|  |
| --- |
| **Capítulo 5:**  **Modelo Vista Controlador** |
| ***Peticiones web, controladores, formularios y vistas.*** |
|  |
| Gracias a Spring MVC podremos crear aplicaciones robustas basadas en el patrón Modelo Vista Controlador (MVC) aportando herramientas de fácil comprensión y utilización y encapsulando el manejo y manipulación de los Servlets. |
|  |
| **Javier Sevilla Sánchez**  Trabajo de fin de carrera de la Ingeniería Técnica en Informática de Gestión (I.T.I.G) |
|  |

Contenido

[¿Qué es Spring MVC? 4](#_Toc293370303)

[Características de Spring MVC 4](#_Toc293370304)

[DispatcherServlet 5](#_Toc293370305)

[Configuración de DispatcherServlet y ContextLoaderListener 6](#_Toc293370306)

[Usando Spring MVC sin anotaciones 8](#_Toc293370307)

[Creando nuestro primer Controlador 9](#_Toc293370308)

[El objeto ModelAndView 10](#_Toc293370309)

[Configuración del controlador 10](#_Toc293370310)

[View Resolvers 10](#_Toc293370311)

[Crear un JSP 11](#_Toc293370312)

[Pasos de la petición 11](#_Toc293370313)

[Usando Spring MVC con Anotaciones y Con Spring 3 12](#_Toc293370314)

[Definir el controlador en el fichero \*-servlet.xml. 13](#_Toc293370315)

[Mapeando peticiones con @RequestMapping 13](#_Toc293370316)

[Peticiones URL, usando @RequestMapping 14](#_Toc293370317)

[Mapeo avanzado con @RequestMapping 15](#_Toc293370318)

[Tipos devueltos soportados 17](#_Toc293370319)

[Enlazando parámetros de la petición a parámetros de métodos con @RequestParam 18](#_Toc293370320)

[@RequestBody 18](#_Toc293370321)

[@ResponseBody 19](#_Toc293370322)

[HttpEntity 19](#_Toc293370323)

[Proporcionando un enlace a los datos del modelo con @ModelAttribute 20](#_Toc293370324)

[Especificando atributos para almacenar en la sesión 20](#_Toc293370325)

[Uso de cookies con @CookieValue 21](#_Toc293370326)

[Mapeando cabeceras de peticiones con @RequestHeader 21](#_Toc293370327)

[Personalizando el enlace a Datos 22](#_Toc293370328)

[Mapeo de controladores 22](#_Toc293370329)

[Interceptando peticiones con HandlerInterceptor 23](#_Toc293370330)

[Resolver vistas 23](#_Toc293370331)

[ViewResolver 23](#_Toc293370332)

[Encadenando ViewResolvers 24](#_Toc293370333)

[Redireccionando a vistas 24](#_Toc293370334)

[RedirectView 25](#_Toc293370335)

[El prefijo “redirect:” 25](#_Toc293370336)

[El prefijo “forward:” 25](#_Toc293370337)

[ContentNegotiatingViewResolver 26](#_Toc293370338)

[Configuración regional 27](#_Toc293370339)

[AcceptHeaderLocaleResolver 28](#_Toc293370340)

[CookieLocaleResolver 28](#_Toc293370341)

[SessionLocaleResolver 28](#_Toc293370342)

[LocaleChangeInterceptor 28](#_Toc293370343)

[Uso de temas 29](#_Toc293370344)

[Definiendo los temas 29](#_Toc293370345)

[Resolutores de tema 29](#_Toc293370346)

[Subida de ficheros 30](#_Toc293370347)

[Subiendo un fichero desde un formulario 30](#_Toc293370348)

[Manejo de Excepciones 32](#_Toc293370349)

[@ExceptionHandler 32](#_Toc293370350)

[Convenios 34](#_Toc293370351)

# ¿Qué es Spring MVC?

La arquitectura MVC (Modelo vista controlador) se basa en la separación de los datos y modelo de la aplicación (Modelo), la interfaz de usuario (comúnmente un navegador que recibe código HTML) y la interacción entre ambos, el controlador.

En una aplicación MVC, la gestión de estado, la validación y el flujo de trabajo son temas fundamentales y principal foco de atención. Debido a la naturaleza del protocolo HTTP no se dispone de estado, con lo que se dificulta la tarea.

Spring construye su parte MVC entorno al DispatcherServlet, el cual despacha las peticiones a los manejadores, con asignaciones de controlador configurables, resolutores de vistas, resolutor de la configuración local, de temas así como para la subida de ficheros. A lo largo de Spring ha habido cambios desde la definición inicial de los controladores en el fichero xml (que será lo primero que explicaremos) hasta las actuales anotaciones @Controller y @RequestMapping que ofrecen mayor flexibilidad.

En Spring se pueden utilizar también objetos command (anteriores a la versión 3) pero actualmente se pueden utilizar cualquier objeto como un command. Este objeto será útil para la extracción de información de formularios. Así con Spring no tendremos que duplicar los objetos teniendo, por ejemplo, un POJO con cadenas que trasformaremos en el objeto más complejo.

El ViewResolutor (o resolutor de vista) de Spring es extremadamente flexible, a pesar de que un controlador puede escribir directamente la respuesta, normalmente devuelve un objeto ModelAndView que contiene el nombre de la vista y los objetos del modelo. El modelo es pasado a la vista la cual puede ser JSP o Velocity.

El framework de Spring, específicamente el MVC, está diseñado para facilitar tanto la construcción de controladores, como las vistas que están asociadas así como la interacción con los objetos del modelo. Todo esto de la manera más flexible y con la posibilidad de integrar otros marcos de trabajo conocidos como Struts o JSF.

# Características de Spring MVC

* Una clara separación de roles.
* Una potente y sencilla configuración entre el marco de trabajo y las clases de la aplicación como las JavaBeans.
* Adaptabilidad, no intrusión y flexibilidad.
* Reusabilidad del código empresarial, sin necesidad de duplicado, pudiendo usar código empresarial existente como comandos o formularios reflejándolos como una clase del marco de trabajo en particular.
* Validaciones y enlaces personalizados, enlazando valores reales como fechas y números evitando la conversión de cadenas y duplicado.
* Mapeo y resolutores de vista personalizables, distintas estrategias de mapeo y de resolutores de vista que van desde la simple URL hasta sofisticadas estrategias.
* Trasferencia del modelo flexible basada en pares nombre/valor, compatible con cualquier tecnología de vista.
* Se pueden configurar tanto temas como configuración local de diversas maneras compatibles con JSP o Velocity.
* Spring también tiene una potente librería de tag que ofrece tanto el enlace de datos, temas o formularios.
* Los beans tienen un ámbito de aplicación de petición o sesión, esto no es específico de Spring pero Spring lo potencia.
* Spring MVC es compatible con otros marcos de trabajo web como Struts, WebWork etc. Si no se desea usar Spring MVC se pueden utilizar otras características de Spring e integrar otro framework que hará uso del contexto de Spring.

# DispatcherServlet

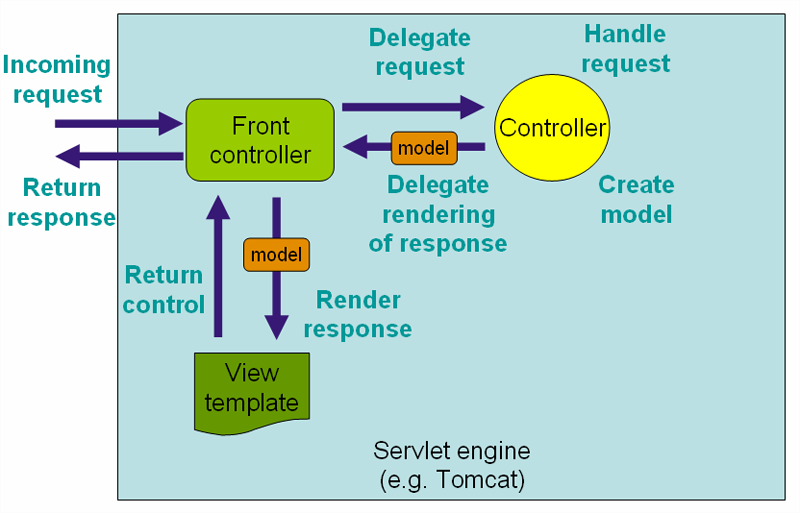
Con cada una de las peticiones que llegan desde el navegador del cliente al servidor, este tendrá que darle sentido, tanto por las distintas URL, métodos, (principalmente POST y GET) así como los parámetros, objetos JSON etc. El servidor analizará la petición y deberá decidir qué controlador será quién se haga cargo de ella. En las aplicaciones JEE comunes todo esto se define en el fichero web.xml en el que asociamos patrones de las peticiones para que se haga cargo un Servlet.

Spring con DispatcherServlet lo que hace es añadir una capa más, de manera que todas las peticiones que lleguen (y que queramos que se haga cargo el framework de Spring) llegarán a el Servlet DispatcherServlet. Éste será el encargado, tras analizar la petición, de pasar la responsabilidad de “el qué hacer” al controlador definido, es decir, a otra clase. El HandlerMapping es la clase encargada de decidir qué clase controladora está mapeada para determinada petición.

Es decir, cuando una petición sale del navegador tiene la URL solicitada y también puede tener datos adicionales como datos de un formulario, esto llegará al DispatcherServlet (siempre que cumpla el patrón que para él hayamos configurado). Este Servlet frontal lo que hará será consultar al HandlerMapping si la petición está contenida y de ser así que Controlador está asociado. Si existe coincidencia se le devolverá un controlador y un método de este el cuál ejecutará y tras su ejecución devolverá un objeto ModelAndView el cual encapsula una vista y los datos generados tras la ejecución del controlador y que deberá interpretar la vista (normalmente un JSP). Para resolver la vista Spring se valdrá de la clase ViewResolver. Al cliente le llegará el código HTML tras la ejecución e interpretación del JSP.

El framework de Spring MVC, así es como otros marcos de trabajo web en los que hay un controlador principal que atiende la llamada y la delega en un secundario. Pero es algo más que eso, ya que está integrado al contenedor de Spring y así puede disfrutar de todas las ventajas de la inyección de dependencia y de la inversión de control así como cualquier característica de Spring.

El flujo de trabajo del procesado de una petición está en el siguiente diagrama, la petición llega ofrecida por el servidor (por ejemplo Tomcat o Glassfish) es delegada al controler frontal (el DispatcherServlet) este delega en un controlador secundario que devolverá un objeto ModelAndView (que contendrá el modelo y la vista) tras su ejecución al controlador frontal que se encargará de buscar una vista, pasarle el modelo para después dar una respuesta al cliente, el navegador.



Esto ha sido una breve introducción y resumen de lo que Spring MVC hace más comúnmente, aunque cada una de las piezas comentadas realmente puede tener un comportamiento más complejo, esta introducción nos da una primera impresión de los fundamentos de éste marco de trabajo, más tarde daremos una explicación más detallada, pero de momento lo primero que debemos hacer es configurar nuestro DispatcherServlet en el contenedor de Spring.

## Configuración de DispatcherServlet y ContextLoaderListener

Como hemos dicho, DispatcherServlet será el Servlet que funciona como conector principal con los controladores de Spring. Para que esto ocurra, hemos de configurar la clase como otro Servlet más en el fichero web.xml y asignarle un patrón de URL. Si la petición coincide con el patrón definido será redireccionado a este controlador.

<servlet>

<servlet-name>helloWorldMvc</servlet-name>

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

</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>dispatcherServlet</servlet-name>

<url-pattern>\*.html</url-pattern>

</servlet-mapping>

El nombre que le demos, en este caso le hemos llamado “helloWorldMvc” es importante si lo que queremos es configurar varios, Spring te permite tener tantos como queramos. Por otro lado, por defecto Spring intentará abrir el fichero helloWorldMvc-servlet.xml.

En cuanto al patrón hemos utilizado éste (también el más utilizado en la comunidad Spring junto a \*.htm) como podríamos haber elegido cualquier otro. Bien es cierto que es una buena práctica darle extensión ya que es muy posible que quisiéramos dejar absolutamente todo en manos del Servlet de Spring.

DispatcherServlet ahora abrirá el contexto de aplicación web que hayamos definido (o el por defecto, nombre-servlet.xml). Como ya comentamos en capítulos anteriores y como seguiremos viendo más adelante, es una práctica muy buena dividir los ficheros de configuración del contexto de igual modo que las capas lógicas más significativas.

Así tendríamos cuatro ficheros dividiendo las 4 capas más importantes, la capa de seguridad, la capa de servicio, la capa de persistencia y con esta la capa web. Si dividimos los ficheros de configuración de contexto de forma coherente con la lógica de la aplicación ganaremos en reusabilidad, mantenibilidad y facilitaremos la comprensión.

De hecho, cualquiera de estos cuatro iniciales podrían ser reemplazables, es decir, imaginemos que tenemos un fichero \*-data.xml en el que hay una configuración de contexto que hace uso del motor de persistencia JPA y otro \*-data.xml con Hibernate, reemplazar uno por otro sería facilísimo y no tendría que afectar a las otras capas.

Para asegurarnos que todos los archivos de configuración se abren necesitamos configurar un cargador de contexto en el archivo web.xml. Éste será ContextLoaderListener cuya configuración en el web.xml es:

<context-param>

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

<param-value>

/WEB-INF/exampleapp-service.xml

/WEB-INF/exampleapp-data.xml

/WEB-INF/exampleapp-security.xml

</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

Dentro de la etiqueta “context-param” definimos tanto el nombre del parámetro (será contextConfigLocation) y una lista de valores que hacen referencia a cada uno de los archivos de configuración de contexto de Spring. Aquí se han definido por servicio, datos y seguridad. Aún no se han comentado la seguridad pero es conveniente que se vaya tratando este tema ya que una buena modularización es crucial.

El DispatcherServlet es una clase que hereda de HttpServlet y como tal se declara en el web.xml, por ello se le debe mapear las peticiones que queremos que maneje, este proceso es un estándar de JEE.

El WebApplicationContext es una extensión del ApplicationContext con algunas características necesarias para las aplicaciones web se diferencia de un AplicationContext normal en que es capaz de resolver temas y que sabe en que Servlet está asociado. El WebApplicationContext está ligado al ServletContext y usando métodos estáticos en la clase RequestContextUtils podemos acceder al WebApplicationContext si es que necesitamos acceder a él.

El DispatcherServlet de Spring usa beans especiales para procesar la petición y devolver una vista apropiada. Estos beans son parte del marco de trabajo de Spring y se pueden configurar en el WebApplicationContext de la misma manera que configuras cualquier otro bean. Sin embargo para la mayoría de los beans se proporcionan parámetros por defecto, con lo que no es necesario configurarlo.

Los beans especiales del WebApplicationContext son los siguientes:

* Controladores.
* Manejadores de mapeo o handlerMappings, manejan la ejecución de una lista de pre-procesos y post-procesos y controlan si son ejecutados si se cumple una determinada condición.
* Resolutores de vista o ViewResolvers, los cuales resuelven los nombres de las vistas con las vistas.
* localeResolver o resolutor de configuración local, el cual resuelve la localización del cliente para darle una vista internacionalizada.
* Resoltores de tema o ThemeResolver, resuelven un tema y apariencia, (por ejemplo dar un layout personalizado)
* MultipartFileResolver, que ofrece funcionalidades para subir ficheros desde formularios.
* HandlerExceptionResolvers, contiene funcionalidades para manejar excepciones.

# Usando Spring MVC sin anotaciones

En los últimos años el desarrollo del software se ha ido mudando de las clásicas aplicaciones de escritorio al patrón cliente servidor y de este a la aplicación web gracias al Modelo Vista Controlador.

* Para desarrollarlas con Spring básicamente necesitaremos hacer lo siguiente:
* Hacer una clase Controladora que interprete la petición del cliente y sus parámetros, manipule, busque, cree o destruya los objetos del modelo y le devuelva una respuesta.
* Configurar el controlador en el fichero \*-servlet.xml (este paso no será necesario con las anotaciones).
* Crear una página JSP la cual será la parte de la vista, es decir una plantilla para mostrar los datos y que, como ya sabemos, tras su interpretación se le enviará un .html al cliente.
* Configurar el archivo que resuelva la página.

## Creando nuestro primer Controlador

Como si de un interfaz se tratase, un controlador actúa como pieza que interconecta la aplicación con el usuario. Si el lector está habituado al uso de otros marcos de trabajo Modelo Vista Controlador como Struts verá que es bastante similar al concepto Action con la salvedad de que al ser un bean definido en el contexto Spring podrá beneficiarse de la inyección de dependencia así como de la Programación Orientada a Aspectos (AOP).

Por lo general un objeto Controlador delegará la responsabilidad de la acción a un objeto de servicio designado para tal tarea (Darse de alta, crear, borrar, etc).

Hagamos un ejemplo sencillo de lo que podría ser una página de inicio para una aplicación, Ésta página lo único que mostrará es la fecha actual.

**package** youdiversity.web;

**import** java.util.Date;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** org.springframework.web.portlet.ModelAndView;

**import** org.springframework.web.portlet.mvc.AbstractController;

/\*\*

\*

\* **@author** javiersevilla

\*/

**public** **class** WelcomeController **extends** AbstractController {

**public** WelcomeController(){

}

**protected** ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response) **throws** Exception {

**return** **new** ModelAndView("home","dateMessage",**new** Date());

}

}

Este controlador es de lo más sencillo que podemos hacer con Spring, pero sirve para ilustrarnos cómo tras ejecutar el método se devuelve un objeto ModelAndView el cual contendrá el nombre de la vista asociada (“home”) y un objeto (u objetos) que representarán el modelo.

## El objeto ModelAndView

Una vez que tengamos los resultados se han de mandar al navegador. En el controlador anterior tan sólo se devolvía un objeto de nombre dateMessage que era de tipo Date con la fecha del día de hoy.

Tal y como hemos visto, al crear un objeto ModelAndView hemos pasado en el primer parámetro el nombre de la vista asociada. El encargado de interpretar este nombre con el fichero .jsp correspondiente será el resolutor de visualización. Así la definición del controlador en el fichero \*-servlet.xml será de la siguiente manera:

Existen varios constructores del objeto ModelAndView, entre ellos se pueden encontrar el constructor vacío, con el parámetro de la vista (ya sea el nombre en String o un objeto View) y también con la vista y los objetos del modelo.

## Configuración del controlador

Para dar de alta el controlador que acabamos de escribir iremos al archivo de configuración de contexto del DispatcherServlet (como ya dijimos \*-servlet.xml siendo \* el nombre que le demos al dispatcherServlet en el fichero web.xml).

En este caso no vamos a inyectar ningún otro objeto, en ejemplos posteriores veremos que es tan sencillo como se hace con el resto de los bean.

El siguiente código muestra cómo hacerlo:

<bean class=*"org.youdiversity.web.controllers.WelcomeController"* name=*"/welcome.htm"*/>

Algo que puede sorprender es que no lleva propiedad “id” mientras que sí que lleva “name” y éste es un patrón URL. Esto es tan sencillo como que el atributo “id” no puede llevar caracteres especiales, como la barra inclinada.

Cuando llegue una petición que cumpla dicho patrón le pasará la llamada.

## View Resolvers

Las Java Servlet Pages (JSP) se han impuesto como la tecnología java para desarrollo web. La forma en la que se presentan, como un .xml y con todos los tags de una .html, hacen que sea una forma muy amigable de presentar contenido web.

Para ayudar a Spring a buscar qué JSP utilizar para cada visualización se ha de incluir un nuevo bean de tipo ViewResolver. El más sencillo es InternalResourceViewResolver cuya definición es la siguiente.

<bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"* p:prefix=*"/WEB-INF/jsp/"* p:suffix=*".jsp"*/>

Spring tiene varios ViewResolvers pero para las visualizaciones mostradas por JSP InternalResourceViewResolver es el más sencillo. Su funcionamiento es tan simple como añadir tanto el prefijo como el sufijo al nombre de la vista devuelta en el objeto ModelAndView.

Así si nuestro controller devuelve “welcome” como nombre de la vista en el objeto ModelAndView Spring buscará el archivo /WEB-INF/jsp/welcome.jsp.

## Crear un JSP

Como último paso deberíamos crear el fichero “welcome.jsp”, el siguiente código ilustra su contenido:

<%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"ISO-8859-1"*%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

<head>

<title>WELCOME</title>

</head>

<body>

${dateMessage}<br>

</body>

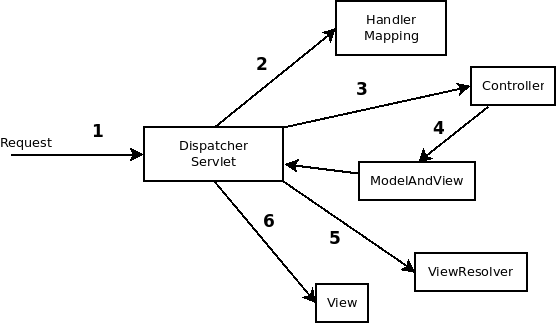
</html>

# Pasos de la petición

Ahora lo tenemos todo, el controlador, la página jsp, lo hemos registrado en el contexto de DispatcherServlet y tenemos un ViewResolver.

Pero, ¿Qué ocurre cuando se recibe una petición?, los acontecimientos son los siguientes:

1. El servidor le pasa al DispatcherServlet la petición (para ello lo configuramos y le dimos un patrón en el web.xml).
2. DispatcherServlet buscará qué controlador se encarga de la URL.
3. Nuestro controlador devuelve un ModelAndView con un nombre lógico de la vista y un objeto o un mapa de objetos con los objetos del modelo.
4. DispatcherServlet buscará en el ViewResolver la vista con el nombre lógico (en nuestro caso welcome) y devolverá la ruta completa (WEB-INF/jsp/welcome.jsp).
5. DispatcherServlet reenviará la petición al jsp.



Con esto hemos completado la manera más básica de operar con controladores sin anotaciones o con marcos de trabajo inferiores a Spring 3. En los siguientes puntos veremos todas las ventajas que la versión 3 nos aporta.

# Usando Spring MVC con Anotaciones y Con Spring 3

Los Controladores facilitan el acceso al comportamiento de la aplicación que suelen definir a través de un interfaz de servicio. Los controladores interpretan la entrada del usuario y devuelven un modelo representado en una vista. Spring implementa controladores de una manera muy abstracta que permite crear una gran variedad de controladores.

A partir de la versión 2.5, Spring introdujo una serie de anotaciones para los controladores MVC como @Controller, @RequestMapping, @ModelAttribute etc. Estas anotaciones están disponibles tanto para Servlet MVC o Portlet MVC. Los controladores que son implementados de este modo no han de extender de clases específicas como AbstractController o implementar interfaces como Controller. Tampoco tienen dependencias con las API de Servlet o Portlet aunque se pueden configurar fácilmente el acceso a ellas.

**package** org.youdiversity.web.controllers;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.servlet.ModelAndView;

@Controller

**public** **class** HelloWorldController {

@RequestMapping("/helloWorld")

**public** ModelAndView helloWorld() {

ModelAndView mav = **new** ModelAndView();

mav.setViewName("helloWorld");

mav.addObject("message", "Hello World!");

**return** mav;

}

}

Como se puede ver, las anotaciones @Controller y @RequestMapping permiten una gran flexibilidad de métodos y firmas. En este ejemplo no hay parámetros y devuelve un ModelAndView. Existen múltiples estrategias como se explicarán más adelante en este capítulo. Con ModelAndView, @Controller y @RequestMapping tenemos los pilares básicos para construir con Spring MVC.

## Definir el controlador en el fichero \*-servlet.xml.

La anotación @Controller indica que una clase en particular sirve como un controlador. Spring como ya hemos dicho, no necesita que extiendas de ninguna implementación de Controller.

El dispatcher escanea en búsqueda de clases que tengan métodos mapeados y detecta anotaciones @RequestMapping y @Controller.

Se pueden definir estos controladores directamente en el contexto del dispatcher, sin embargo la anotación @Controller permite que se autodetecte. Para ello tendremos que incluir el siguiente tag en fichero de configuración de contexto:

<context:component-scan base-package=*"org.youdiversity.web.controllers"*/>

## Mapeando peticiones con @RequestMapping

La anotación @RequestMapping se usa para mapear URL ya sea en una clase entera o en un método en particular. Normalmente si mapeamos a nivel de clase se usarán un método u otro dependiendo del método de petición (GET o POST) o los parámetros de la petición.

El siguiente código muestra las cabeceras de dos métodos el los cuáles uno será llamado cuando se hagan peticiones POST y la segunda cuando sean GET:

@RequestMapping(method = RequestMethod.*POST*)

**public** String processSubmit(@ModelAttribute AdminDataPicker …

…

@RequestMapping(method = RequestMethod.*GET*)

**public** String setupForm(Model model) {

…

En el ejemplo el @RequestMapping es usado para cada método. Lo que hará que se diferencien será tanto los parámetros de la petición, si hay sub-URL o si la petición es GET o POST.

El @RequestMapping a nivel de clase no es necesario, si no lo hacemos cada uno de los métodos será a nivel absoluto. Como en el siguiente ejemplo:

**package** org.youdiversity.web.controllers;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.servlet.ModelAndView;

@Controller

**public** **class** HelloWorldController {

@RequestMapping("/helloWorld")

**public** ModelAndView helloWorld() {

ModelAndView mav = **new** ModelAndView();

mav.setViewName("helloWorld");

mav.addObject("message", "Hello World!");

**return** mav;

}

}

### Peticiones URL, usando @RequestMapping

Para acceder a partes de la petición URL se pueden usar plantillas del modo “http://www.ejemplo.com/usuarios/546” con @ReuestMapping lo mapearemos del siguiente modo @RequestMapping(value=“/usuarios/{usuarioId}”). Usaremos @PathVariable para designar cada uno de las valores contenidos en la URL. El siguiente ejemplo muestra este uso:

@RequestMapping("/user/{userId}")

**public** ModelAndView userHandler (@PathVariable("userId") Long userId) {

ModelAndView mav = **new** ModelAndView("user/profile");

mav.addObject(userDao.get(userId));

**return** mav;

}

De este modo si recibiésemos una petición /user/12 el número 12 sería una variable de path que en este caso nos servirá como en id del usuario.

Sólo si se ha compilado en modo debug no se necesitará la anotación @PathVariable para enlazar con el valor de la variable.

También se pueden usar varias variables

@RequestMapping("/user/{userId}/{subjectId")

**public** ModelAndView userHandler (@PathVariable("userId") Long userId,

@PathVariable("subjectId") Long subjectId) {

...

**return** mav;

}

O incluso definir una a nivel global y otras a nivel particular

@Controller

@RequestMapping("/user/{userId}")

**public** **class** UserProfileController {

**private** UserDao userDao;

@Autowired

**public** UserProfileController(UserDao userDao) {

**this**.userDao = userDao;

}

@RequestMapping("/subject/{subjectId}")

**public** ModelAndView userHandler (@PathVariable("userId") Long userId,

@PathVariable("subjectId") Long subjectId) {

...

**return** mav;

}

}

### Mapeo avanzado con @RequestMapping

Además de las URL la anotación @RequestMapping soporta expresiones de direcciones, como al estilo de Ant (por ejemplo: /users/\*/subject/{subjectId}). Los resolutores de direcciones MethodNameResolver y PathMethodNameResolver buscaran primero rutas explícitas y tras descartar cada una de ellas las rutas que tengan una expresión como la citada serán las que queden por defecto si no ha habido una coincidencia anterior.

Si se tienen un único método por defecto (sin un mapeo específico), entonces todas las peticiones sin métodos con mapeos más específicos serán servidas a él.

Se pueden especificar que el método está esperando determinados parámetros que han de estar o no estar, es decir, @RequestMapping (value = "/subject/{subjectd}", params = "myParam=myValue"). Así este método sólo será mapeado si además de cumplir el path de la URL también tenga el parámetro “myParam” y con el valor “myValue”. Para indicar lo contrario sería indicar “!myParam”, así ese método nunca será seleccionado si encuentra el parámetro “myParam”.

De igual forma, los mapeos pueden ser configurados teniendo en cuenta también el contenido de la cabecera.

@RequestMapping(value="/subject/{subjectId}",headers = "content-type=text/\*")

**public** ModelAndView userHandler(@PathVariable("userId") Long userId,

@PathVariable("subjectId") Long subjectId) {

...

**return** mav;

}

El método anterior sólo es ejecutado cuando el contenido es text/\* como por ejemplo text/html.

#### Argumentos de método soportados

Los métodos anotados con @RequestMapping pueden tener una firma muy flexible. Muchos de ellos pueden ser usados de modo arbitrario.

Se pueden elegir distintas implementaciones del objeto Request o Response del API de Servlet como ServletRequest o HTTPServletRequest.

Que aparezca un objeto de sesión del API de Servlet fuerza la presencia de una sesión. Con lo que nunca tendrá un valor nulo.

Los objetos org.springframework.web.context.request.WebRequest o org.springframework.web.context.request.NativeWebRequest Te permiten el acceso genérico a un parámetro de la petición así como el acceso a los atributos Request/Session sin que haya vínculos al API nativo Servlet/Portlet.

java.util.Locale es determinado según el mayor resolutor de locale que esté disponible, de hecho, el LocaleResolver es un Servlet.

Se puede acceder a la petición con java.io.InputStream o con java.io.Reader de igual manera que con el API Servlet.

También se puede generar una respuesta con java.io.OutputStream así como con java.io.Writer como usásemos el API de Servlet.

El objeto java.security.Principal contiene el usuario autenticado.

Los parámetros de acceso de la anotación @PathVariable obtienen el acceso a las variables de la URI.

Con la anotación @RequestParam se hace referencia a los parámetros de la petición, es decir a los parámetros específicos de la petición de la Request. Los parámetros son convertidos en el tipo declarado en el método. Más tarde haremos referencia a como enlazar (o hacer binding) estos valores.

La anotación @RequestHeader se utiliza para acceder a los parámetros de la cabecera de la petición. Los valores de los parámetros serán convertidos en los tipos declarados.

La anotación @RequestBody se utiliza para anotar parámetros que se hallen en el cuerpo de la petición. Los valores serán convertidos en los tipos declarados usando HttpMessageConverteres. Más tarde veremos cómo hacerlo.

Con el parámetro HttpEntity<?> podremos acceder a las cabeceras HTTP y su contenido de la petición. El flujo de la petición será convertido en la entidad body usando HttpMessageConverteres. También lo veremos más tarde.

Podemos usar java.util.Map, org.springframework.ui.Model o org.springframework.ui.ModelMap para guardar los objetos de modelo.

Los objetos Command o Form los podemos enlazar con parámetros como propiedades de los beans, campos, conversiones con algún tipo personalizado, dependiendo de los métodos de @InitBinder y de la configuración del HandlerAdapter.

Con org.springframework.validation.Errors o org.springframework.validation.BindingResult se valida un comando o formulario.

Con org.springframework.web.bind.support.SessionStatus se maneja el estado para hacer el proceso de un formulario completo. El cual desencadena la limpieza de los atributos de la sesión que se han quedado indicados por la anotación @SessionAttributes.

Los parámetros de errores (Errors) o de resultado del enlace (BindingResult) han de seguir el modelo del objeto que ha sido enlazado inmediatamente. Es decir, si se enlaza un modelo, y un @ModelAttribute en la firma del método no podremos poner un único BindingResult, sino que tendremos que poner uno a continuación de cada uno de los objetos enlazados.

### Tipos devueltos soportados

El objeto ModelAndView en el que se haya el modelo con los objetos y los resultados de la referencia a los datos de la anotación @ModelAttribute.

El objeto Model, en este caso el nombre de la vista será determinado implícitamente a través RequestToViewNameTranslator y el objeto Model, de igual forma a ModelAndView, contendrá los objetos y resultados.

Un mapa (implementación de Map), que representará los objetos del modelo. De igual forma cuando se devuelve un Model la vista será determinada por RequestToViewNameTranslator.

Una vista, es decir, un objeto View. El modelo estará implícitamente determinado por los objetos del comando y los objetos con anotaciones @ModelAttribute. De igual forma, los objetos de comando estarán implícitos como modelo.

Una cadena que representa la vista lógica. De igual modo, tendrá implícito el modelo con los objetos de comando y los objetos anotados con @ModelAttribute.

Nada, es decir un método con firma void. En este caso el método manejará la petición por sí mismo, es decir, escribiendo la respuesta directamente con contenido, declarando un argumento de tipo ServletResponse o HttpResponse o si el nombre de la vista se ha de deducir a través del objeto RequestToViewNameTranslator.

Si el método está anotado con @ResponseBody, el tipo a devolver tendrá que estar escrito en el cuerpo de la respuesta HTTP. El valor del tipo devuelto será convertido en el tipo del argumento del método declarado usando HttpMessageConverters.

Una entidad HttpEntity<?> o ResponseEntity<?>, objeto que proporciona acceso a las cabeceras de respuesta HTTP y a sus contenidos. El cuerpo de la entidad será convertida en el flujo de respuesta usando HttpMessageConverters.

Cualquier otro tipo devuelto es considerado un único objeto de modelo. De este modo, como pasa de otros muchos, se le añadirán los objetos command así como los anotados con @ModelAttribute y la vista será determinada.

## Enlazando parámetros de la petición a parámetros de métodos con @RequestParam

Se usa @RequestParam para enlazar los parámetros de la petición con los parámetros de los métodos del controlador.

El siguiente código muestra un ejemplo:

@Controller

@SessionAttributes(types = AdminDataPicker.**class**)

@RequestMapping("/public/createAdmin")

**public** **class** CreateAdminController {

@Autowired

**private** AdminDao adminDao;

@RequestMapping(method = RequestMethod.*GET*)

**public** String setupForm(Model model) {

AdminDataPicker adminDataPicker = **new** AdminDataPicker();

model.addAttribute("adminDataPicker", adminDataPicker);

**this**.refreshData(model);

**return** "public/createAdmin";

}

**private** **void** refreshData(Model model) {

}

@RequestMapping(method = RequestMethod.*POST*)

**public** String processSubmit(

@ModelAttribute AdminDataPicker adminDataPicker, BindingResult result, SessionStatus status, Model model) {

**new** AdminValidator().validate(adminDataPicker, result, **this**.adminDao);

**if** (result.hasErrors()) {

**this**.refreshData(model);

**return** "public/createAdmin";

} **else** {

Admin user = **this**.adminDao.saveFromDataPicker(adminDataPicker);

status.setComplete();

**return** "redirect:/userProfile/" + user.getId();

}

}

}

Los parámetros que usan esta anotación se requieren por defecto, si queremos que sean opcionales tendremos que añadir el atributo “required” con un valor “false”.

## @RequestBody

La anotación @RequestBody cuando se usa sobre un parámetro del método indica a Spring que ese parámetro del método ha de ser enlazado con el valor de la petición HTTP. Por ejemplo:

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

public void handle(@RequestBody String body, Writer writer) throws IOException {

writer.write(body);

}

Se puede convertir el cuerpo de la petición usando HttpMessageConverter. Este objeto también convierte el mensaje de solicitud a un objeto de respuesta. DispatcherServlet soporta el procesamiento de la anotación.

## 

## @ResponseBody

La anotación @ResponseBody es la análoga de la respuesta de @RequestBody. En este caso puesto sobre un método indica a Spring que el valor de retorno del método será una cadena con el cuerpo de la respuesta.

@RequestMapping("answerSingleQuestion")

@ResponseBody

**public** String addAnswer(@RequestParam Long questionId,

@RequestParam(required = **false**) List<String> answersId) {

List<Long> answersIdLongs = **new** ArrayList<Long>();

**if** (answersId != **null**) {

**for** (String answerId : answersId) {

answersIdLongs.add(**new** Long(answerId));

}

}

QuestionAnswered questionAnswered =

**this**.questionDao.saveQuestionAnsweredByUser(questionId, answersIdLongs);

**return** questionAnswered.getCorrect() ?

"Pregunta contestada correctamente" :

"Pregunta no contestada correctamente";

}

Spring convertirá la cadena en una respuesta HTTP usando el conversor HttpMessageConverter.

## HttpEntity

Similar a @RequestBody o @ResponseBody, el HttpEntity tiene acceso al cuerpo de la petición o de la respuesta y permite acceder a las cabeceras de las peticiones o de las respuestas.

El siguiente método es un ejemplo:

@RequestMapping("/something")

public ResponseEntity<String> handle(HttpEntity<byte[]> requestEntity) throws UnsupportedEncodingException {

String requestHeader = requestEntity.getHeaders().getFirst("MyRequestHeader"));

byte[] requestBody = requestEntity.getBody();

HttpHeaders responseHeaders = new HttpHeaders();

responseHeaders.set("MyResponseHeader", "MyValue");

return new ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREATED);

## 

## Proporcionando un enlace a los datos del modelo con @ModelAttribute

Se usa @ModelAttribute cuando lo pones en un parámetro del método indicando que pertenece a un atributo del modelo o cuando lo pones sobre un método indicando que el valor devuelto pertenece a un objeto de modelo (comúnmente para iniciarlo).

Para iniciarlo:

@ModelAttribute("types")

**public** Collection<RoleType> populateRoleTypes() {

**return** RoleType.values();

}

O como hemos dicho, para enlazarlo con un parámetro del método:

@RequestMapping(method = RequestMethod.*POST*)

**public** String processSubmit(

@ModelAttribute AdminDataPicker adminDataPicker,

BindingResult result, SessionStatus status, Model model) {

**new** AdminValidator().validate(

adminDataPicker, result, **this**.adminDao);

**if** (result.hasErrors()) {

**this**.refreshData(model);

**return** "public/createAdmin";

} **else** {

Admin user = **this**.adminDao

.saveFromDataPicker(adminDataPicker);

status.setComplete();

**return** "redirect:/userProfile/" + user.getId();

}

## }

## Especificando atributos para almacenar en la sesión

La anotación @SessionAttributes indica que atributos de sesión son usados por un manejador en particular. Esto lista una serie de nombres de atributos del modelo que se almacenaran como atributos de sesión.

@Controller

@SessionAttributes(types = TeacherDataPicker.**class**)

@RequestMapping("/admin/createTeacher")

**public** **class** CreateTeacherController {

## 

## Uso de cookies con @CookieValue

La anotación @CookieValue permite enlazar una cookie HTTP con un método. El siguiente código muestra cómo hacerlo:

@RequestMapping("/displayHeaderInfo.do")

public void displayHeaderInfo(@CookieValue("JSESSIONID") String cookie) {

//...

}

## Mapeando cabeceras de peticiones con @RequestHeader

La anotación @RequestHeader permite enlazar los parámetros de los métodos con la cabecera de la petición.

Dada la siguiente cabecera:

Host localhost:8080

Accept text/html,application/xhtml+xml,application/xml;q=0.9

Accept-Language fr,en-gb;q=0.7,en;q=0.3

Accept-Encoding gzip,deflate

Accept-Charset ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive 300

Un ejemplo sería el siguiente:

@RequestMapping("/displayHeaderInfo.do")

public void displayHeaderInfo(@RequestHeader("Accept-Encoding") String encoding,

@RequestHeader("Keep-Alive") long keepAlive) {

//...

}

## Personalizando el enlace a Datos

Se pueden usar personalizaciones de enlace tanto con una clase WebBindingInitializer como con métodos exclusivos para cada controlador gracias a la anotación @InitBinder.

El siguiente método es un ejemplo

@InitBinder

**public** **void** initBinder(HttpServletRequest request, ServletRequestDataBinder binder)

**throws** Exception {

binder.registerCustomEditor(

Carrer.**class**, **new** CarrerCustomEditor(**this**.carrerDao));

binder.registerCustomEditor(

Subject.**class**, **new** SubjectCustomEditor(**this**.subjectDao));

SimpleDateFormat dateFormat =

**new** SimpleDateFormat("dd/MM/yyyy");

dateFormat.setLenient(**false**);

binder.registerCustomEditor(

Date.**class**, **new** CustomDateEditor(dateFormat, **false**));

}

Para externalizar y normalizar el enlace de datos a nivel de aplicación implementaremos una clase de WebBindingInitializer.

<bean class=*"org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter"*>

<property name=*"webBindingInitializer"*>

<bean class=*"com.campus.web.binding.CampusBindingInitializer"*/>

</property>

</bean>

# Mapeo de controladores

Como hemos visto, en las versiones anteriores a Spring 2.5, era necesario definir HandlerMapping en el contexto de la aplicación web. Ahora el DispatcherServlet busca anotaciones @RequestMapping en los controladores. Hay diversas propiedades que están inicializadas por defecto y que para modificarlas tendremos que sobrescribir como puedan ser interceptores, manejadores por defecto, orden de los mapeos, si queremos usar expresiones o rutas completas … entre otras.

<beans>

<bean id="handlerMapping" class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">

<property name="interceptors">

<bean class="example.MyInterceptor"/>

</property>

</bean>

<beans>

## Interceptando peticiones con HandlerInterceptor

El mecanismo de mapeo incluye interceptores que son útiles cuando se desea aplicar una cierta funcionalidad a determinadas peticiones, por ejemplo, chequeando la principal.

Los interceptores localizados en el manejador del mapeo deben implementar HandlerInterceptor. Este interfaz define tres métodos, preHandle, postHandle y afterCompletion. El primero es llamado antes de que se ejecute el manejador actual, el segundo después y el último después de que la petición se haya terminado. Estos métodos otorgan completa flexibilidad para procesar cualquier tipo de preproceso y postproceso.

El método preHandle devuelve un valor booleano, dependiendo del valor devuelto se continuará o cancelará la ejecución del controlador. Si el valor devuelto es falso el DispatcherServlet asumirá que el propio interceptor ha redireccionado y tomado control de la petición.

# Resolver vistas

Todos los marcos de trabajo MVC otorgan maneras de direccionar las vistas. Spring proporciona resolutores de vistas los cuales permiten enlazar el modelo sin atarse a ninguna tecnología de vista específica. Spring permite usar vistas JSP, Velocity o XSLT.

ViewResolver y View son los interfaces que Spring utiliza.

## ViewResolver

Como hemos comentado, todos los métodos en los controladores de Spring MVC deben resolver con el nombre de una vista lógica, de manera implícita o explícita. Las vistas de Spring son mapeadas con una vista lógica resuelta por el resolutor de vista. Spring proporciona los siguientes resolutores:

* AbstractCachingViewResolver
* XmlViewResolver
* ResourceBundleViewResolver
* UrlBasedViewResolver
* InternalResourceViewResolver
* VelocityViewResolver / FreeMarkerViewResolver
* ContentNegotiatingViewResolver

Con UrlBasedViewResolver las traducciones de resolución de vista se hacen desde una URL

<bean id=*"viewResolver"*

class=*"org.springframework.web.servlet.view.UrlBasedViewResolver"*>

<property name=*"viewClass"*

value=*"org.springframework.web.servlet.view.JstlView"*/>

<property name=*"prefix"* value=*"/WEB-INF/jsp/"*/>

<property name=*"suffix"* value=*".jsp"*/>

</bean>

Cuando se devuelve ejemplo como vista lógica se traducirá el fichero físico /WEB-INF/jsp/ejemplo.jsp.

Cuando se combinan varias tecnologías de vista se pueden usar ResourceBundleViewResolver del siguiente modo:

<bean id=*"viewResolver"*

class=*"org.springframework.web.servlet.view.ResourceBundleViewResolver"*>

<property name=*"basename"* value=*"views"*/>

<property name=*"defaultParentView"* value=*"parentView"*/>

</bean>

## Encadenando ViewResolvers

Spring permite tener múltiples resolutores de vista encadenados. Se puede especificar el orden con el atributo order, a mayor valor de este más tarde se posicionará.

## Redireccionando a vistas

Como hemos dicho un controlador devuelve un nombre de vista lógica. Para tecnologías de vista como JSP que se procesan a través de motores JSP o Servlet, esta resolución está típicamente manejada a través de la combinación de InternalResourceViewResolver e InternalResourceView los cuales redireccionan o incluyen a través del API de Servlet con los métodos RequestDispatcher.forward() o RequestDispatcher.include().

En ocasiones es deseable una tarea de redirección HTTP de vuelta al cliente antes de devolver la vista. Esto es deseable cuando un controlador ha sido llamado con datos enviados con POST y la respuesta es en verdad una delegación de otro controlador que también ve los mismos datos enviados con POST. Esto es potencialmente problemático si puede confundir con otros datos esperados también enviados por POST. Otra razón por la que se redirige antes de enviar a la vista es para eliminar la posibilidad de que el usuario haga un submit múltiple, es decir, un envío de un formulario varias veces. En este escenario, el navegador enviará un primer POST con datos entonces el recibirá una respuesta con una redirección a una URL distinta, finalmente el navegador ejecutará una redirección GET a la URL contenida en la respuesta de redirección. Así, desde la perspectiva del navegador, la página actual no refleja el resultado de un POST sino de un GET. El efecto final es que no hay manera que el usuario pueda accidentalmente mandar los mismos datos en múltiples ocasiones. El refresco fuerza un GET del resultado de la página, no un re-envío de los datos por POST.

### RedirectView

Una manera de forzar una redirección como resultado de la respuesta de un controlador es crear y devolver una instancia de RedirectView, en este caso, DispatcherServlet no hará uso del mecanismo normal de resolución de vista. En cambio, como ya se ha dado la redirección a la vista el DispatcherServlet simplemente deja que la vista haga su trabajo.

La tarea de RedirectView es una llamada al método de redirección HttpServletResponse.sendRedirect(), el cual devuelve al navegador una redirección HTTP. Todos los atributos del modelo serán parámetros de consulta HTTP, esto significa que el modelo debe contener solo objetos (Cadenas o valores convertibles a cadenas), los cuales han de estar preparados para ser convertidos en parámetros textuales de una consulta HTTP.

### El prefijo “redirect:”

Aunque el uso de RedirectView funciona bien, si es controlador se crea en RedirectView, no hay manera de evitar que el controlador sea consciente de que está pasando una redirección. Esto no es del todo óptimo, el controlador no debería preocuparse de que la respuesta sea controlada. En general sólo se debería operar en términos de los nombres de las vistas.

El prefijo especial redirect: permite lograr esto. Si un nombre de vista es devuelto con un prefijo redirect: el UrlBasedViewResolver reconocerá este aspecto especial como una redirección. El resto del nombre de la vista será tratado como la redirección URL.

El efecto final es como si el controlador devolviese una redirección RedirectView, pero ahora el mismo controlador puede operar en términos de nombre de vista lógica. Lo importante de esto es que el mismo controlador no sabe que la redirección está pasando.

### El prefijo “forward:”

También existe la posibilidad de usar un prefijo especial forward: para los nombres de vistas que serán resueltas por UrlBasedViewResolver. Esto crea un InternalResourceView, el cual ejecuta un RequestDispatcher.forward(). Este prefijo no es útil con InternalResourceViewResolver y InternalResourceView pero puede ser útil cuando se está usando otro tipo de tecnología para la vista pero se quiere forzar un forward a un recurso controlado por un motor Servlet o JSP.

Como con redirect: el prefijo forward: es inyectado en el controlador, el controlador no detecta que nada especial esté pasando en términos de manejar la respuesta.

## ContentNegotiatingViewResolver

La clase ContentNegotiatingViewResolver no resuelve las vistas ella misma, en su lugar delega a otros resolutores seleccionando la vista que se asemeja a la representación de la petición del cliente. Dos estrategias son llevadas a cabo:

Usar una URI distinta para cada recurso, normalmente usando una extensión distinta. Por ejemplo la URI, <http://ejemplo.org/usuarios.pdf> y <http://ejemplo.org/usuarios.xml>, uno pediría una representación PDF de los usuarios y la otra una representación XML.

Para permitir múltiples representaciones de un recurso Spring dispone del ContentNegotiantingViewResolver el cual resuelve una vista basada en la extensión del fichero o de la cabecera de la petición HTTP. ContentNegotiantingViewResolver no interpreta la resolución de la vista sino que delega a una lista de resolutores de vista que especificarán a través de la propiedad ViewResolver.

La clase ContentNegotiatingViewResolver selecciona un resolutor de vista comparando el tipo de la petición (content-type) con los distintos ViewResolvers. El primer ViewResolver que sea compatible con el content-type devolverá la representación al cliente. Si no hubiese una coincidencia de contenido por los distintos ViewResolvers se consultará la propiedad DefaultViews. Esta última opción es apropiada para vistas singleton que hará una representación adecuada de los recursos sin importar el nombre de vista lógico. Se pueden incluir caracteres comodín, como por ejemplo text/\* en el que el content-type podrá ser text/html, text/xml etc.

Para ayudar a la resolución de la vista basada en extensiones de ficheros se usa la propiedad mediaTypes en el bean ContentNegotiantingViewResolver para especificar un mapeo de extensiones de content-type.

<bean class=*"org.springframework.web.servlet.view.ContentNegotiatingViewResolver"*>

<property name=*"mediaTypes"*>

<map>

<entry key=*"atom"* value=*"application/atom+xml"*/>

<entry key=*"html"* value=*"text/html"*/>

<entry key=*"json"* value=*"application/json"*/>

</map>

</property>

<property name=*"viewResolvers"*>

<list>

<bean class=*"org.springframework.web.servlet.view.BeanNameViewResolver"*/>

<bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"* value=*"/WEB-INF/jsp/"*/>

<property name=*"suffix"* value=*".jsp"*/>

</bean>

</list>

</property>

<property name=*"defaultViews"*>

<list>

<bean

class=*"org.springframework.web.servlet.view.json.MappingJacksonJsonView"*/>

</list>

</property>

</bean>

<bean id=*"content"*

class=*"com.springsource.samples.rest.SampleContentAtomView"*/>

El InternalRessourceViewResolver maneja la traducción de la visa en páginas JSP, mientras que el BeanNameViewResolver devuelve una vista basada en el nombre del bean. En este ejemplo el content bean es una clase que hereda de AbstractAtomFeedView la cual devuelve un Atom feed RSS.

En la siguiente configuración si una configuración es creada con la extensión .html el resolutor de vista buscará una vista cuyo media type coincida con text/html. El InternalResourceViewResolver otorga la coincidencia para la vista text/html. Si la petición es creada con una extensión .atom, el resolutor de vista buscará una vista que coincida con el media type application/atom+xml. Esta vista será proporcionada por el resolutor BeanNameViewREsolver que mapea SampleContentAtomView como si el nombre de la vista devuelto fuera el contenido. Si la petición es creada por una extensión .json la instancia del bean MappingJacksonJsonViewserá elegido ya que es la opción por defecto. Por otro lado, las solicitudes se pueden hacer sin extensión del archivo pero con la cabecera Accept definida con un media-type, en tal caso se hará la misma resolución.

# Configuración regional

La mayor parte de la arquitectura de Spring soporta internacionalización, incluida la parte Spring MVC. DispatcherServlet permite resolver mensajes de manera automática usando el locale del cliente gracias a los objetos LocaleResolver.

Cuando llega una petición el DispatcherServlet busca un resolutor local, si se encuentra uno se intentará usar para asignar una configuración regional. Mediante el método RequestContext.getLocale() se pueden obtener el Locale que fue resuelto por el LocaleResolver.

A parte de la configuración automática de resolución de Locale, también se puede configurar un interceptor para que maneje el mapeo para cambiar el Locale en determinadas circunstancias, como por ejemplo, como parte de la URI, o por parámetros.

Los resolutores de configuración regional y los interceptores están definidos en el paquete org.springframework.web.wervlet.i18n.

## AcceptHeaderLocaleResolver

Este resolutor busca en el parámetro de la cabecera de la petición accept-language que fue enviada por el cliente. Normalmente este parámetro contiene el Locale del sistema operativo.

## CookieLocaleResolver

Este resolutor busca en una cookie que debe existir en el cliente para ver si hay un Locale específico. Si lo hay, se usará ese locale específico. Se puede definir el nombre de la cookie, su vida así como aplicarle un path el cual será visible para una parte específica de la aplicación (/ para toda la aplicación).

## SessionLocaleResolver

El SessionLocaleResolver permite recibir Locales desde la sesión que quizá fue asociada con una petición del usuario.

## LocaleChangeInterceptor

Se puede activar la opción de poder cambiar los Locale añadiendo LocaleChangeInterceptor a uno de los manejadores de mapeo. Eso hará que detecte un parámetro en la petición y cambie el Locale. Se llamará a setLocale del objeto LocaleResolver que está en el contexto. El siguiente ejemplo muestra cómo las llamadas a los recursos \*.view contienen un parámetro que se llama siteLanguage que cambiará el Locale.

<bean id=*"localeChangeInterceptor"*

class=*"org.springframework.web.servlet.i18n.LocaleChangeInterceptor"*>

<property name=*"paramName"* value=*"siteLanguage"*/>

</bean>

<bean id=*"localeResolver"*

class=*"org.springframework.web.servlet.i18n.CookieLocaleResolver"*/>

<bean id=*"urlMapping"*

class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>

<property name=*"interceptors"*>

<list>

<ref bean=*"localeChangeInterceptor"*/>

</list>

</property>

<property name=*"mappings"*>

<value>/\*\*/\*.view=someController</value>

</property>

</bean>

# Uso de temas

Se pueden aplicar temas al marco de trabajo de Spring MVC para el conjunto del look-and-feel de una aplicación. La resolución del tema es muy similar a la resolución del Locale. Un tema es una colección de recursos estáticos, hojas de estilo e imágenes que afectan al aspecto visual de la aplicación.

## Definiendo los temas

Para usar temas en la aplicación deberemos configurar una implementación del interfaz ThemeSource. El contexto de aplicación web implementa ThemeSource, pero delega sus responsabilidades a una implementación dedicada. Por defecto, esta será ResourceBundleThemeSource que leerá los ficheros de propiedades de la raíz del classpath. Se podrán usar implementaciones personalizadas de ThemeSource. Cuando se quiere configurar ResourceBundleThemeSource o se quiere definir otro específico se ha de hacer en el contexto de la aplicación. Si queremos usar ResourceBundleThemeSource habrá que definir un par de propiedades en un fichero.

Luego utilizaremos el tag <spring:theme en los ficheros JSP que es muy similar al tag <spring:message.

...

<link rel="stylesheet" href="<spring:theme code='styleSheet'/>" type="text/css"/>

...

<body style="background=<spring:theme code='background'/>">

...

Por defecto ResourceBundleThemeSource usa un nombre base vacío como prefijo. Así, el fichero de properties es cargado desde la raíz del classpath. ResourceBundleThemeSource utiliza el mecanismo de carga del paquete estándar de Java para leer los recursos, siendo así también internacionalizable.

## Resolutores de tema

Después de definir los temas hay que elegir cuál de ellos usar. El DispatcherServlet buscará un bean llamado themeResolver para saber que implementación de ThemeResolver se usará. ThemeResolver funciona de manera muy similar a LocaleResolver, se detecta que tema se usará en una petición en particular y también se puede cambiar el tema. Spring ofrece los siguientes resolutores de tema:

* FixedThemeResolver, Selecciona un tema fijo usando la propiedad defaultThemeName.
* SessionThemeResolver, El tema es mantenido en la sesión HTTP del usuario el cuál permanecerá toda la sesión, pero no estará persistida entre sesiones.
* CookieThemeResolver, El tema se guarda en una cookie en el navegador del cliente.

Spring también tiene la clase ThemeChangeInterceptor que permite los cambios de tema en cada una de las peticiones con un único parámetro.

# Subida de ficheros

Spring permite la subida de ficheros multipart. Se puede activar esta característica con el objeto MultipartResolver. Spring proporciona un resolutor multiparte para usar Commons FileUpload de Apache.

El siguiente ejemplo muestra cómo usar CommonsMultipartResolver:

<bean id=*"multipartResolver"* class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>

<property name=*"maxUploadSize"* value=*"100000"*/>

</bean>

Por supuesto también se necesita poner los .jar necesarios en el classpath para que el resolutor multipart funcione, es decir, el fichero commons-fileupload.jar.

Cuando el DispatcherServlet de Spring detecta una petición multi-part activa el resolutor que haya sido declarado en el contexto. El resolutor entonces envuelve la petición HTTP actual en una MultipartHttpServletRequest que permite la subida por partes de ficheros.

## Subiendo un fichero desde un formulario

Después de que el resolutor complete su trabajo, la petición se procesa como si hubiese sido cualquier otra. Primero hay que crear un formulario con un input de tipo file. Habrá que definir el atributo enctype con el valor “multipart/form-data”, eso permitirá al navegador saber cómo ha de enviar el fichero. El siguiente ejemplo muestra cómo hacerlo:

<html>

<head>

<title>Upload a file please</title>

</head>

<body>

<h1>Please upload a file</h1>

<form method="post" action="/form" enctype="multipart/form-data">

<input type="text" name="name"/>

<input type="file" name="file"/>

<input type="submit"/>

</form>

</body>

</html>

El siguiente paso será crear un controlador que maneje la petición de la subida del fichero. Este controlador es muy similar a cualquier otro, con la salvedad que usa MultipartHttpServletRequest o MultipartFile como parámetros de método. El siguiente código muestra cómo hacerlo:

@Controller

public class FileUpoadController {

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

public String handleFormUpload(@RequestParam("name") String name,

@RequestParam("file") MultipartFile file) {

if (!file.isEmpty()) {

byte[] bytes = file.getBytes();

// store the bytes somewhere

return "redirect:uploadSuccess";

} else {

return "redirect:uploadFailure";

}

}

}

Cabe destacar cómo los atributos anotados con @RequestParam son los elementos input del fichero.

Finalmente, se tendrán que declarar el controlador y su resolutor en el contexto de la aplicación.

<beans>

<bean id="multipartResolver"

class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>

<bean id="fileUploadController" class="examples.FileUploadController"/>

</beans>

# Manejo de Excepciones

La clase de Spring HandlerExceptionResolver se encarga de la excepción de excepciones que puedan ocurrir mientras se hace se ejecuta la petición a un controlador. HandlerExceptionResolver tiene un comportamiento similar al mapeo que se puede hacer en el descriptor xml de la aplicación web. Sin embargo, proporciona mayor flexibilidad para manejar las excepciones. Proporciona información sobre qué manejador se estaba ejecutando mientras se lanzó la excepción. Por otra parte, una manera programática de manejar excepciones te da más opciones de responder apropiadamente antes de que la petición sea pasada a otra URL.

Aunque se pueda implementar la interfaz HandlerExceptionResolver, que se implementa únicamente con el método resolverException(Exception, Handler) y devolviendo un ModelAndView. También se puede usar el SimpleMappingExceptionResolver el cual te permite resolver el nombre de la clase de la excepción que haya podido ser lanzada y mapearla con el nombre de una vista. Esta funcionalidad es equivalente al mapeo de excepciones del API de Servlet, pero también se puede implementar de una manera más ajustada mapeos de excepciones de diferentes controladores.

Por defecto, el DispatcherServlet registra DefaultHandlerExceptionResolver, este resolutor maneja una serie de excepciones estándar de Spring asignándolas un específico código de respuesta. La siguiente tabla muestra esta equivalencia:

|  |  |
| --- | --- |
| Excepción | Código de estado HTTP |
| ConversionNotSupportedException | 500 (Error interno del servidor) |
| HttpMediaTypeNotAcceptableException | 406 (No aceptable) |
| HttpMediaTypeNotSupportedException | 415 (Tipo de medio no soportado) |
| HttpMessageNotReadableException | 400 (Mala petición) |
| HttpMessageNotWritableException | 500 (Error interno de servidor) |
| HttpRequestMethodNotSupportedException | 405 (Método no soportado) |
| MissingServletRequestParameterException | 400 (Mala petición) |
| NoSuchRequestHandlingMethodException | 404 (No encontrado) |
| TypeMismatchException | 400 (Mala petición) |

## @ExceptionHandler

Otra alternativa al interfaz HandlerExceptionResolver es la anotación @ExceptionHandler. Se usa @ExceptionHandler como una anotación de método, para especificar qué método será invocado cuando una excepción en particular sea lanzada mientras la ejecución de algún método del controlador.

@Controller

public class SimpleController {

@ExceptionHandler(IOException.class)

public String handleIOException(IOException ex, HttpServletRequest request) {

return ClassUtils.getShortName(ex.getClass());

}

}

De este modo se invocará este método cuando una excepción sea lanzada.

# 

# Convenios

La mejor manera de mantener y crear un proyecto es seguir determinadas pautas y convenios que harán más sostenible la configuración necesaria, el mapeo de controladores, la resolución de vistas, las instancias ModelAndView y demás. Esto hará más consistente y mantenible el código.