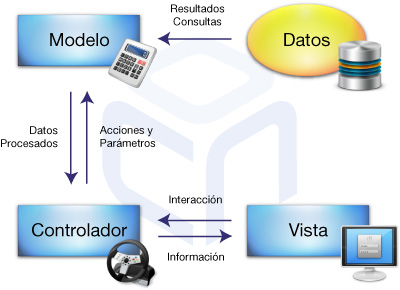
**Modelo Vista Controlador**

# El Modelo Vista Controlador (MVC) es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del. Este patrón de diseño se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento



[http://bits.wikimedia.org/static-1.23wmf18/skins/common/images/magnify-clip.png](http://es.wikipedia.org/wiki/Archivo:ModelViewControllerDiagram_es.svg)

Un diagrama sencillo que muestra la relación entre el modelo, la vista y el controlador. Nota: las líneas sólidas indican una asociación directa, y las punteadas una indirecta (por ejemplo, [patrón Observer](http://es.wikipedia.org/wiki/Observer_(patr%C3%B3n_de_dise%C3%B1o))).

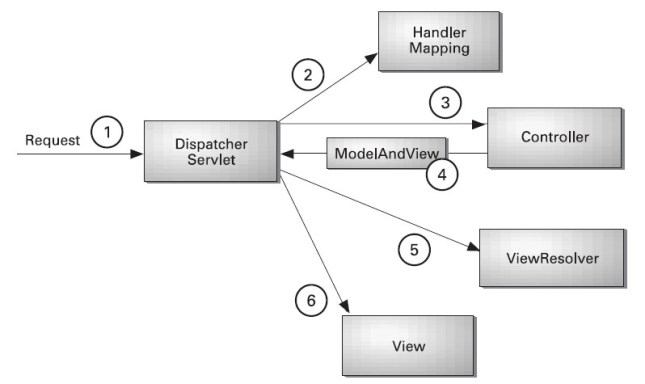
De manera genérica, los componentes de MVC se podrían definir como sigue:

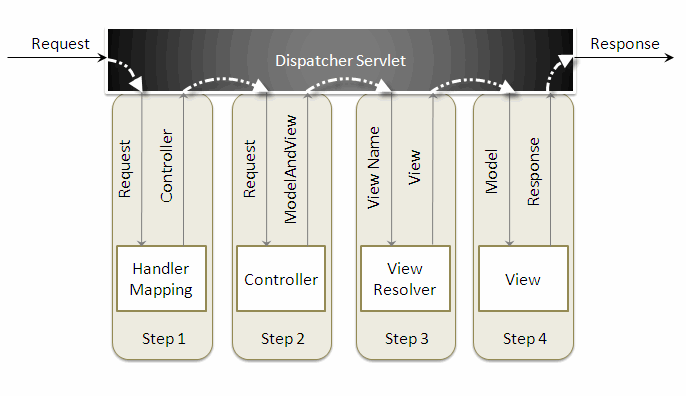
* El **Modelo**: Es la representación de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio). Las peticiones de acceso o manipulación de información llegan al 'modelo' a través del 'controlador'.
* El **Controlador**: Responde a eventos (usualmente acciones del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento o un registro en una base de datos). También puede enviar comandos a su 'vista' asociada si se solicita un cambio en la forma en que se presenta de 'modelo' (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos), por tanto se podría decir que el 'controlador' hace de intermediario entre la 'vista' y el 'modelo'.
* La **Vista**: Presenta el 'modelo' (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la [interfaz de usuario](http://es.wikipedia.org/wiki/Interfaz_de_usuario)) por tanto requiere de dicho 'modelo' la información que debe representar como salida.

**Spring Web y MVC**

Spring proporciona herramientas para implementar la capa web de nuestras aplicaciones (principalmente Spring MVC y Spring Webflow), aunque también es posible integrarlo con otros frameworks como Struts, JSF, Wicket… En este post veremos cómo inicializar Spring en una aplicación web y cómo funciona Spring MVC.

En la siguiente imagen vemos el esquema de cómo funciona [Spring MVC](http://static.springsource.org/spring/docs/2.0.x/reference/mvc.html).

[](http://hop2croft.files.wordpress.com/2011/09/springmvclifecycle.jpg)



**DispatcherServlet**. Este servlet se va a encargar de gestionar toda la lógica en nuestra aplicación.

**El flujo básico en una aplicación bajo Spring MVC es el siguiente:**

* La petición llega a nuestro DispatcherServlet **(1)**
* El DispatcherServlet  tendrá que encontrar que controlador va a tratar la petición. Para ello elDispatcherServlet tiene que encontrar el manejador asociado a la url de la petición. Todo esto se realiza en la fase de **HandlerMapping** **(2)**.
  + El Handler Mapping como indicamos tiene el objetivo de indicar al dispatcher  
    cual será el componente que debe recibir el request enviado por el usuario.  
    Por lo cual el Dispatcher Servlet, le pregunta a uno o más Handler Mapping  
    cual será le Controller que recibirá el Request.   
    Dentro del Spring existe varios Handler Mapping los cuales tiene distintas  
    capacidades de poder mapear a los controladores.

En el siguiente cuadro se indican los Handler Mapping que posee Spring.

|  |  |
| --- | --- |
| Handler Mapping | Como mapea el Request |
| BeanNameUrlHandlerMapping | Mapea controladores a URL basandose en el nombre del Bean |
| SimpleUrlHandlerMapping | Mapea controladores a URL basandose en una colección de propiedades que se definen en el Spring application context. |
| ControllerClassNameHandlerMapping | Mapea controladores a URL utilizando el Controller Class Name |
| CommonsPathMapHandlerMapping | Mapea controladores a URL usando metadatas en el codigo del controlador. Esta metadata es definida usando Jakarta Commons Atributes. |

* Una vez encontrado ese **Controller**, nuestro DispatcherServlet  le dejará gestionar a éste la petición **(3)**. En nuestro controlador se deberá realizar todo la lógica de negocio de nuestra aplicación, es decir, aquí llamaremos a nuestra capa de servicios.

Para poder implementar un Controller sobre Spring es necesario que se cree una clase que herede de los Controller que han sido implementados por Spring, los cuales dependiendo de la funcionalidad a realizar es el Controller que se deberá utilizar. En el siguiente cuadro se puede ver qué tipos de Controller están disponibles sobre Spring.

|  |  |  |
| --- | --- | --- |
| Controller type | Classes | Usarlo  cuando ... |
| View | ParameterizableViewController UrlFilenameViewController | Para cuando un controlador solo necesita desplegar informacion |
| Simple | Controller (interface) AbstractController | Para controladores simples que solo se utilizan como Simples Servlet |
| Throwaway | ThrowawayController | Para manejar los request como un Commando |
| Multiaction | MultiActionController | Para implementar una serie de acciones con similar logica |
| Command | BaseCommandController AbstractCommandController | Si tus controladores reciben parametros estos son manejados dentro de un objeto |
| Form | AbstractFormController SimpleFormController | Para desplegar y procesar un formulario, bajo el mismo componente |
| Wizard | AbstractWizardFormController | Para realizar una interacción más rica con el usuario a través de varias pantallas |

* El controlador devolverá al Dispatcher un objeto de tipo **ModelAndView** **(4)**. ¿Qué quiere decir ésto? En pocas palabras Model serán los valores que obtengamos de nuestra capa de servicio y View será el nombre de la vista en la que queremos mostrar la información que va contenida dentro de ese Model.

El ModelAndView tiene como función la de:

1- Entregar un nombre lógica a la vista que deberá realizar el despliegue del Model  
2- Entregar un nombre lógico al Model que está asociado a este componente  
3- Inyectar el objeto Model el cual tiene los datos que serán desplegados en la Vista.

* Una vez pasado este objeto ModelAndView al DispatcherServlet, será éste el que tendrá que asociar el nombre de la vista retornada por el controlador a una vista concreta (página jsp, jsf,…). Este proceso viene indicado en la imagen como **ViewResolver** **(4)**.

El ViewResolver es el encargado de realizar el mapping entre el nombre lógico de la vista y el componente. En el siguiente cuadro se indican los ViewResolver disponibles a ser configurados sobre Spring.

|  |  |
| --- | --- |
| **View Resolver** | **Como trabaja ....** |
| InternalResourceViewResolver | Resuelve el nombre logico utilizando el mapping a velocity y JSP |
| BeanNameViewResolver | Resuleve el nombre logico utilizando Bean definidos en el Spring Context |
| ResourceBundleViewResolver | Define el mapping entre los nombres logicos y las vistas asociadas , definiendolo en un archivo de propiedades |
| XmlViewResolver | Define el mapping entre los nombres logicos y las vistas asociadas , definiendolo en un archivo XML |

* Finalmente y una vez resuelta la vista, nuestro DispatcherServlet  tendrá que pasar nuestro Model (los valores recogidos en el controlador a través de nuestra capa de servicios) a la vista concreta **View (5)**.

# Controllers mediante anotaciones

Las anotaciones sirven para simplificarnos enormemente el trabajo a la hora de crear nuestros Controllers, al mismo tiempo que resulta mucho más elegante y queda mucho mejor estructurado nuestro proyecto.

Primero veamos un sencillísimo ejemplo de controlador mediante anotaciones.

|  |
| --- |
| @Controller  public class Controlador {  @RequestMapping("/page")   public ModelAndView pageHandler(HttpServletRequest request) {    return new ModelAndView("page");   }     @RequestMapping("/carganombre")   public ModelAndView cargadorNombreHandler(HttpServletRequest request) {     String nombre = request.getParameter("nombre");   String apellido = request.getParameter("apellido");     ModelAndView mav = new ModelAndView();   mav.addObject("nombre", nombre);   mav.addObject("apellido", apellido);   mav.setViewName("page");     return mav;   }  } |

La etiqueta **@Controller** la emplea Spring para escanear las clases (mediante unos beans específicos que definiremos más adelante) y así detectar cuales de estas clases forman los Controllers de nuestra aplicación.

La etiqueta **@RequestMapping** la emplea Spring para conocer a que Controller o método de un Controller tiene que direccionar cada llamada del cliente. En el ejemplo, le estamos informando de que todas las llamadas a “page.htm” van a ser manejadas por el método *pageHandler* de nuestro Controller, y que las llamadas a ”carganombre.htm” serán manejadas por el método *cargadorNombreHandler*.

Pero las posibilidades no acaban aquí, también podemos distinguir diferentes tipos de llamadas a una misma dirección, y tratarlas según nos convenga, por ejemplo.

|  |
| --- |
| @Controller  @RequestMapping("/page")  public class Controlador {    @RequestMapping(method = RequestMethod.GET)  public ModelAndView pageHandler(...) {    ...   }    @RequestMapping(method = RequestMethod.POST)  public ModelAndView cargadorNombreHandler(...) {    ...   }    } |

De este modo, si la llamada a la dirección se realiza por GET o POST la trataremos de diferente forma. Para mas información sobre los argumentos y tipos soportados por @RequestMapping, podéis acudir al siguiente enlace y mirar en la sección [*13.11.4. Supported handler method arguments and return types*](http://static.springsource.org/spring/docs/2.5.x/reference/mvc.html#mvc-annotation).

Otra etiqueta, que puede resultarnos interesante si no se pasan muchos parámetros en la llamada, es la etiqueta @RequestParam, con la que nos podemos ahorrar unas lineas de programación, si antes hemos utilizado el objeto HttpServletRequest para recoger manualmente los parámetros de la llamada, podríamos utilizar la etiqueta @RequestParam para mecanizar este paso.

|  |
| --- |
| @RequestMapping("/carganombre")  public ModelAndView cargadorNombreHandler(@RequestParam("nombre") String nombre, @RequestParam("apellido") String apellido) {        ModelAndView mav = new ModelAndView();      mav.addObject("nombre", nombre);      mav.addObject("apellido", apellido);      mav.setViewName("page");        return mav;  } |

Spring Annotations nos proporciona mas etiquetas mas configurar nuestros Controllers, algunas como**@Autowired** y **@Qualifier** para la inyección de dependencias, etc…

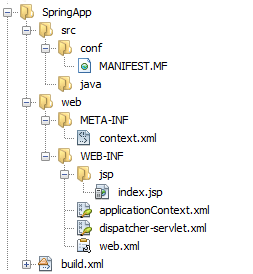
El siguiente y último paso que tendremos que hacer, es configurar debidamente nuestros beans en Spring para que detecte nuestros Controllers mediante anotaciones.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="<http://www.springframework.org/schema/beans>"  xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"  xmlns:p="<http://www.springframework.org/schema/p>"  xmlns:aop="<http://www.springframework.org/schema/aop>"  xmlns:tx="<http://www.springframework.org/schema/tx>"  xmlns:context="<http://www.springframework.org/schema/context>"  xsi:schemaLocation="<http://www.springframework.org/schema/beans> <http://www.springframework.org/schema/beans/spring-beans-3.0.xsd> <http://www.springframework.org/schema/context> <http://www.springframework.org/schema/context/spring-context-3.0.xsd>">    <!-- Anotaciones -->  <bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"/>  <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter"/>    <!-- Detector de controladores mediante anotaciones -->  <context:component-scan base-package="com.programacionj2ee.controllers"/>    ...  </beans> |

Lo primero que debemos hacer es declarar el componente xmlns:context y xsi:schemaLocation debidamente, después  configuramos los beans [DefaultAnnotationHandlerMapping](http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/web/servlet/mvc/annotation/DefaultAnnotationHandlerMapping.html) y [AnnotationMethodHandlerAdapter](http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/web/servlet/mvc/annotation/AnnotationMethodHandlerAdapter.html), y por último el component-scanque se encargara de buscar en el paquete indicado todas las clases con la etiqueta @Controller para registrarlas como Controllers.

### ****Estructura de un proyecto con Spring Framework****

Un proyecto Spring se trata de un proyecto Java Enterprise Edition con una serie de archivos propios para la configuración del proyecto. Aquí podemos ver una estructura básica de un proyecto Spring.

[caption id="attachment\_410" align="aligncenter" width="253"][](http://www.programacionj2ee.com/wp-content/uploads/2013/08/spring-structure.png)Estructura de proyecto Spring[/caption]

Una vez compilado se genera un archivo con formato .war (Web Application Archive), que no es mas que un archivo .zip comprimido renombrado. Este archivo será el que finalmente subiremos al servidor para ser desplegado y que arranque la aplicación.

**Directorios destacables**

* **src -** Se encuentran las clases Java de la aplicación.
* **web -** Directorio raíz de la aplicación web, donde se encuentran todos los recursos de la aplicación.
* **web/WEB-INF -** Este directorio está protegido por el contenedor de la aplicación y no es accesible desde un cliente externo, para poder acceder a los recursos almacenados en esta ubicación deberemos hacerlo desde las clases Java de la propia aplicación.
* **web/lib -** Aunque no se encuentra en esta estructura, este directorio se utiliza para almacenar las librerías y dependencias de la aplicación.

**Archivos destacables**

* **web/WEB-INF/web.xml** - Despliega los recursos básicos de la aplicación, en una aplicación con Spring declararemos un [DispatcherServlet](http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html" \o "DispatcherServlet" \t "_blank), un [ContextLoaderListener](http://static.springsource.org/spring/docs/3.1.x/javadoc-api/org/springframework/web/context/ContextLoaderListener.html" \o "ContextLoaderListener" \t "_blank) y estableceremos otro archivo .xml (generalmente llamado applicationContext.xml) donde configuraremos los recursos que serán creados en el despliegue de la aplicación.
* **web/WEB-INF/dispatcher-servlet.xml** - Configura un nuevo contexto de aplicación con las dependencias del [DispatcherServlet](http://static.springsource.org/spring/docs/3.2.x/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html" \o "DispatcherServlet" \t "_blank) configurado desde web.xml, su nombre viene determinado por el atributo <servlet-name />especificado en web.xml más la cadena “-servlet.xml”.
* **web/WEB-INF/applicationContext.xml –**Configura el contexto de la aplicación, inicializando en el despliegue de la aplicación los recursos generales básicos de la misma, DataSources, MessageSources, Marshallers y otros recursos de la aplicación.

# ¿Qué ventajas tiene Spring MVC respecto a frameworks antecedentes como Struts?

Struts es un framework mucho más antiguo, por lo que Spring ha aprendido de la experiencia adquirida de usar Struts para no cometer los mismos errores.

A continuación se enumeran un conjunto de ventajas:

1. Spring MVC ofrece una división limpia entre Controllers, Models (JavaBeans) y Views.
2. Spring MVC es muy flexible, ya que implementa toda su estructura mediante interfaces, no como Struts que obliga a heredar de clases concretas tanto en sus Actions como en sus Forms.
3. Spring MVC provee interceptores y controllers que permiten interpretar y adaptar el comportamiento común en el manejo de múltiples requests.
4. Los controllers de Spring MVC se configuran mediante IoC como los demás objetos, lo cual los hace fácilmente testeables e integrables con otros objetos que estén en el contexto de Spring, y por tanto sean manejables por éste.
5. Las partes de Spring MVC son más fácilmente testeables que las de Struts, debido a que evita la herencia de una clase de manera forzosa y una dependencia directa en el controller del servlet que despacha las peticiones.
6. No existen ActionForms, se enlaza directamente con los beans de negocio.
7. Struts obliga a extender la clase Action, mientras que Spring MVC no, aunque proporciona una serie de implementaciones de Controllers para que el usuario los utilice. Existe una gran variedad de Controladores.
8. Spring tiene una interfaz bien definida para la capa de negocio.
9. Spring ofrece mejor integración con tecnologías distintas a JSP, como Velocity,XSLT,FreeMaker y XL.

**¿Qué ventajas tiene Spring MVC en la versión 3 respecto a la 2?**

1. Se ha introduce un nombre es espacio MVC que simplifica la configuración.
2. Se han insertado anotaciones adicionales como @CookieValue (permite relacionar un atributo de un método a una cookie) y @RequestMappings (asociar directamente en la clase la posibilidad de asociar una petición Url a un controlador específico).
3. El tipo ConversionService es una alternativa más simple y robusta que los PropertyEditors de JavaBeans.
4. Se soporta un formateo de números con el atributo @NumberFormat.
5. Se soportan el formateo de fechas, calendarios y joda time con el atributo @DatetimeFormat, siempre que la librería Joda Time esté dentro del classpath.
6. Se soporta la validación para entradas @Controller con la etiqueta @Valid, si se proporciona una implementación de la Jsr-303 en el classpath.
7. Soporte para leer y escribir Xml, si Jaxb está dentro del classpath.
8. Soporte para leer y escribir Json, si Jackson está dentor del classpath.

**¿Qué pasos hay que hacer para crear los controladores?**

1. Definir una clase que implementa la interfaz del controlador.
2. Insertar dicha clase como un objeto en el contexto de Spring.
3. Asignar el nombre a una Uri que posteriormente será invocada por el usuario.
4. Especificar la extensión de la uti en el DispatcherServlet de Spring, configurado en web.xml, para que se pueda buscar internamente.

**Empezando a trabajar con MVC**

Lo primero que debemos tener en cuenta cuando utilizamos Spring en una aplicación web es cómo el contexto de aplicación se integra en el ciclo de vida de ésta. Para conseguir que nuestro contexto se inicie antes de los servlets de la aplicación y se cierre cuando paremos, debemos añadir un servlet listener a nuestro web.xml:

|  |  |
| --- | --- |
|  | <context-param>    <param-name>contextConfigLocation</param-name>    <!-- Fichero que contiene el contexto global de aplicación -->    <param-value>/WEB-INF/hospital-webapp-config.xml</param-value>  </context-param>  <listener>    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener> |

Este recurso se encarga básicamente de dos aspectos fundamentales.

1. Realiza la configuración automáticamente en tiempo de despliegue del ApplicationContext (o ServletContext) de la aplicación.
2. Crea un [WebApplicationContext](http://static.springsource.org/spring/docs/3.1.x/javadoc-api/org/springframework/web/context/WebApplicationContext.html" \o "WebApplicationContext" \t "_blank) (Interface to provide configuration for a web application) que nos facilita el acceso al [ApplicationContext](http://static.springsource.org/spring/docs/3.1.x/javadoc-api/org/springframework/context/ApplicationContext.html" \o "ApplicationContext" \t "_blank) a través de [ServletContextAware](http://static.springsource.org/spring/docs/3.1.x/javadoc-api/org/springframework/web/context/ServletContextAware.html) y el método getServletContext().

**IMPORTANTE**

El ServletContext es el lugar donde habitan los beans (recursos instanciados y configurados) de Spring, solo existe UN ServletContext por aplicación. Por otra parte, cada Servlet tiene su propio ServletConfig.

~~El listener se encargará de~~ **~~cargar el contexto de aplicación~~** ~~antes de que se inicialicen los servlets, que, a partir de ese momento, lo tendrán disponible. Esto es lo que conocemos como “root context”, puesto que es accesible por todos los servlets.~~

El siguiente paso es entender cómo nuestra aplicación va a atender las peticiones web con Spring MVC. Este proceso siempre sigue el mismo patrón:

1. Se recibe una petición a una URL
2. Se llama al método que atiende
3. El método se ejecuta y, opcionalmente, devuelve un valor
4. Se renderiza la respuesta

Para atender una petición Spring MVC utiliza la clase [DispatcherServlet](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/web/servlet/DispatcherServlet.html" \t "_blank). Esta actúa como un controlador frontal y se encarga de coordinar todas las peticiones delegando su resolución en los beans de infraestructura web de la aplicación.

Cada módulo web de la aplicación tendrá su propio DispatcherServlet con un contexto de aplicación local y con acceso al “root context”. Idealmente, situaremos todos aquellos componentes que estén más relacionados con la capa web (controladores, vistas…) en estos contextos locales, mientras que los beans que pertenezcan a la capa de aplicación (servicios, repositorios…) serán compartidos a través del “root context”. Veamos un ejemplo de cómo configurar un DispatcherServlet para el módulo de administración de nuestra aplicación:

|  |  |
| --- | --- |
| WEB-INF/web.xml <?xml version="1.0" encoding="UTF-8"?>  <web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">  <!-- Processes application requests -->  <servlet>  <servlet-name>appServlet</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>appServlet</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>  </web-app>  Descripción del web.xml   * Registramos el DispatcherServlet como un Servlet llamado appServlet. * Mapeamos este Servlet para manejar las solicitudes entrantes que comienzan con "/". * Usamos el parámetro init ContextConfigLocation para personalizar la ubicación del archivo XML de configuración base para el SpringApplicationContext que es cargado por el DispatcherServlet, en lugar de depender de la ubicación predeterminada. |  |

Cuando un DispatcherServlet recibe una petición llamará al interfaz [HandlerMapping](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/web/servlet/HandlerMapping.html) (Interface to be implemented by objects that define a mapping between requests and handler objects.). Este es el encargado de decidir qué Handler debe resolver la petición e indicárselo al DispatcherServlet. Una vez qué sepa el Handler al que debe delegar la llamada, lo invocará a través del interfaz [HandlerAdapter](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/web/servlet/HandlerAdapter.html).

¿Qué es un Handler? Es a lo que normalmente llamamos controlador en las aplicaciones MVC (Modelo Vista Controlador). Para indicarle a Spring MVC que una clase es un controlador contamos con la anotación @Controller mientras que para darle la capacidad de atender determinadas peticiones utilizaremos la anotación @RequestMapping:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Controller  public class PatientController {      @RequestMapping("/patients.htm") // Se pueden utilizar comodines: \*    public String listPatients(Model model) {       ...    }  } |

Imaginemos que la aplicación recibe la siguiente petición:<http://myserver:8080/hospital/admin/patients.htm>

Esta URL indicaría lo siguiente:

* Protocolo: HTTP
* Servidor: myserver
* Puerto: 8080
* Webapp: hospital
* Servlet: admin
* Request mapping: patients.htm

De esta forma, el DispatcherServlet sabe que ante esta petición, debe llamar al método *listPatients* de la clase *PatientController*. Además, el objeto model contendrá los parámetros de la llamada. Otra forma de obtener estos parámetros es mediante la anotación @RequestParam que también realizará la conversión de tipo:

<http://myserver:8080/hospital/admin/showPatient.htm?patient_id=1>

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Controller  public class PatientController {      @RequestMapping("/showPatient")    public String showPatient(@RequestParam("patient\_id") Integer id, Model model) {       ...       return "patient";    }  } |

También es posible extraer valores de la URI mediante URI Templates y la anotación @PathVariable:

<http://myserver:8080/hospital/admin/patients/1>

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @Controller  public class PatientController {      private PatientService patientService;      @Autowired    public PatientController(PatientService patientService) {      this.patientService=patientService    }      @RequestMapping("/patients/{patientId}")    public String showPatient(@PathVariable("patientId") Integer id, Model model) {       Patient patient = patientService.findById(id);       model.addAttribute("patient",patient);       return "showPatient";    }  } |

Una vez que un controlador atiende una petición y almacena el resultado en el objeto model, ¿qué devuelve? Normalmente será el nombre de una vista (por defecto se trata como la ruta a una JSP), pero también podría no devolver nada (en cuyo caso se seleccionará la vista por defecto). También sería posible devolver un objeto de tipo View, pero no es lo habitual.

En caso de que el controlador devuelva el nombre de una vista al DispatcherServlet, este delegará su resolución al [ViewResolver](http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/web/servlet/ViewResolver.html" \t "_blank) que identificará la vista que corresponde al nombre indicado para que sea renderizada y devuelta al cliente que realizó la petición. Veamos un ejemplo de configuración de ViewResolver:

|  |  |
| --- | --- |
| 1  2  3  4 | <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">    <property name="prefix" value="/WEB-INF/views/">    <property name="suffix" value=".jsp">  </bean> |

Este ViewResolver concatena “/WEB-INF/views/”+nombreDeVista+”.jsp” para identificar la vista que debe renderizarse. En el ejemplo anterior: “/WEB-INF/views/showPatient.jsp”

El siguiente paso sería implementar la vista showPatient.jsp (${patient.X} hace referencia al objeto Patient que insertarmos en el modelo desde el controlador):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <html>    <head><title>Patient</title></head>    <body>      Patient: ${patient.name}, ${patient.lastName}      Room: ${patient.roomId}    </body>  </html> |

Por último, debemos registrar nuestro controlador en el contexto local a este servlet. Para ello, [debemos definirlo como si se tratase de cualquier otro bean en el fichero /WEB-INF/admin-config.xml o habilitar el escaneo de componentes](http://csilgoortiz.com/2012/04/24/comprendiendo-spring-5-configuracion-basada-en-anotaciones-y-basada-en-java/).

En esta imagen se puede ver los pasos que sigue una llamada para ser resuelta por Spring MVC:  
