] {"Hombre","Mujer"});
                return "formularioAltaPersona";
}
```

Formularios

- Otra opción para inicializar los objetos necesarios para el formulario, sería crear un método anotado con @**ModelAttribute**, indicando la clave del objeto del Modelo que disparará la ejecución de este método.

```
@ModelAttribute("persona")
public Persona initPersona(){
        return new Persona();
}
```

Formularios

- Para definir un conjunto de checkbox o radiobuttons a partir de un collection haríamos.

```
<form:radiobuttons path="sexo" items="${listadoSexos}"/>
```

Validaciones

- Spring MVC soporta validaciones de JSR-303.
- Para aplicarlas se necesita una implementación como **hibernate-validator**, para añadirla con Maven.

```
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-validator</artifactId>
        <version>4.0.2.GA</version>
</dependency>
<dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.5.6</version>
</dependency>
```

Validaciones

- Para activar la validación entre vista y controlador, se añade a los parámetros de los métodos del controlador, la anotación @**Valid**.

```
@RequestMapping(method = RequestMethod.POST)
public Persona altaPersona(@Valid @RequestBody Persona persona) {}
```

- La presencia de esta clase, obligará a cumplir las validaciones definidas en la clase del parámetro empleando JSR-303.

Validaciones

- Si además se quiere conocer el estado de la validación para ejecutar la lógica del controlador, se puede indicar en los parámetros que se recibe un objeto **BindingResult**.

```
public String altaPersona(
        @Valid @ModelAttribute("persona") Persona p,
        BindingResult result,
        Model model){}
```

- Teniendo este objeto un método **hasErrors**() que indica si hay errores de validación.

Validaciones

- Las anotaciones están definidas en el paquete **javax.validation.constraints**.
- Y son
  - **@Max**
  - **@Min**
  - **@NotNull**
  - **@Null**
  - **@Future**
  - **@Past**
  - **@Size**
  - **@Pattern**

Validaciones

- Un ejemplo de su uso

```
@Size(min=6,max=20, message="El password debe tener al menos 6 caracteres.")
String password;
@Min(value=2)
int id;
```

Validaciones Custom

- Se pueden definir validadores nuevos e incluirlos en la validación automatizada, para ello hay que implementar la **inteface**.

```
org.springframework.validation.Validator
```

- Ejemplo de Validador

```
public class PersonaValidator implements Validator {
	@Override
	public boolean supports(Class<?> clazz) {
		return Persona.class.equals(clazz);
	}
	@Override
	public void validate(Object obj, Errors e) {
		Persona persona = (Persona) obj;
		e.rejectValue("nombre", "formulario.persona.error.nombre");
	}
}
```

Validaciones Custom

- Y añadir una instancia de ese validador al Binder del controlador, creando un método en el controlador, anotado con **@InitBinder**

```
@InitBinder
protected void initBinder(final WebDataBinder binder) {
        binder.addValidators(new PersonaValidator());
}
```

Validaciones

- Si se desean ver los errores de validación asociados a un campo de un formulario, se puede emplear la etiqueta <**errors**> de la librería de etiquetas de formularios de Spring.

```
<form:errors path="*"/>
```

- Hay que indicar un **path**, que indicará de que propiedad se muestran los errores, es equivalente al **path** de los input..
  - * Es para todas

Servicios Web REST

- Servicios basados en HTTP.
- Orientados a recursos.
- Cada recurso será referenciado por una URI única.
- Se emplean los METHOD HTTP, para crear un CRUD (Alta-Baja-Modificación-Consulta) de las entidades.

Servicios Web REST

- **GET**. Empleado para consultar información (SELECT). Los datos se retornan en el cuerpo de la respuesta, normalmente como JSON.
- **POST**. Empleado para dar de alta un nuevo elemento (INSERT). La información del elemento se envía en el cuerpo de la petición, normalmente como JSON.
- **PUT**. Empleado para modificar un elemento (UPDATE). Similar a POST.
- **DELETE**. Empleado para borrar un elemento.

Servicios Web REST

- El API de Spring no soporta de forma nativa la conversión de objetos a JSON, para ello se recurre al API de Binding de Jackson.
- Con maven se consigue añadiendo

```
<dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.5.3</version>
</dependency>
```

Servicios Web REST

- Si se desea controlar como realizar el binding, se puede emplear la anotación **@JsonFormat**

```
public class Persona {
		private int id;
		private String nombre;

		@JsonFormat(shape=Shape.STRING, pattern="dd-MM-yyyy")
		private Date fechaNacimiento;
}
```

Servicios Web REST

- Se emplearán las anotaciones
  - **@RestController**: Indica que el controlador, lo es de un servicio Rest. Como **@Controller**, pero por defecto lo devuelto por los métodos está anotado con **@ResponseBody**.
  - **@ResponseBody**: Indica que se ha de "pintar" lo retornado por el método en el cuerpo de la respuesta, para ello se emplearán **HttpMessageConverter**.
  - **@RequestBody**: Similar al anterior pero para el cuerpo de la petición.

Servicios Web REST

- Si se desea consumir un servicio Rest desde java, se puede emplear la clase **RestTemplate** de Spring, que permite fácilmente consumir este tipo de servicios.

```
String REST_SERVICE_URI = "http://localhost:8080/SpringMVC4";
RestTemplate restTemplate = new RestTemplate();
Persona persona = restTemplate.getForObject(
            REST_SERVICE_URI+"/persona/1", Persona.class);
Persona persona = restTemplate.postForObject(
                        REST_SERVICE_URI+"/persona",
                        new Persona(12,"Victor",new Date()),   Persona.class);
restTemplate.put(REST_SERVICE_URI+"/persona/1",
                        new Persona(12,"Victor",new Date()),   Persona.class);
restTemplate.delete(REST_SERVICE_URI+"/persona/1");
```

Mensajes

- Se pueden definir ficheros e recursos internacionalizables, para ello hay que definir el siguiente bean, indicando el nombre base del fichero de properties.

```
<bean id="messageSource"
        class="org.springframework.context.support.ReloadableResourceB
undleMessageSource">
        <property name="basename"
                value="/WEB-INF/messages/messages" />
</bean>
```

Mensajes

- Para mostrar los mensajes en las JSP, se pueden emplear las etiquetas de JSTL.
- Para añadir JSTL con Maven

```
<dependency>
        <groupId>jstl</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
</dependency>
```

- Para añadir las etiquetas de formato

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

- La etiqueta para leer el properties

```
<fmt:message key="<clave en el properties>"/>:
```

Mensajes

- O también se puede emplear la librería de etiquetas que proporciona Spring MVC

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>
```

- Que proporciona la etiqueta

```
<spring:message code="<clave en el properties>"/>
```

i18n

- Se puede cambiar el **Locale** de la request a través de un parámetro de la petición, incluyendo el siguiente interceptor.

```
<mvc:interceptors>
        <bean class=
"org.springframework.web.servlet.i18n.LocaleChangeInterceptor" />
</mvc:interceptors>
```

- El parámetro que se debe enviar es

```
http://.....?locale=es
```

i18n

- El **Locale**, se puede almacenar en una Cookie, para no tener que estar enviándola como parámetro en cada **request**.

```
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver" />
```

- O también se puede almacenar en la **session**

```
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.SessionLocaleResolver" />
```

Temas

- Si se desea emplear **Temas**, habrá que definir un origen de los **Temas**, para ello se define un **ThemeSource**, el más habitual será

```
<bean id="themeSource“ class=
"org.springframework.ui.context.support.ResourceBundleThemeSource">
        <property name="basenamePrefix" value="theme-" />
</bean>
```

- Donde se indica el nombre base de unos ficheros de **properties**, que deberán existir para cada **Tema** a emplear

```
theme-default.properties
theme-aqua.properties
```

Temas

- Se puede cambiar el **Tema** a través de un parámetro de la **request**, incluyendo el siguiente interceptor.

```
<mvc:interceptors>
        <bean id="themeChangeInterceptor“ class=
        "org.springframework.web.servlet.theme.ThemeChangeInterceptor">
                <property name="paramName" value="theme" />
        </bean>
</mvc:interceptors>
```

- El parámetro que se debe enviar es

```
http://.....?theme=aqua
```

Temas

- El **Tema** empleado, se puede almacenar en una Cookie, para no tener que estar enviándola como parámetro en cada **request**.

```
<bean id="themeResolver“ class=
            "org.springframework.web.servlet.theme.CookieThemeResolver">
            <property name="defaultThemeName" value="default" />
</bean>
```

- O también se puede almacenar en la **Session**

```
<bean id="themeResolver"
class="org.springframework.web.servlet.i18n.SessionThemeResolver" >
            <property name="defaultThemeName" value="default" />
</bean>
```

Temas

- La etiqueta <spring:theme>

```
<link  rel="stylesheet" href="<spring:theme code='css'/>" type="text/css" />
<spring:theme code="welcome.message" />
```

- Permite leer propiedades del fichero de properties de Tema.

@VictorHerrero1

Víctor Herrero Cazurro

victorherrerocazurro

victorherrerocazurro