Spring Framework

MVC

Rubén Gómez García

Version 1.0.0 2019-03-23

Contenidos

Capítulo 1. Introduccion

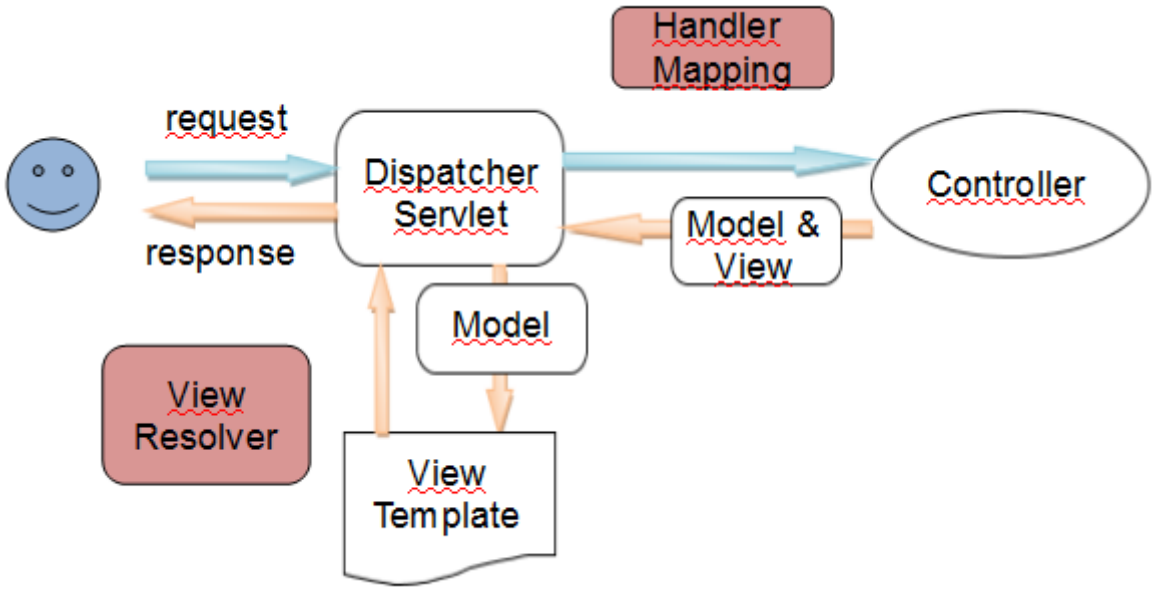
Spring MVC, como su nombre indica es un framework que implementa Modelo-Vista-Controlador, esto quiere decir que proporcionará componentes especializados en cada una de esas tareas.

Para incorporar las librerias con Maven, se añade al pom.xml

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

Capítulo 2. Arquitectura

Spring MVC, como la mayoría de frameworks MVC, se basa en el patrón **FrontController**, en este caso el componente que realiza esta tarea es **DispatcherServlet**.



2.1. DispatcherServlet

El **DispatcherServlet**, realiza las siguientes tareas.

- Consulta con los **HandlerMapping**, que **Controller** ha de resolver la petición.
- Una vez el **HandlerMapping** le retorna que **Controller** ha de invocar, lo invoca para que resuelva la petición.
- Recoge los datos del **Model** que le envía el **Controller** como respuesta y el identificador de la **View** (o la propia **View** dependerá de la implementación del **Controller**) que se empleará para mostrar dichos datos.
- Consulta a la cadena de **ViewResolver** cual es la **View** a emplear, basandose en el identificador que le ha retornado el **Controller**.
- Procesa la **View** y el resultado lo retorna como resultado de la petición.

La configuración del **DispatcherServlet** se puede realizar siguiendo dos formatos

- Con ficheros XML. Para ello se han de declarar el servlet en el **web.xml**

```
<servlet>
    <servlet-name>miApp</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/miApp-servlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>miApp</servlet-name>
    <url-pattern>/expedientesx/*</url-pattern>
</servlet-mapping>
```

De no incluir el parametro de configuracion **contextConfigLocation** para el 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** en este caso **miApp-servlet.xml**

Se puede incluir más de un fichero de configuracion de contexto, separandolos con comas.

- Con clases anotadas al estilo **JavaConfig**. Para ello el API proporciona una interface que se ha de implementar **WebApplicationInitializer** y allí se ha de registrar el servlet.

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        WebApplicationContext context = getContext();
        ServletRegistration.Dynamic dispatcher = servletContext.addServlet(
"DispatcherServlet", new DispatcherServlet(context));
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");
    }

    private AnnotationConfigWebApplicationContext getContext() {
        AnnotationConfigWebApplicationContext context = new
AnnotationConfigWebApplicationContext();
        context.setConfigLocation(this.getClass().getPackage().getName());
        return context;
    }
}
```

2.2. ContextLoaderListener

Adicionalmente, se puede definir otro contexto de Spring global a la aplicación, para ello se ha de

declarar el listener **ContextLoaderListener**, que al igual que el **DispatcherServlet** puede ser declarado de dos formas.

- Con XML

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:aplicacion.xml,
        /WEB-INF/seguridad.xml
    </param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
```

Se puede incluir más de un fichero de configuracion de contexto, separandolos con comas.

- Con JavaConfig

```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {

        WebApplicationContext context = getContext();
        servletContext.addListener(new ContextLoaderListener(context));
    }

    private AnnotationConfigWebApplicationContext getContext() {
        AnnotationConfigWebApplicationContext context = new
AnnotationConfigWebApplicationContext();
        context.setConfigLocation("expedientesx.cfg");
        return context;
    }
}
```

La clase **AnnotationConfigWebApplicationContext** es una clase capaz de descubrir y considerar los Beans declarados en clases anotadas con **@Configuration**

2.3. Namespace MVC

Se incluye el siguiente namespace con algunas etiquetas nuevas, que favorecen la configuracion del contexto

```
xmlns:mvc="http://www.springframework.org/schema/mvc"
```

2.4. ResourceHandler (Acceso a recursos directamente)

No todas las peticiones que se realizan a la aplicación necesitarán que se ejecute un **Controller**, algunas de ellas harán referencia a imagenes, hojas de estilo, . . . Se puede añadir con XML o JavaConfig

Con XML

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

Donde **mapping** hace referencia al patrón de URL de la petición y **location** al directorio dentro de **src/main/webapp** donde encontrar los recursos.

La forma de abordar esta explicación, es retomar la arquitectura y el patrón **FrontController**, y la no necesidad de un **Controller** para ofrecer un recurso estatico, los **Controller** son necesarios para los recursos dinamicos, para los estaticos introducen demasida complejidad de forma inecesaria.

Con JavaConfig, se ha de hacer extender la clase **@Configuration** de **WebMvcConfigurerAdapter** y sobrescribir el método **addResourceHandlers** con lo siguiente.

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

Donde **ResourceHandler** hace referencia al patrón de URL de la petición y **ResourceLocation** al directorio donde encontrar los recursos.

La forma de abordar esta explicación, es retomar la arquitectura y el patrón **FrontController**, y la no necesidad de un **Controller** para ofrecer un recurso estatico, los **Controller** son necesarios para los recursos dinamicos, para los estaticos introducen demasida complejidad de forma inecesaria.

2.5. Default Servlet Handler

Cuando los recursos estaticos, estan situados en la carpeta **webapp**, se pueden sustituir las configuraciones anteriores por

```
<mvc:default-servlet-handler/>
```

o

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {
    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable();
    }
}
```

2.6. ViewController (Asignar URL a View)

En ocasiones se necesita acceder a una **View** directamente, sin pasar por un controlador, para ello Srping MVC ofrece los **ViewControllers**. Se puede añadir con XML o JavaConfig

Con XML

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

Con JavaConfig, de nuevo se ha de hacer extender la clase **@Configuration** de la clase **WebMvcConfigurerAdapter**, en este caso implementando el método

```
@Override
public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("index");
}
```

En este caso **ViewController** representa el path que le llega al **DispatcherServlet** y **ViewName** el nombre de la **View** que deberá ser resuelto por un **ViewResolver**.

Los **ViewController** se resuelven posteriormente a los **Controller** anotados con **RequestMapping**, por lo que si se emplean mappings con path similares en ambos escenarios, nunca se llegará a los **ViewController**, para conseguirlo se ha de configurar la precedencia del **ViewControllerRegistry** a un valor inferior al del **RequestMappingHandlerMapping**.

Los **ViewController** no pueden acceder a elementos del Modelo definidos con **@ModelAttribute**, ya que estos son interpretados por el **RequestMappingHandlerMapping**, que no participa en el proceso de resolución de los **ViewController**

Capítulo 3. HandlerMapping

Es el primero de los componentes necesarios dentro del flujo de Spring MVC, siendo el encargado de encontrar el controlador capaz de procesar la petición recibida.

Este componente extrae de la URL un Path, que coteja con las entradas configuradas dependiendo de la implementación empleada.

Para activar los HandlerMapping unicamente hay que declararlos en el contexto de Spring como Beans.

Dado que se pueden configurar varios **HandlerMapping**, para establecer en que orden se han de emplear, existe la propiedad **Order**

El API proporciona las siguientes implementaciones

- **BeanNameUrlHandlerMapping**: Usa el nombre del Bean **Controller** como mapeo **<bean name="/inicio.htm" . . . >**, debe comenzar por /.
- **SimpleUrlHandlerMapping**: Mapea mediante propiedades **<prop key="/verClientes.htm">beanControlador</prop>**
- **ControllerClassNameHandlerMapping**: Usa el nombre de la clase asumiendo que termina en **Controller** y sustituyéndo la palabra **Controller** por **.htm**
- **DefaultAnnotationHandlerMapping**: Emplea la propiedad path de la anotación **@RequestMapping**

Las implementaciones por defecto en Spring MVC 3 son **BeanNameUrlHandlerMapping** y **DefaultAnnotationHandlerMapping**

3.1. BeanNameUrlHandlerMapping

Al emplear esta configuración, cuando lleguen peticiones con path **/helloWoorld.html**, el **Controller** que lo procesará será de tipo **EjemploAbstractController**

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

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

3.2. SimpleUrlHandlerMapping

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
    <props>
        <prop key="/helloWorld.htm">helloWorldController</prop>
    </props>
    </property>
</bean>
<bean name="helloWorldController" class="org.ejemplos.springmvc.HelloWorldController"
/>
```

3.3. ControllerClassNameHandlerMapping

```
public class HelloWorldController extend AbstractController{
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response)
            throws Exception {
        return new ModelAndView("otro");
    }
}
```

3.4. DefaultAnnotationHandlerMapping

```
@RequestMapping("helloWorld")
public class HelloWorldController extend AbstractController{
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response)
            throws Exception {
        return new ModelAndView("otro");
    }
}
```

3.5. RequestMappingHandlerMapping

Esta implementacion permite interpretar las anotaciones **@RequestMapping** en los controladores, haciendo coincidir la url, con el atributo **path** de dichas anotaciones.

```
@RequestMapping("helloWorld")
public class HelloWorldController extend AbstractController{
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response)
            throws Exception {
        return new ModelAndView("otro");
    }
}
```

El espacio de nombres **mvc**, ofrece una etiqueta que simplifica la configuracion

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

Tambien se ofrece una anotacion **@EnableWebMvc** a añadir a la clase **@Configuration** para la configuracion con JavaConfig, esta anotación, define por convencion una pila de **HandlerMapping**, ya que en realidad lo que hace es cargar la clase **WebMvcConfigurationSupport** como calse **@Configuration**, en esta clase se describen los **HandlerMapping** cargados.

```
@Configuration
@EnableWebMvc
public class ContextoGlobal {
}
```

En la ultima version de Spring no es necesario añadirlo, la unica diferencia al añadirlo, es que se consideran menos path validos para cada **@RequestMapping** definido, con ella solo **/helloWorld** y sin ella **/helloWorld**, **/helloWorld.**\ * y **/helloWorld/**

Capítulo 4. Controller

El siguiente de los componentes en el que delega el **DispatcherServlet**, será el encargado de ejecutar la logica de negocio.

Spring proporciona las siguientes implementaciones

- **AbstractController**
- **ParametrizableViewController**
- **AbstractCommandController**
- **SimpleFormController**
- **AbstractWizardFormController**
- **@Controller**

4.1. @Controller

Anotacion de Clase, que permite indicar que una clase contiene funcionalidades de Controlador.

```
@Controller
public class HelloWorldController {
    @RequestMapping("helloWorld")
    public String helloWorld(){
        return "exito";
    }
}
```

La firma de los métodos de la clase anotada es flexible, puede retornar

- String
- View
- ModelAndView
- Objeto (Anotado con @ResponseBody)

Si se desea que el retorno provoque una redireccion, basta con incluir el prefijo **redirect:**

```
@Controller
public class HelloWorldController {
    @RequestMapping("helloWorld")
    public String helloWorld(){
        return "redirect:/exito";
    }
}
```

Cuando se retorna el id de una View, se hace participe a esa View de la actual Request, cuando se redirecciona, se crea una Request nueva.

Y puede recibir como parámetro

- **Model**: Datos a emplear en la **View**.
- **Map<String, Object> model**: Lo resuelve como Model, permite el desacoplamiento con el API de Spring.
- Parametros anotados con **@PathVariable**: Dato que llega en el path de la Url.
- Parametros anotados con **@RequestParam**: Dato que llega en los parametros de la Url.
- Parametros anotados con **@CookieValue**: Dato que llega en un HTTP cookie
- Parametros anotados con **@RequestHeader**: Dato que llega en un HTTP Header
- Parametros anotados com **@SessionAttribute**: Atributo de la Sesion Http que se desea inyectar en el controlador
- **HttpServletRequest**
- **HttpServletResponse**
- **HttpSession**
- **Locale**
- **Principal**
- Parametros anotados com **@Validation**: Aplica las reglas de validacion sobre el parametro.
- **Errors**: Errores de validacion de aplicar las reglas de validacion sobre los parametros.
- **BindingResult**: Resultado de aplicar las validaciones sobre los parametros recibidos.

```
@PostMapping("/addValidatePhone")
public String submitForm(
    @Valid ValidatedPhone validatedPhone,
    BindingResult result, Model m) {

        if (result.hasErrors()) {
            return "phoneHome";
        }

        m.addAttribute("message", "Successfully saved phone: " + validatedPhone
.toString());
        return "phoneHome";
}
```

- **UriComponentsBuilder**

4.2. Activación de @Controller

Para activar esta anotación, habra que indicarle al contexto de Spring a partir de que paquete

puede buscarla. Se puede hacer con XML y con JavaConfig

Con XML, se emplea la etiqueta **ComponentScan**

```
<context:component-scan base-package="controllers"/>
```

Esta etiqueta activa el descubrimineto de las clases anotadas con @Component, @Repository, @Controller y @Service

Con JavaConfig, se emplea la anotacion **@ComponentScan**

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages={ "controllers" })
public class ContextoGlobal {

}
```

Esta anotacion activa el descubrimineto de las clases anotadas con @Component, @Repository, @Controller y @Service

4.3. @RequestMapping

Es la anotacion que permite resolver si la **HttpRequest** llega o no a un controlador, define un filtro de seleccion de Controladores basado en las caracteristicas del Request.

Se pueden definir

- **method**: Method HTTP
- **path**: Url
- **consumes**: Corresponde a la cabecera Content-Type
- **produces**: Corresponde a la cabecera Accept
- **params**: Permite indicar la obligatoriedad de la presencia de un parametro (params="myParam"), asi como de su ausencia (params="!myParam") o un valor determinado (params="myParam=myValue")
- **headers**: Igual que la anterior pero para cabeceras

4.4. @PathVariable

Anotacion que permite obtener información de la url que provoca la ejecucion del controlador.

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

Para el anterior ejemplo, dada la siguiente url **http://. . . /saludar/Victor**, el valor del parametro **nombre**, será **Victor**

Se pueden definir expresiones regulares para alimentar a los @PathVariable, siguiendo la firma **{varName:regex}**, por ejemplo

@RequestMapping("/spring-web/{symbolicName:[a-z-]}-{version:\\d\\.\\d\\.\\d}{extension:\\.[a-z]}") public void handle(@PathVariable String version, @PathVariable String extension) { // . . . }

4.5. @RequestParam

Anotacion que permite obtener información de los parametros de la url que provoca la ejecucion del controlador.

```
@RequestMapping(path="/saludar")
public ModelAndView saludar(@RequestParam("nombre") String nombre){
}
```

Para el anterior ejemplo, dada la siguiente url **http://. . . /saludar?nombre=Victor**, el valor del parametro **nombre**, será **Victor**

4.6. @SessionAttribute

Anotacion que permite recibir en un método de controlador, un atributo insertado con anterioridad en la Sesion con **@ModelAttribute** y **@SessionAttributes**.

```
@GetMapping("/nuevaFactura")
public String nuevaFactura(@SessionAttribute("login") Login login, @ModelAttribute
Factura factura) {
    return "factura/formulario";
}
```

4.7. @RequestBody

Permite tranformar el contenido del **body** de peticiones **POST** o **PUT** a un objeto java, tipicamente una representación en JSON.

```
@RequestMapping(path="/alta", method=RequestMethod.POST)
public String getDescription(@RequestBody UserStats stats){
    return "resultado";
}

public class UserStats{
    private String firstName;
    private String lastName;
}
```

En el ejemplo anterior, se convertirán a objeto, contenidos del **body** de la petición como por ejemplo

```
{ "firstName" : "Elmer", "lastName" : "Fudd" }
```

Para transformaciones a JSON, se emplea la siguiente libreria de **Jackson**

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.4.2</version>
</dependency>
```

4.8. @ResponseBody

Analogo al anterior, pero para generar un resultado.

Se aplica sobre métodos que retornan un objeto de información.

```
// controller
@ResponseBody
@RequestMapping("/description")
public Description getDescription(@RequestBody UserStats stats){
    return new Description(stats.getFirstName() + " " + stats.getLastname() + " hates
wacky wabbits");
}

public class UserStats{
    private String firstName;
    private String lastName;
    // + getters, setters
}

public class Description{
    private String description;
    // + getters, setters, constructor
}
```

Precisa dar de alta el API de marshall en el classpath.

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.4.2</version>
</dependency>
```

Es muy empleado en servicios REST.

4.9. @SessionAttributes

Tambien se puede asociar a la **session**, para ello se emplea la anotación **@SessionAttributes("nombreDelBeanDelModeloAAlmacenarEnLosAtributosDeLaSession")**, incluyendola como anotación de clase en la clase **Controller** que declare el bean del modelo con **@ModelAttribute**.

```
@Controller
@SessionAttributes("persona")
public class MyController {

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

Los objetos en Model, pueden ser inyectados directamente en los métodos del controlador con **@ModelAttribute**

```
@RequestMapping("/saludar")
public String saludar (@ModelAttribute("persona") Persona persona, Model model) {
    return "exito";
}
```

4.10. @ModelAttribute

Se pueden añadir objetos, con ambito **request**, al **Model** que manejara un controlador, con la anotación **@ModelAttribute**.

```
@Controller
public class MyController {

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

Si se mezcla con **@SessionAttributes** se consigue incluir el objeto en el ambito **session** en lugar de **request**

```
@Controller
@SessionAttributes({"persona"})
public class MyController {

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

Tambien permite que un controlador reciba objetos del modelo, independientemente de si estan en **request** o en **session**.

```
@PostMapping("/processForm")
public String processForm(@ModelAttribute(value = "persona") final Persona persona) {}
```

4.11. @MatrixVariable

Permite recibir pares de segmentos añadidos a una seccion del path, esto significa que se soporta

añadir a nivel del path, el contenido entre /, parametros realizando una division entre ellos con ; indicando en cada division una clave y un valor y posteriormente recoger de forma separada cada una de las divisiones.

Ojo, que no son parametros de la request dado que no aparece **?**

```
//Dada la siguiente peticion, donde a la ultima seccion del path se le ha
//añadido dos parametros q y r.
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
public void findPet(@PathVariable String petId, @MatrixVariable int q) {

    // petId == 42
    // q == 11
}
```

Como se puede realizar la parametrizacion de cualquier seccion del path, se podria llegar a tener varias secciones parametrizadas por lo que para recoger los parametros se tendrá que hacer referencia a la seccion

```
// GET /owners/42;q=11/pets/21;q=22

@GetMapping("/owners/{ownerId}/pets/{petId}")
public void findPet(
        @MatrixVariable(name="q", pathVar="ownerId") int q1,
        @MatrixVariable(name="q", pathVar="petId") int q2) {

    // q1 == 11
    // q2 == 22
}
```

4.12. @InitBinder

Permite redefinir:

- **CustomFormatter**: Permite defirnir transformaciones de tipos, se basa en la interface **Formatter**
- **Validators**: Validadores nuevos a aplicar a los Bean del Modelo, se basa en **Validator**
- **CustomEditor**: Parseos a aplicar a campos de los formularios, se basan en **PropertyEditor**

```
@InitBinder
public void customizeBinding(WebDataBinder binder) {

}
```

4.13. @ExceptionHandler

Permiten definir vistas a emplear cuando se producen excepciones en los métodos de control

```
@ExceptionHandler(CustomException.class)
public ModelAndView handleCustomException(CustomException ex) {

    ModelAndView model = new ModelAndView("error");
    model.addObject("ex", ex);
    return model;
}
```

4.14. @ControllerAdvice

Permiten definir en una clase independiente configuraciones de **@ExceptionHandler**, **@InitBinder** y **@ModelAttribute** que afectaran a los controladores que se desee, siempre que sean procesados por **RequestMappingHandlerMapping**, por ejemplo los **ViewControllers** no se ven afectados por esta funcionalidad.

```
@ControllerAdvice(basePackages="com.viewnext.holamundo.javaconfig.controllers")
public class GlobalConfig {
    @ModelAttribute
    public void initGlobal(Model model) {
        model.addAttribute("persona", new Persona());
    }
}
```

Capítulo 5. ViewResolver

El último componente a definir del flujo es el **ViewResolver**, este componente se encarga de resolver que **View** se ha emplear a partir del objeto **View** retornado por el **Controller**.

Pueden existir distintos **Bean** definidos de tipo **ViewResolver**, pudiendose ordenar con la propiedad **Order**.

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

Se proporcionan varias implementaciones, alguna de ellas

- **InternalResourceViewResolver**: Es el más habitual, permite interpretar el **String** devuelto por el **Controller**, como parte de la url de un recurso, componiendo la URL con un prefijo y un sufijo. Aunque es configurable, emplea por defecto las **View** de tipo **InternalResourceView**, de emplearse **JstlView**, se necesitaria añadir al classpath la dependencia con **jstl**
- **BeanNameViewResolver**: Busca un **Bean** declarado de tipo **View** cuyo **Id** sea igual al **String** retornado por el **Controller**.
- **ContentNegotiatingViewResolver**: Delega en otros **ViewResolver** dependiendo del **ContentType**.
- **FreeMarkerViewResolver**: Similar al **InternalResourceViewResolver**, pero el recurso buscado debe ser una plantilla Freemarker.
- **JasperReportsViewResolver**: Similar al **InternalResourceViewResolver**, pero el recurso buscado debe ser una plantilla JasperReport.
- **ResourceBundleViewResolver**: Busca la implementacion de la View en un fichero de properties.
- **TilesViewResolver**: Busca una plantillas de **Tiles** con nombre igual al **String** retornado por el **Controller**
- **VelocityViewResolver**: Similar al **InternalResourceViewResolver**, pero el recurso buscado debe ser una plantilla Velocity.
- **XmlViewResolver**: Similar a **BeanNameViewResolver**, salvo porque los **Bean** de las **View** han de ser declaradas en el fichero **/WEB-INF/views.xml**
- **XsltViewResolver**: Similar al **InternalResourceViewResolver**, pero el recurso buscado debe ser una plantilla XSLT.

5.1. InternalResourceViewResolver

Se ha de definir el Bean

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

5.2. XmlViewResolver

Se ha de definir el Bean

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

Y en el fichero **/WEB-INF/views.xml**

```
<bean id="pdf/listado" class="com.aplicacion.presentacion.vistas.ListadoPdfView"/>
<bean id="excel/listado" class="com.aplicacion.presentacion.vistas.ListadoExcelView"/>
<bean id="json/listado" class=
"org.springframework.web.servlet.view.json.MappingJacksonJsonView"/>
```

5.3. ResourceBundleViewResolver

Se ha de definir el Bean

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

Y en el fichero **views.properties** que estará en la raiz del classpath.

```
listado.(class)=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfVie
w
listado.url=/WEB-INF/jasperTemplates/reporteAfines.jasper
listado.reportDataKey=listadoKey
```

Donde **url** y **reportDataKey**, son propiedades del objeto **JasperReportsPdfView**, y **listado** el **String** que retorna el **Controller**

Capítulo 6. View

Son los componentes que renderizaran la resuesta a la petición procesada por Spring MVC.

Existen diversas implementaciones dependiendo de la tecnologia encargada de renderizar.

- AbstractExcelView
- AbstractAtomFeedView
- AbstractRssFeedView
- MappingJackson2JsonView
- MappingJackson2XmlView
- AbstractPdfView
- AbstractJasperReportView
- AbstractPdfStamperView
- AbstractTemplateView
- InternalResourceView
- JstlView: Es la que se emplea habitualmente para los JSP, exige la libreria JSTL.
- TilesView
- XsltView

6.1. AbstractExcelView

El API de Spring proporciona una clase abstracta que esta destinada a hacer de puente entre el API capaz de generar un Excel y Spring, pero no genera el Excel, para ello hay que incluir una libreria como **POI**

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

Algunas de las clases que proporciona **POI** son

- HSSFWorkbook
- HSSFSheet
- HSSFRow
- HSSFCell

```
public class PoiExcellView extends AbstractExcelView {
    @Override
    protected void buildExcelDocument(Map<String, Object> model, HSSFWorkbook
workbook, HttpServletRequest request, HttpServletResponse response) throws Exception {
        // model es el objeto Model que viene del Controller
        List<Book> listBooks = (List<Book>) model.get("listBooks");
        // Crear una nueva hoja excel
        HSSFSheet sheet = workbook.createSheet("Java Books");
        sheet.setDefaultColumnWidth(30);
        HSSFRow header = sheet.createRow(0);
        header.createCell(0).setCellValue("Book Title");
        header.createCell(1).setCellValue("Author");
        int rowCount = 1;
        for (Book aBook : listBooks) {
            HSSFRow aRow = sheet.createRow(rowCount++);
            aRow.createCell(0).setCellValue(aBook.getTitle());
            aRow.createCell(1).setCellValue(aBook.getAuthor());
        }
        response.setHeader("Content-disposition", "attachment; filename=books.xls");
    }
}
```

6.2. AbstractPdfView

De forma analoga al anterior, para los PDF, se tiene la libreria **Lowagie**

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

Algunas de las clases que proporciona **Lowagie** son

- Document
- PdfWriter
- Paragraph
- Table

```
public class ITextPdfView extends AbstractPdfView {
    @Override
    protected void buildPdfDocument(Map<String, Object> model, Document doc, PdfWriter
writer, HttpServletRequest request, HttpServletResponse response) throws Exception {
        // model es el objeto Model que viene del Controller
        List<Book> listBooks = (List<Book>) model.get("listBooks");
        doc.add(new Paragraph("Recommended books for Spring framework"));
        Table table = new Table(2);
        table.addCell("Book Title");
        table.addCell("Author");
            for (Book aBook : listBooks) {
            table.addCell(aBook.getTitle());
            table.addCell(aBook.getAuthor());
        }
        doc.add(table);
    }
}
```

6.3. JasperReportsPdfView

En este caso Spring proporciona una clase concreta, que es capaz de procesar las platillas de **JasperReports**, lo unico que necesita es la libreria de **JaspertReport**, la plantilla compilada **jasper** y un objeto **JRBeanCollectionDataSource** que contenga la información a representar en la plantilla.

La plantilla sin compilar será un fichero **jrxml**, que es un xml editable.

```
<dependency>
    <groupId>jasperreports</groupId>
    <artifactId>jasperreports</artifactId>
    <version>3.5.3</version>
</dependency>
```

A tener en cuenta que la version de la libreria de JasperReport debe coincidir con la del programa iReport empleando para generar la plantilla.

```
<bean id="reporteAfines" class=
"org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView">
    <property name="url" value="/WEB-INF/jasperTemplates/reporte.jasper"/>
    <property name="reportDataKey" value="listadoKey"></property>
</bean>
```

reportDataKey indica la clave dentro del objeto **Model** que referencia al objeto **JRBeanCollectionDataSource**

```
@Controller
public class AfinesReportController {
    @RequestMapping("/reporte")
    public String generarReporteAfines(Model model){
        JRBeanCollectionDataSource jrbean = new JRBeanCollectionDataSource(listado,
false);
        model.addAttribute("listadoKey", jrbean);
        return "reporteAfines";
    }
}
```

6.4. MappingJackson2JsonView

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.4.1</version>
</dependency>
```

Es para versiones de Spring posteriores a 4, para la 3 se emplea otro API y la clase **MappingJacksonJsonView**

Los Bean a convertir a JSON, han de tener propiedades.

Capítulo 7. Formularios

Para trabajar con formularios Spring proporciona una libreria de etiquetas

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

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.

Un ejemplo de definición de formulario podria ser

```
<form:form action="altaUsuario" modelAttribute="persona">
    <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>Sexo:</td>
            <form:select path="sexo" items="${listadoSexos}"/>
        </tr>
        <tr>
            <td colspan="2">
            <input type="submit" value="Guardar info" />
            </td>
        </tr>
    </table>
</form:form>
```

En el ejemplo anterior, se han definido a nivel del formulario.

- **action**: Indica la Url del Controlador.
- **modelAttribute**: Indica la clave con la que se envía el objeto que se representa en el formulario. (de forma analoga se puede emplear **commandName**)

No es necesario definir el action si se emplea la misma url para cargar el formulario y para recibirlo, basta con cambiar unicamente el METHOD HTTP No hay diferencia entre **commandName** y **modelAtribute**

Para recuperar en el controlador el objeto enviado, se emplea la anotación **@ModelAttribute**

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 representado en el formulario y presentará la pagina del formulario.
- El POST será invocado desde el formulario recibiendo los datos a traves del objeto previamente inicializado.

```
@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";
}
```

El tener que definir el endpoint **GET** puede parecer engorroso para lo que hace, dado que se podria refactorizar minimizando su impacto haciendo

Con lo que realmente tenemos un **ViewController** definido por nosotros, lo cual nos hace pensar en la posibilidad de emplear el tipo de componente que da Spring para esos menesteres y emplear si cabe algo como **@ControllerAdvice** para añadir el objeto del formulario al modelo, pero los **ViewController** no se ven afectados por los **ControllerAdvice**, por lo que la solución mas optima será la definición de un **Interceptor**

```
public class CommonModelInterceptor extends HandlerInterceptorAdapter {
    @Override
    public void postHandle(HttpServletRequest request,
                           HttpServletResponse response,
                           Object handler,
                           ModelAndView model) throws Exception {
        Persona p = new Persona(null, "", "", null, "Hombre", null);
        model.addAttribute("persona", p);
        model.addAttribute("listadoSexos", new String[]{"Hombre","
Mujer"});
    }
}
```

7.1. Ficheros

Si se desean recibir ficheros desde el cliente, se empleará la tipologia **MultipartFile**

```
@RequestMapping(value = "/uploadFile", method = RequestMethod.POST)
public String submit(@RequestParam("file") final MultipartFile file, final ModelMap
modelMap) {

    modelMap.addAttribute("file", file);
    return "fileUploadView";
}
```

Hay que tener en cuenta que este tipo esta asociado a envios de formulario con **enctype="multipart/form-data"**

```
<form:form method="POST" action="/uploadFile" enctype="multipart/form-data">
    <input type="file" name="file" />
    <input type="submit" value="Submit" />
</form:form>
```

7.2. Etiquetas

Spring proporciona dos librerias de etiquetas

- Formularios
- **<form:form></form:form>**: Crea una etiqueta HTML form.
- **<form:errors></form:errors>**: Permite la visualización de los errores asociados a los campos del **ModelAttribute**
- **<form:checkboxes items="" path=""/>**:
- **<form:checkbox path=""/>**:
- **<form:hidden path=""/>**:
- **<form:input path=""/>**:
- **<form:label path=""/>**:
- **<form:textarea path=""/>**:
- **<form:password path=""/>**:
- **<form:radiobutton path=""/>**:
- **<form:radiobuttons path=""/>**:
- **<form:select path=""/>**:
- **<form:option value=""/>**:
- **<form:options/>**:
- **<form:button/>**:
- Core
- **<spring:argument/>**:
- **<spring:bind path=""/>**:
- **<spring:escapeBody/>**:
- **<spring:eval expression=""/>**:
- **<spring:hasBindErrors name=""/>**:
- **<spring:htmlEscape defaultHtmlEscape=""/>**:
- **<spring:message/>**:
- **<spring:nestedPath path=""/>**:
- **<spring:param name=""/>**:
- **<spring:theme/>**:
- **<spring:transform value=""/>**:
- **<spring:url value=""/>**:

7.3. Paths Absolutos

En ocasiones, se requiere acceder a un controlador desde distintas JSP, las cuales estan a distinto nivel en el path, por ejemplo desde **/gestion/persona** y desde **/administracion**, se quiere acceder a **/buscar**, teniendo en cuenta que la propiedad **action** representa un path relativo, no serviria en mismo formulario, salvo que se pongan path absolutos, para los cual, se necesita obtener la url de la aplicación, hay varias alternativas

- Expresiones EL

```
<form action="${pageContext.request.contextPath}/buscar" method="GET" />
```

- Libreria de etquetas JSTL core

```
<form action="<c:url value="/buscar" />" method="GET" />
```

7.4. Inicialización

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, dado que por defecto un objeto definido como **ModelAttribute** se situa en **HttpServletRequest** que es donde se ira a buscar al renderizar la JSP del formulario.

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

Capítulo 8. 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>5.1.3.Final</version>
</dependency>
```

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

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

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 **Errors**, que tiene un método **hasErrors()** que indica si hay errores de validación.

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

    if (errors.hasErrors()) {
        return "error";
    } else {
        return "ok";
    }
}
```

Y en la clase del **Model**, las anotaciones correspondientes de JSR-303

```
public class Persona {
    @NotEmpty(message="Hay que rellenar el campo nombre")
    private String nombre;
    @NotEmpty
    private String apellido;
    private int edad;
}
```

8.1. Mensajes personalizados

Como se ve en el anterior ejemplo, se ha personalizado el mensaje para la validación **@NotEmpty** del campo **nombre**

Se puede definir el mensaje en un properties, teniendo en cuenta que el property tendra la siguiente firma

```
<validador>.<entidad>.<caracteristica>
```

Por ejemplo para la validación anterior de **nombre**

```
notempty.persona.nombre = Hay que rellenar el campo nombre
```

Tambien se puede referenciar a una propiedad cualquiera, pudiendo ser cualquier clave.

```
@NotEmpty(message="{notempty.persona.nombre}")
private String nombre;
```

8.2. Anotaciones JSR-303

Las anotaciones están definidas en el paquete **javax.validation.constraints**.

- **@Max**
- **@Min**
- **@NotNull**
- **@Null**
- **@Future**
- **@Past**
- **@Size**
- **@Pattern**

8.3. Validaciones Custom

Se pueden definir validadores nuevos e incluirlos en la validación automatizada, para ello hay que implementar la interface **org.springframework.validation.Validator**

```
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");
    }
}
```

El metodo de supports, indica que clases se soportan para esta validación, si retornase true, aceptaria todas, no es lo habitual ya que tendrá al menos una caracteristica concreta que será la validada.

Una vez definido el validador, para añadirlo al flujo de validación de un **Controller**, se ha de añadir una instancia de ese validador al **Binder** del **Controller**, creando un método en el **Controller**, anotado con **@InitBinder**

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

Los errores asociados a estas validaciones pueden ser visualizados en la **View** empleando la etiqueta <**form:errors**/>

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

La propiedad path, es el camino que hay que seguir en el objeto de **Model** para acceder a la propiedad validada.