Formación
BDI

Curso Spring Framework

Módulo 3
Spring Web MVC

Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com

Temas Spring Web MVC

Introducción Spring MVC

- Que es Spring MVC?
- Características Spring MVC
- DispatchServlet y Configuración Context Application

Controller

- Anotación @Controller
- Anotación @RequestMapping
- Plantillas de URL con parámetros (URI template)
- Argumentos en Métodos Handler
- Tipos de retorno en Métodos Handler

¿Que es Spring MVC?

¿Que es Spring MVC?

Un Framework de aplicaciones web basado en MVC que toma ventajas de los siguientes principios de diseño

- Inyección de Dependencia
- Orientado al uso de Interfaces
- Extenso uso de clases POJO
- Desarrollo testeable

Características Spring MVC

Características de spring MVC

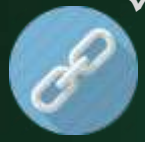
- Clara separación de funciones
  - ✓ Controller, validator, command object (form object), DispatcherServlet, handler mapping, view resolver, etc.
  - ✓ Llevan a cabo una tarea específica y pueden ser reemplazables sin afectar a los demás
- Configuración Robusta pero sencilla de realizar
  - ✓ Tanto para las clases del framework como las clases propias de la aplicación como JavaBeans

Características de spring MVC

- Adaptabilidad, no intrusivo, y flexibilidad
  - ✓ Diferentes formas de definir los métodos en los controladores, diversas firmas de métodos que necesitemos implementar para un escenario determinado, por ejemplo usando anotaciones en los argumentos (tales como @RequestParam, @RequestHeader, @PathVariable, etc)
- ✓ Conversiones Configurables (binding) y validaciones

Características de spring MVC

- Mapeo URL Configurable de los controladores (handler mapping) y resolución de vista (view resolution)
  - ✓ Diferentes estrategias para la resolución de vista, por ejemplo configuración basada en URL.



- Transferencia del objeto model, desde el controlador a la vista y viceversa
  - ✓ Usando Map (llave/valor) fácil integración con cualquier tecnología de vista

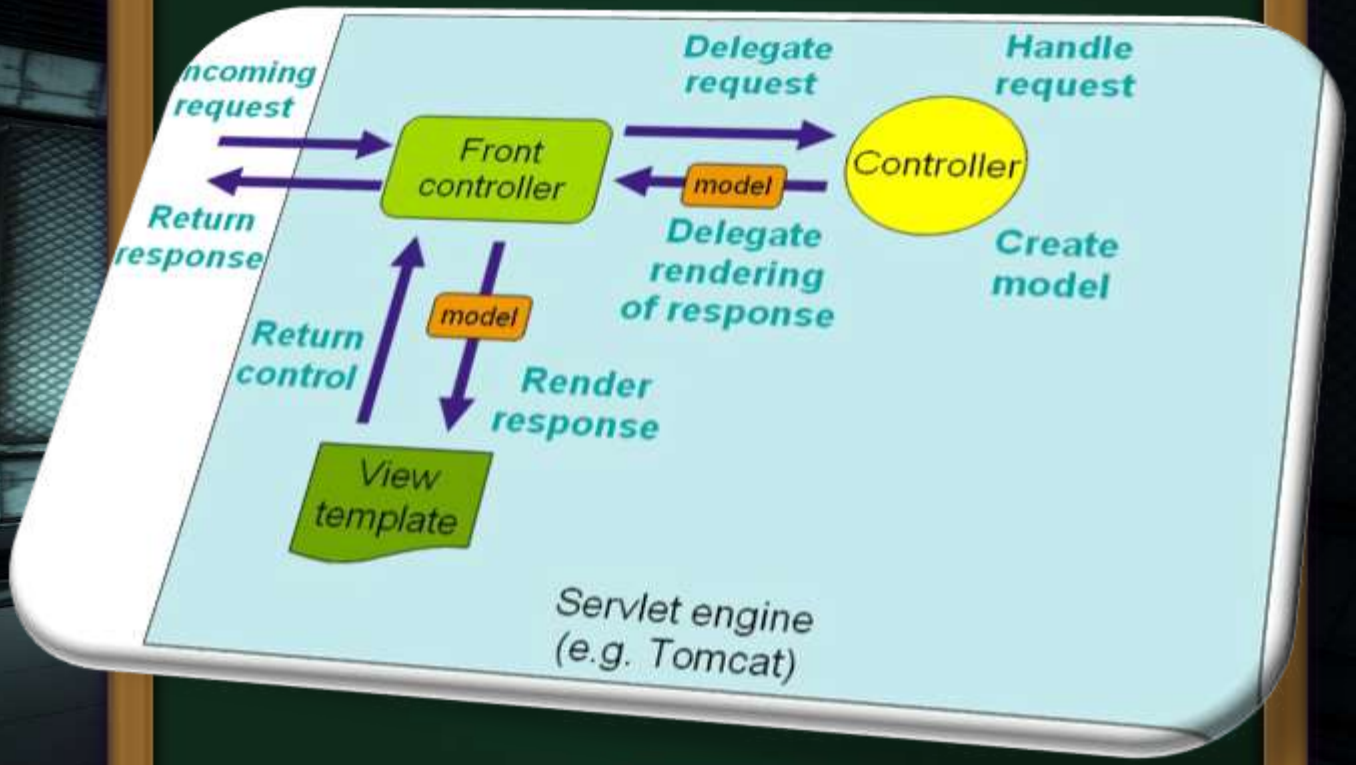
- Personalizable Locale y resolución tema
- Potente librería de etiquetas JSP: Spring tag library

DispatchServlet

¿Qué es DispatcherServlet?

- Juega el papel del Front controller en Spring MVC, similar al filtro Filter Dispatcher de Struts2
- Coordina las peticiones web (HTTP request) y ciclo de vida
- Observa el request y aplica el controller apropiado según la url (handler mapping)
- Configurado en web.xml

¿Qué es DispatcherServlet?



Cómo Funciona Spring

El Front Controller DispatchServlet recibe una solicitud HTTP del navegador

El DispatchServlet aplica un Controlador basado en la URL (Handler mapping) y asigna el request al Controller

El Controller se relaciona con componentes de la lógica de negocio y envía datos a la vista usando el objeto Model

El Controller retorna/asigna el nombre de la vista lógica a despachar

Se selecciona un ViewResolver, el cuál aplica un determinado tipo de vista (JSP, PDF, Excel, etc.)

Finalmente la vista es mostrada al cliente usando los valores del objeto Model

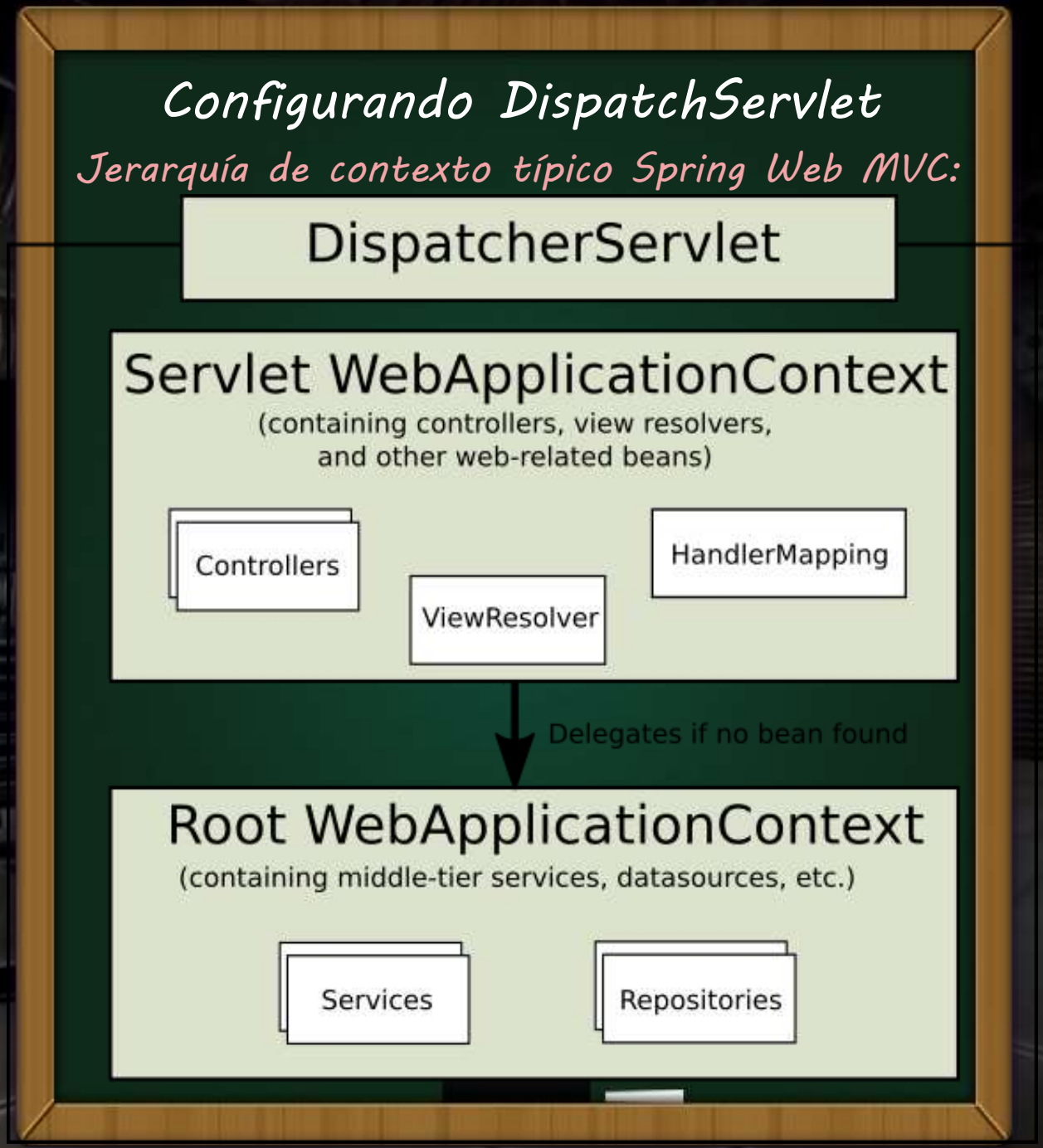
Configurando DispatchServlet

- Se configura en el archivo web.xml como cualquier Servlet:

```
<web-app>

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

   <servlet-mapping>
      <servlet-name>ejemplo</servlet-name>
      <url-pattern>/</url-pattern>
   </servlet-mapping>

</web-app>
```

Configurando DispatchServlet

Jerarquía de contexto típico Spring Web MVC:

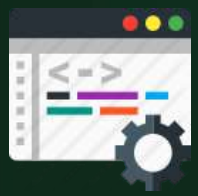


Configuración del Contexto de Spring

Configuración Contexto de Spring

Podemos configurar dos (o más) archivos de configuración Spring en el web.xml:

- ✓ Un archivo de contexto "root", define los beans asociado al contexto de aplicación y lógica de negocio (datasource, hibernate, daos, transacciones), compartidos por todos los servlets y filters



- ✓ Un archivo de contexto "web", uno por cada DispatchServlet, define los beans asociado al contexto web (controllers, interceptors, view resolver etc)

Configuración Contexto de Spring

Ejemplos de archivos de contexto "root", pueden ser definidos en el archivo web.xml o bien se pueden importar en otros archivos de contexto, ejemplos

- root-context.xml (otros nombres típicos applicationContext.xml o app-config.xml)
- applicationContext-hibernate.xml
- applicationContext-dataSource.xml
- applicationContext-security.xml

Configuración Contexto de Spring

En general los nombres que usemos de estos archivos de configuración de spring, ya sea para “root” o “web” puede ser cualquiera que queramos, pero siempre debe quedar bien definido en el archivo web.xml

- Típicamente para "web context" usamos servlet-context.xml (a veces se usa mvc-config.xml o dispatcher-servlet.xml)
- Típicamente para “root context" usamos root-context.xml (a veces se usa app-config.xml o applicationContext.xml)

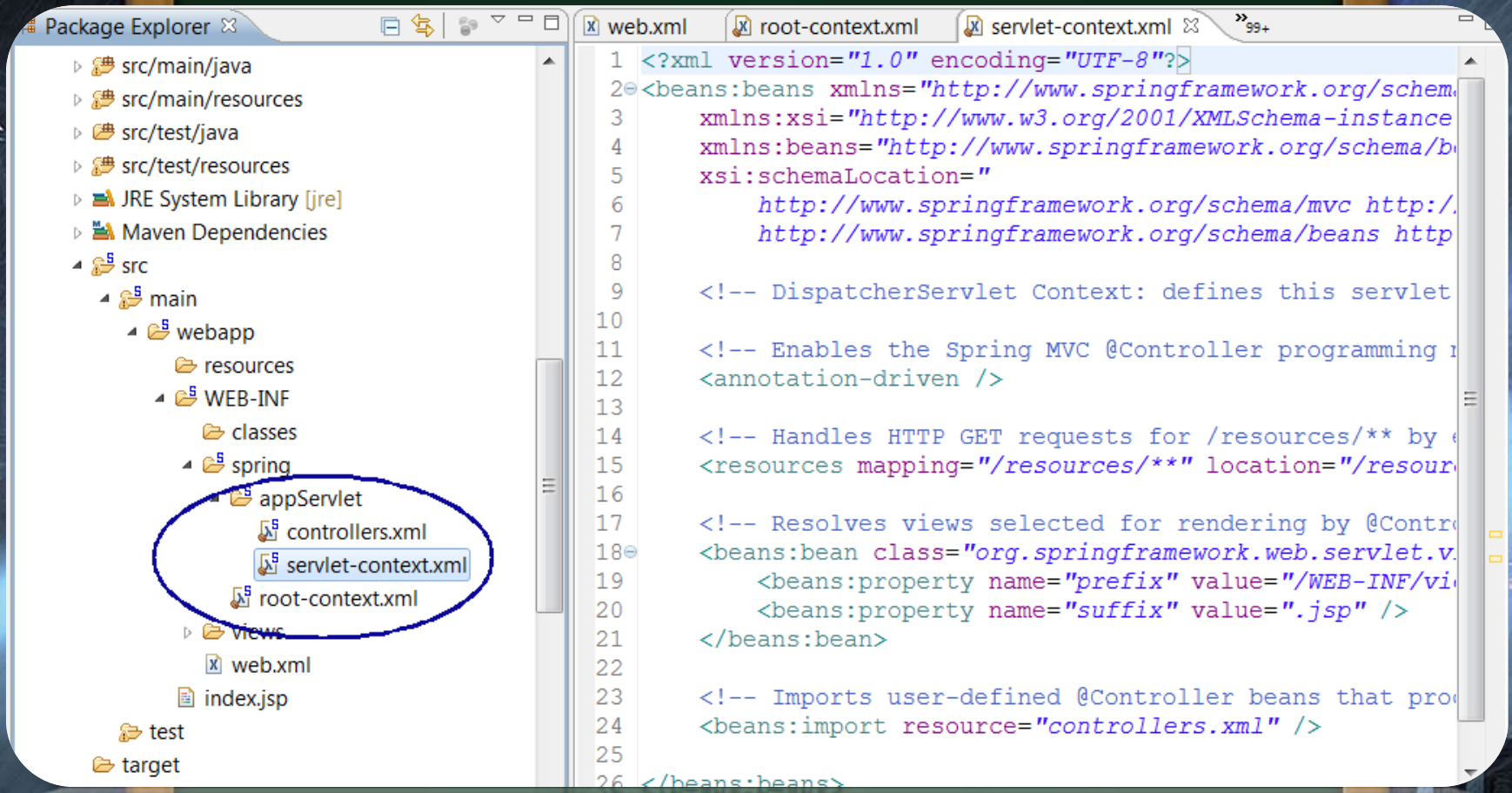
Ejemplo Configuración Contexto de Spring en web·xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <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>
```

Root context

Web context

Ejemplo Configuración Contexto de Spring en web.xml



Nombre/Ubicación por defecto del archivo de contexto Web

- El nombre y ubicación del archivo XML de contexto "Web" es típicamente configurado en el DispatcherServlet, como parámetro init (contextConfigLocation) en web.xml
  - Como hemos visto en la diapositiva anterior
- En caso de que no se encuentre configurado, Spring busca el archivo de contexto web por defecto, es decir:

/WEB-INF/[servlet-name]-servlet.xml

Nombre/Ubicación por defecto del archivo de contexto Web

El archivo de configuración web, lleva el nombre del servlet (<servlet-name>) seguido de "-servlet.xml". En este caso, appServlet-servlet.xml, el nombre del Servlet es “appServlet” lo podemos ver configurado en el web.xml mencionado a continuación:

```
<!-- Spring buscara el archivo de contexto web por defecto:
     /WEB-INF/appServlet-servlet.xml -->
<servlet>
  <servlet-name>appServlet</servlet-name>
  <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Archivo de Contexto Web

Aquí se definen los beans Spring Framework MVC, ejemplo

- Controllers
- Handler mappings
- Interceptors
- View resolvers
- Locale resolvers
- Theme resolvers
- Multipart file resolver
- Hanlder exception resolvers

Controllers
Spring MVC

¿Qué hace un controlador?

- Los Controladores proporcionan acceso a la lógica de negocio
  - Normalmente, un controlador delega el proceso de lógica de negocio a un conjunto de componentes de servicios

    Los servicios a su vez acceden a las bases de datos mediante la interfaz Dao (Objeto de Acceso a Datos)
- Los Controladores reciben parámetros del usuario (input) y lo convierten en un objeto del modelo, poblando en sus atributos los datos enviados, como resultado de la lógica de negocio

@Controller: Controladores con anotaciones

Recién en Spring 2.5 se introducen los controladores en base a anotaciones

- No necesita extender de ninguna clases base controllers ni tampoco implementar la interfaz Controller
- Los controladores en base a anotaciones tampoco tienen directa dependencia de los Servlet o Portlet
- En otras palabras los controladores anotados son clases POJO anotados, no heredan de nada
- Usamos las anotación @Controller para definir nuestos controladores en Spring

Ejemplo Controlador Anotado

- Controlador basado en anotación no tiene que extender de las clases base Spring ni implementar interfaces específicas::

```
@Controller
public class HolaMundoController {

    // No hay reglas ni limitaciones en el nombre y firma del método
    @RequestMapping("/hola")
    public ModelAndView holaMundo() {
        ModelAndView mav = new ModelAndView();
        // asignamos la vista (sólo el nombre)
        mav.setViewName("inicio");
        // agregamos un atributo a la vista
        mav.addObject("mensaje", "Hola Mundo!");
        return mav;
    }
}
```

Habilitar uso de anotaciones En los Controladores

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

    <mvc:annotation-driven />

// ...
</beans>
```

Auto-detectar Controladores Anotados

- A través del escáner de componentes, todo bean controller anotado con @Controller son automáticamente detectados, y las instancias de éstos son automáticamente creadas
- No hay necesidad de declararlos en el archivo de configuración XML de contexto

```
<beans ...
    <mvc:annotation-driven />
    <context:component-scan base-
package="com.formacionbdi.ejemplos.catalogo.web"/>
// ...
</beans>
```

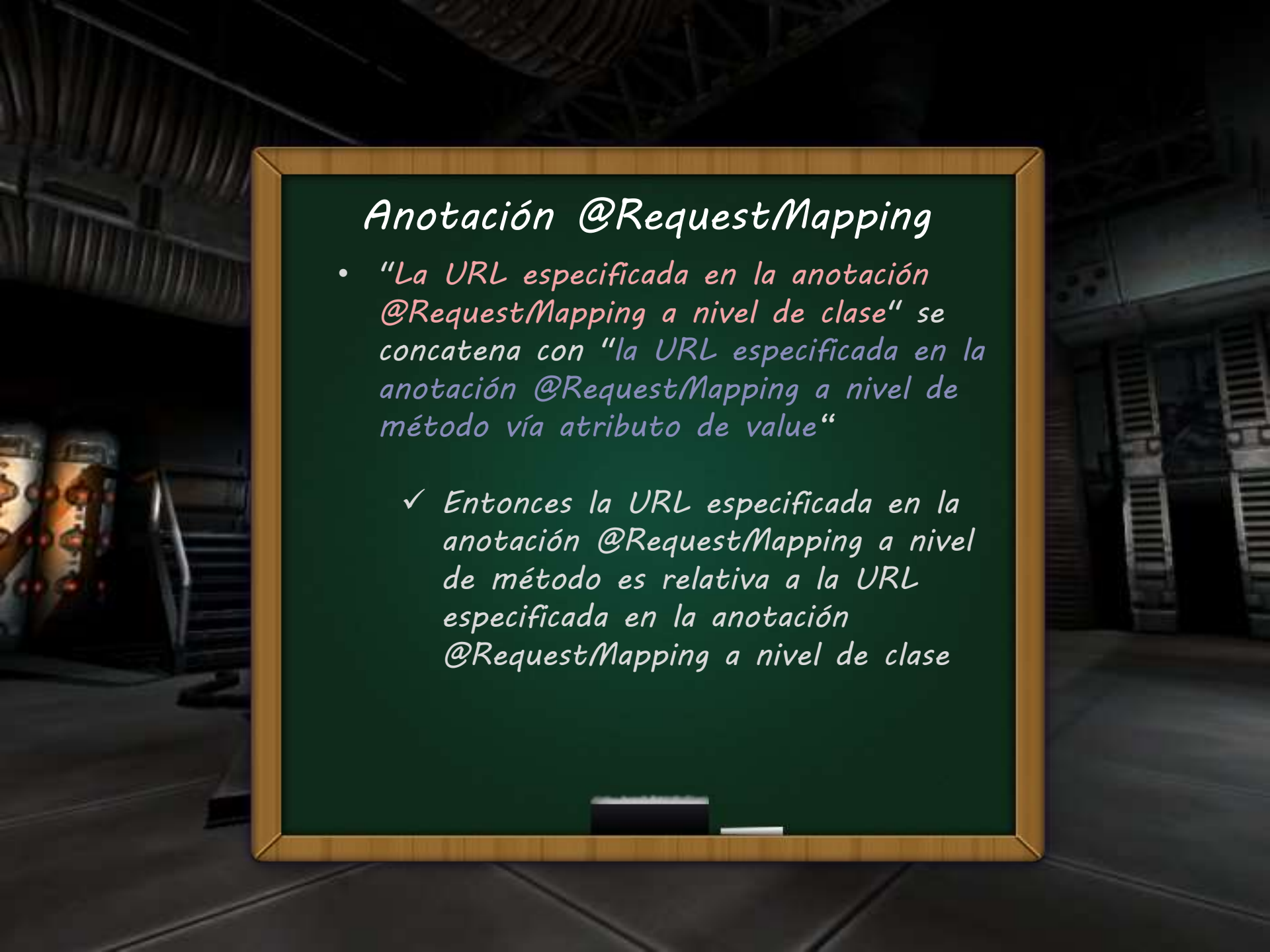
Auto-detectar Controladores Anotados

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

 <mvc:annotation-driven />
    <context:component-scan base-
package="com.formacionbdi.ejemplos.catalogo.web"/>

// ...
</beans>
```

Anotación @RequestMapping

- @RequestMapping es usada para mapear las URLs hacia métodos handler de una clase Controller

- La anotación @RequestMapping se puede especificar:
  - A nivel de la clase
  - A nivel de método


Anotación @RequestMapping
• "La URL especificada en la anotación @RequestMapping a nivel de clase" se concatena con "la URL especificada en la anotación @RequestMapping a nivel de método vía atributo de value"
✓ Entonces la URL especificada en la anotación @RequestMapping a nivel de método es relativa a la URL especificada en la anotación @RequestMapping a nivel de clase

Anotación @RequestMapping

```
@Controller
@RequestMapping("/agenda") // Usada a nivel de clase
public class AgendaController {

private IAgendaDao agendaDao;

    @Autowired
    public AgendaController(IAgendaDao agendaDao) {
        this.agendaDao = agendaDao;
    }

    // Ejemplo http://locahost:8080/miapp/agenda
    @RequestMapping(method = RequestMethod.GET) // Usada a nivel de método
    public Map<String, Agenda> getTodas() {
        return agendaDao.listarTodas();
    }

    // Ejemplo http://locahost:8080/miapp/agenda/4 o
    // http://locahost:8080/miapp/agenda/5
    @RequestMapping(value="/{day}", method=RequestMethod.GET) //Usada a nivel método
    public Map<String, Agenda> getPorDia (@PathVariable
                                          @DateTimeFormat(iso=ISO.DATE) Date dia,
                                          Model model) {

        return agendaDao.getReunionesPorDia(dia);
    }
```

... Continúa en la siguiente página

Anotación @RequestMapping

... Continuación de la diapositiva anterior

```
    // Ejemplo http://locahost:8080/miapp/agenda/nueva
    // Muestra al usuario el formulario en pantalla
    @RequestMapping(value="/nueva", method=RequestMethod.GET)
    public Agenda form() {
        return new Agenda();
    }

    // Ejemplo http://locahost:8080/miapp/agenda/nueva
    // Procesa el envío del formulario
    @RequestMapping(value="/nueva", method=RequestMethod.POST)
    public String crear(@Valid Agenda agenda, BindingResult result) {

        if (result.hasErrors()) {
            return "agenda/nueva";
        }
        agendaDao.guardar(agenda);
        return "redirect:/agenda";
    }
}
```

Anotación @RequestMapping sólo a nivel de métodos

"No es requerida la anotación @RequestMapping a nivel de clase. Sin esta, todas las rutas son absolutas, y no relativas

```
@Controller
public class ClinicController {

    private Clinic clinic;

    @Autowired
    public ClinicController(Clinic clinic) {
        this.clinic = clinic;
    }

    // Maneja http://locahost:8080/mi-app/
    @RequestMapping("/")
    public void welcomeHandler() {}

    // Maneja http://locahost:8080/mi-app/vets
    @RequestMapping("/vets")
    public ModelMap vetsHandler() {
        return new ModelMap(this.clinic.getVets());
    }
}
```

URI Template (Plantillas URL)

¿Qué es una URI Template?

- Una URI Template es un string URL que contiene uno o más nombres de variables (variables de url)
  - Variables que tienen forma de {nombreDeVariable}

```
@RequestMapping(value="/cliente/{clienteId}"))
```

  - El nombreDeVariable necesita ser pasado como argumento del método handler acompañado de la anotación @PathVariable

```
public String findCliente(@PathVariable String clienteId, Etc ...)
```

¿Qué es una URI Template?

- Cuando se substituyen los valores de esas variables, el URI template se convierte en una URL concreta

/cliente/3
/cliente/5

- El nombreDeVariable necesita ser pasado como argumento del método handler acompañado de la anotación @PathVariable
- Entonces, el request URI es comparado con alguna plantilla URI (URI Template) y si coincide se invoca el metodo handler correspondiente

¿Qué es una URI Template?

- Supongamos el siguiente request URL

http://locahost:8080/mi-appspring/cliente/3

- El valor 3 será capturado en el argumento "clienteId" del tipo String.

```
@RequestMapping(value="/cliente/{clienteId}", method=RequestMethod.GET)
public String findCliente(@PathVariable String clienteId, Model model) {
   // Ahora podemos usar la variable clienteId
   // en nuestra lógica de negocio
   Cliente cliente = clienteDao.findPorId(clienteId);
   model.addAttribute("cliente", cliente);
   return "cliente/detalle";
}
```

¿Qué es una URI Template?

- Podemos usar multiples anotaciones @PathVariable para capturar multiples variable URI Template

- Supongamos el siguiente request URL

http://locahost:8080/mi-appspring/cliente/3/factura/5

  - El valor 3 será capturado en el argumento "clienteId" del tipo String
  - El valor 5 será capturado en el argumento "facturaId" del tipo String

¿Qué es una URI Template?

```
@RequestMapping(value="/cliente/{clienteId}/{facturaId}",
method=RequestMethod.GET)
public String findFactura(@PathVariable String clienteId,
                          @PathVariable String facturaId,
                          Model model) {

   Cliente cliente = clienteDao.findPorId(clienteId);
   Factura factura = cliente.getFactura(facturaId);
   model.addAttribute("factura", factura);
   return "cliente/factura";

}
```

Argumentos en Métodos Handler

Objetos auto-creados por Spring

Podemos usar cualquiera de estos objetos como argumentos del los métodos handler del controlador. Estos serán creados automaticamente por Spring y pasados al método:

- HttpServletRequest o HttpServletResponse
  - Objeto Request o response (Servlet API)
- HttpSession
  - Objeto Session (Servlet API)
- java.util.Locale (Configuración Regional)
  - Para obtener el objeto locale actual del request
- java.security.Principal
  - Usuario autenticado

@PathVariable y @RequestParam

@PathVariable

- Parámetros anotados para acceder a variables de las plantillas URL o URI template
- Extrae los datos desde el request URI
- http://host/catalogo/items/123
- Los valores de los parámetros son convertidos y pasados como argumentos en los métodos handler usando la anotación @PathVariable

@PathVariable y @RequestParam

@RequestParam(“name")

- Parámetros anotados para acceder a específicos parámetros del Servlet request.
- Extrae los datos desde request query parameters URL
- http://host/catalogo/items/?name=abc
- Los valores de los parámetros son convertidos y pasados como argumentos en los métodos handler usando la anotación @RequestParam

@PathVariable valores URI Path

```
// Usamos la anotación @PathVariable para poblar valores
// en parámetros de métodos del controlador
@Controller
@RequestMapping("/catalogo")
public class CatalogoController {

   // Ejemplo ../catalogo/4 o ../catalogo/10
   // 4 y 10 son convertidos a tipos int por Spring.
   @RequestMapping(value="/{prodId}")
   public String getData(@PathVariable int prodId,
                         ModelMap model) {
      Producto producto = productoDao.getProductoPorId(prodId);
      model.addAttribute("producto", producto);
      return "form";
   }
// ...
```

@RequestParam para valores Query Parameters

```
// Usamos la anotación @RequestParam para poblar
// valores query request en parámetros de métodos del controlador
@Controller
@RequestMapping("/catalogo")
public class CatalogoController {

   // Ejemplo ../catalogo?pordId=4 or ../catalogo?pordId=10.
   // 4 y 10 son convertidos a tipos int por Spring.
   public String getData(@RequestParam("prodId") int prodId,
                         ModelMap model) {
      Producto producto = productoDao.getProductoPorId(prodId);
      model.addAttribute("producto", producto);
      return "form";
   }
// ...
```

Request Header y Body

@RequestHeader("name")	@RequestBody
• Parámetros anotados para acceder a específicas cabeceras HTTP (request HTTP headers)	• Parámetros anotados para acceder al cuerpo HTTP (request HTTP body) • Los valores de los parámetros son convertidos y pasados como argumentos en los métodos handler usando HttpMessageConverter

@RequestBody

```
// La anotación @RequestBody indica que un argumento del método
// deberá ser poblado con el valor del HTTP request body
@RequestMapping(value = "/algunaurl", method =
RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer)
throws IOException {
    writer.write(body);
}
```

Tipos de objeto Model como argumentos en Métodos Handler

Implícitos tipos de Models como argumentos Handler

Un objeto Model es creado por Spring y es pasado como argumento en los métodos handler del controlador

- Podemos asignar atributos a la vista/form vía llave/valor

Los objetos Models son expuesto en las vistas (ej. archivo jsp)

- Vista puede acceder a estos modelos (atributos del objeto model) usando lengiaje de expreciones (EL)

Implícitos Models como argumentos Handler

Tipos de Model soportados:

- java.util.Map: el más genérico y típico de todos
- org.springframework.ui.Model: contenedor de atributos para la vista
- org.springframework.ui.ModelMap: Soporta invocación de métodos en cadena y auto-generación de nombres de atributos del modelo

Ejemplo objeto Implícito Model como argumento Handler

```
import org.springframework.ui.Model;

@RequestMapping(value = "/buscar-hoteles", method =
RequestMethod.GET)
public String listar(SearchCriteria criteria, Model model) {

   List<Hotel> hotels = bookingService.findHotels(criteria);
   // Ageregamos un atributo al objeto model. La vista podrá acceder
   // al atributo "hotelList" vía ${hotelList}
   model.addAttribute("hotelList", hotels);
   // Retornamos el nombre lógico de la vista "hotels/list",
   // lo que resulta el despliegue de la vista "hotels/list.jsp".
   return "hotels/list";
}
```

Vista (JSP) que accede al objeto Model

```
<table class="summary">
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Dirección</th>
      <th>Ciudad, Estado</th>
      <th>Detalle</th>
    </tr>
  </thead>
  <tbody>
  <c:forEach var="hotel" items="${hotelList}">
    <tr>
      <td>${hotel.name}</td>
      <td>${hotel.address}</td>
      <td>${hotel.city}, ${hotel.state}, ${hotel.country}</td>
      <td><a href="hotels/${hotel.id}">Ver Hotel</a></td>
    </tr>
  </c:forEach>
  <c:if test="${empty hotelList}">
    <tr>
      <td colspan="5">No se han encontrado hoteles</td>
    </tr>
  </c:if>
  </tbody>
</table>
```

Ejemplo objeto ModelMap como argumento Handler

```
import org.springframework.ui.ModelMap;

@RequestMapping("/deposito")
public String deposito(
                @RequestParam("numeroCuenta") int numeroCuenta,
                @RequestParam("monto") double monto,
                ModelMap modelMap) {

   cuentaDao.depositar(numeroCuenta, monto);
   // Objeto ModelMap permite llamada en cadena
   modelMap.addAttribute("numeroCuenta", numeroCuenta)
                .addAttribute("balance",
                                cuentaDao.getBalance(numeroCuenta));
   return "success";
}
```

Tipos de retorno en Métodos Handler (para Selección de Vista)

Objeto ModelAndView (como tipo de retorno)

```
@Controller
@RequestMapping("/carro")
public class CarroDeComprasController{

   @RequestMapping(value="/ver")
   public ModelAndView verCarro() {
      List itemsCarro = // Obtenemos la lista de items del carro de compras

      //Creamos el objeto ModelAndView y asignamos el nombre de la vista
      //lo que resulta el despliegue de la vista "catalogo/verCarro.jsp".
      ModelAndView mav = new ModelAndView("catalogo/verCarro");

      // Agregamos atributos al objeto ModelAndView
      // La vista podrá acceder al atributo "itemsCarro" vía ${itemsCarro}
      mav.addObject("itemsCarro", itemsCarro);

      // Retornamos el objeto ModelAndView
      return mav;
   }
}
```

Objeto String (como tipo de retorno)

Retornamos un String que es interpretado como el nombre lógico de la vista

Es la forma más comúnmente utilizada

```
@Controller
@RequestMapping("/carro")
public class CarroDeComprasController{

   @RequestMapping(value="/ver")
   public String verCarro(Model model) {
      List itemsCarro = // Obtenemos la lista de items del carro de compras

      // Agregamos atributos al objeto Model pasado por argumento
      // La vista podrá acceder al atributo "itemsCarro" vía ${itemsCarro}
      model.addAttribute("itemsCarro", itemsCarro);

      // Retornamos el objeto String con el nombre de la vista
      // lo que resulta el despliegue de la vista "catalogo/verCarro.jsp".
      return "catalogo/verCarro";
   }
}
```

void
(como tipo de retorno)

Cuando el nombre de la vista es implícita, es decir no se define en ninguna parte en el método handler, es determinada mediante RequestToViewNameTranslator, vía Mapping URL (Request Mapping)

```
// El nombre de la vista se establece implícitamente a "nombreVista123"
@RequestMapping(value="/nombreVista123", method=RequestMethod.GET)
public void usandoRequestToViewNameTranslator(Model model) {
   model.addAttribute("foo", "bar");
   model.addAttribute("fruit", "apple");
}
```

Tipos de retorno en Métodos Handler (para crear Respuesta)

Anotación de método @ResponseBody

Si el método es anotado con @ResponseBody, se declara el tipo de retorno como String y su contenido es almacenado en la respuesta dentro del cuerpo HTTP (response HTTP body)

No hay selección de vista

Comúnmente usado para peticiones del tipo AJAX y RESTful (XML, JSON)

```
@RequestMapping(value="/response/annotation", method=RequestMethod.GET)
public @ResponseBody String responseBody() {
  return "Un String ResponseBody";
}
```

Anotación de método @ResponseBody

Provee acceso hacia las cabeceras HTTP de la Respuesta (Servlet Reponse HTTP Headers)

El entity body será convertido hacia el stream de salida (response stream) usando HttpMessageConverter

```
@RequestMapping(value="/response/entity/headers", method=RequestMethod.GET)
public ResponseEntity<String> responseEntityCustomHeaders() {
   HttpHeaders headers = new HttpHeaders();
   headers.setContentType(MediaType.TEXT_PLAIN);
   return new ResponseEntity<String>("Un String ResponseBody con cabecera
                                      personalizada Content-Type=text/plain",
                                      headers, HttpStatus.OK);
}
```

Views vs @ResponseBody

Views vs @ResponseBody

Dos esquemas para desplegar una respuesta (render o salida)

- ViewResolver + View
- HttpMessageConverter

Trabajan de formas diferentes

- Dibujar una vista vía el retorno del nombre lógico de la vista como un String
- Escribir un mensaje en la respuesta vía retorno de @ResponseBody o ResponseEntity

Views vs @ResponseBody

Cuál usar?

- Usamos ViewResolver + View para generar documentos en el navegador, ejemplo HTML, PDF, XLS etc
- Usamos @ResponseBody para el intercambio de datos con web servicios y Ajax, ejemplo JSON, XML, etc

Anotación @SessionAttributes

Anotación @SessionAttributes

Define atributos de sesión utilizados por un controlador

Contiene una lista de nombres de atributos del objeto view model, cuyos valores u objetos son almacenados transparentemente en la sesión HTTP, típicamente beans asociados a un formulario (objeto comando o de formulario) para que sean persistentes y accesibles entre request (solicitudes posteriores)

Ideal para formularios tipo Wizard

```
@Controller
@RequestMapping("/editarUsuario")
@SessionAttributes("usuario")
public class EditarUsuarioController {
   // ...
}
```

Handler Mapping (Gestores de Mapeos)

Clase DefaultAnnotationHandlerMapping

En versiones anteriores de Spring 2.5, nosotros los desarrolladores estábamos obligados a definir un bean HandlerMapping (SimpleUrlHandlerMapping) en el archivo de contexto de aplicación Web para mapear las solicitudes/request HTTP web entrantes a los controladores

Desde la versión Spring 2.5 en adelante, el Front Controller de Spring (DispatcherServlet) usa la clase DefaultAnnotationHandlerMapping por defecto para mapear los controladores, usando anotaciones

Clase DefaultAnnotationHandlerMapping

DefaultAnnotationHandlerMapping busca las anotaciones @RequestMapping en los controladores anotados con @Controllers

El elemento <mvc:annotation-driven /> registra la clase DefaultAnnotationHandlerMapping (por lo que NO es necesario declarar el beans de forma explícita en el XML del contexto web)

Sin embargo, en caso de que necesitemos personalizar las configuraciones de esta clase, ahí si que será necesario definir el bean DefaultAnnotationHandlerMapping con los valores de atributos personalizados

Atributos de la clase DefaultAnnotationHandlerMapping

Tres casos típicos en que podríamos necesitar personalizar las configuraciones de la clase DefaultAnnotationHandlerMapping:

1.- Interceptors: Asignar Interceptores Web HTTP

2.- defaultHandler: Asignar un Controlador por defecto a utilizar, en caso de que falle el handler mapping y no encuentre un controlador asignado al request URL.

3.- Order: Asignar Orden de importancia en ejecución

Interceptores

Agregar un interceptor al mapping

El siguiente ejemplo muestra cómo agregar un interceptor en el contexto web de spring, utilizando la etiqueta <mvc:interceptors>

```
<beans>
 <!-- Configuración de interceptores basado en URI -->
   <mvc:interceptors>
      <mvc:interceptor>
         <mvc:mapping path="/**" />
         <bean id="loggingInterceptor"
class="com.formacionbdi.ejemplo.LoggingInterceptor" />
      </mvc:interceptor>
   </mvc:interceptors>
<beans>
```

¿Qué hace un Interceptor?

Los Interceptores son útiles cuando se requiere aplicar funcionalidades/procesos específicas para ciertos request, es decir son fragmentos de códigos reutilizables que interceptan una petición web (HTTP REQUEST) para agregar funcionalidad transversal

- Ejemplo: Autenticación, autorización, control de transacciones, logging, validaciones, etc

¿Qué hace un Interceptor?

Los Interceptores deben implementar la interfaz HandlerInterceptor o extender de la clase abstracta HandlerInterceptorAdapter

- Método boolean preHandler(..): Cuando este método retorna true, el DispatcherServlet continúa con la ejecución del controlador y si tiene más interceptores asociados continúa con la ejecución en cadena; cuando retorna false, el DispatcherServlet asume que el interceptor se hace cargo del requests (y, por ejemplo, muestra o redirige hacia una determinada vista) y NO continúa ejecutando el controlador ni los demás interceptores asociados (si es que los tiene)
- Método void postHandler(..) implementar algo después de que se haya invocado el handler (Método handler request del controlador)

Ejemplo interceptor

Logger Interceptor

```
public class LoggingInterceptor extends HandlerInterceptorAdapter {

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

   public boolean preHandle(HttpServletRequest request,
                    HttpServletResponse response,
                    Object handler) throws Exception {
      logger.info("LoggingInterceptor: preHandle() ALGÚN PROCESO ANTES");
      return true;
   }

   public void postHandle(HttpServletRequest request,
                    HttpServletResponse response,
                    Object handler,
                    ModelAndView modelAndView) throws Exception {
      logger.info("LoggingInterceptor: postHandle() ALGÚN PROCESO DESPUÉS");
   }
}
```

Ejemplo interceptor

Configuración Logger Interceptor en Web Context Spring XML

```
<!-- Configuración de interceptores basado en URI -->
  <mvc:interceptors>
    <mvc:interceptor>
      <mvc:mapping path="/**" />
      <ref bean="loggingInterceptor" />
    </mvc:interceptor>
  </mvc:interceptors>

<!-- Interceptores -->
<bean id="loggingInterceptor"
class="com.formacionbdi.ejemplo.LoggingInterceptor" />
```

Resolviendo las Vistas

Resolvedores de vistas

Todo método handler de los controladores en Spring Web MVC deben resolver/detectar el nombre lógico de la vista

- Explícitamente (por ejemplo, mediante el retorno de un String, View o ModelAndView)
- Implícitamente (es decir, en base de convenciones vía URL mapping, tipo de retorno void)

Resolvedores de vistas

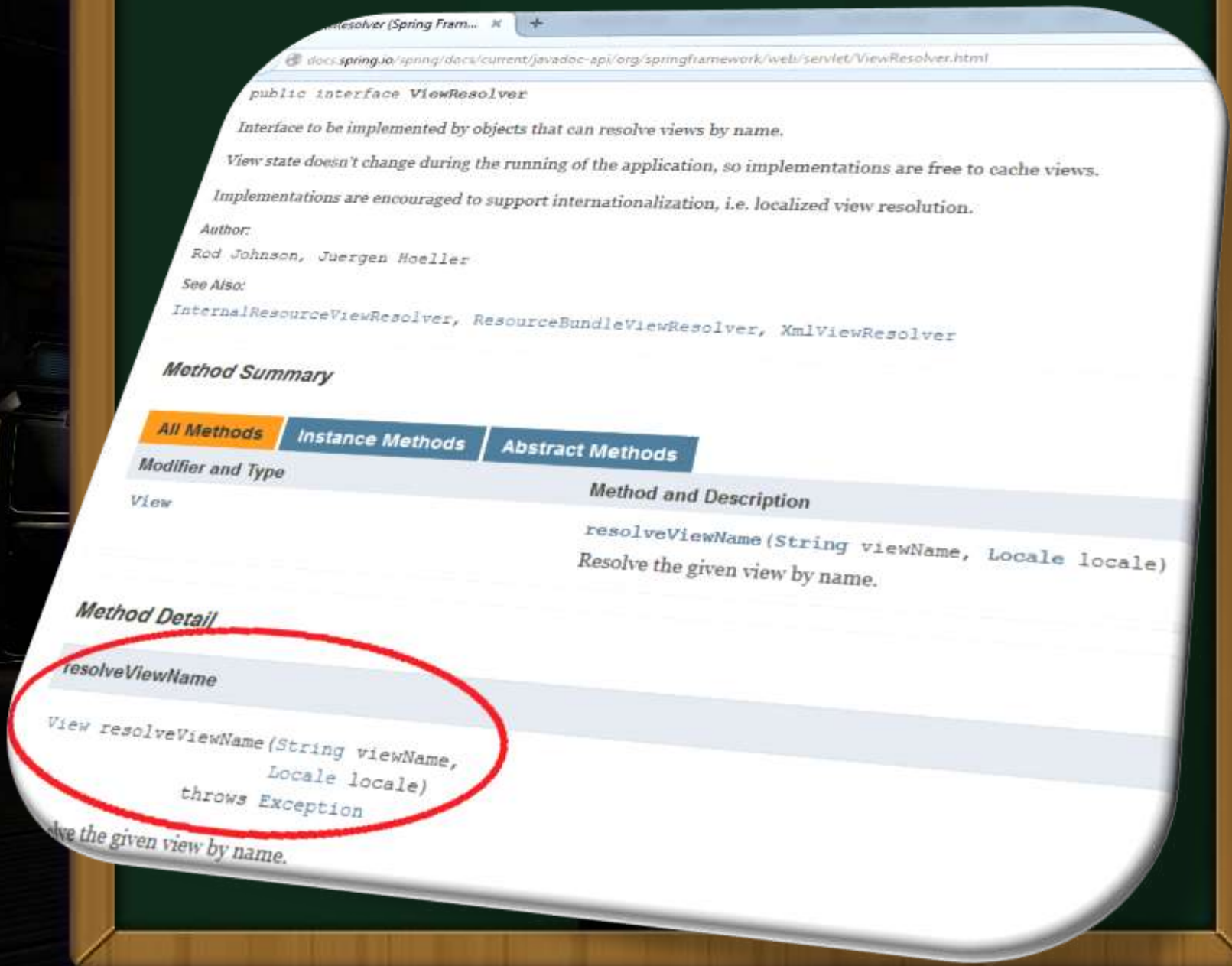
Las Vistas en Spring son abordadas por un nombre lógico y se resuelven por medio de un view resolver (resolución de la vista)

Spring pasa por una serie de view resolvers, es decir objetos encargados de resolver la vista o resolvedores o resolutores (en secuencia), hasta que se encuentre un view resolver adecuado que pueda manejar el nombre lógico de la vista

Spring por defecto, incluye un conjunto de view resolvers (resolvedores de vistas)

Cada view resolvers implementa la interfaz ViewResolver

Interfaz ViewResolver




ViewResolvers incluidos en Spring
UrlBasedViewResolver
InternalResourceViewResolver
BeanNameViewResolver
ResourceBundleViewResolver
AbstractCachingViewResolver
XmlViewResolver
ContentNegotiatingViewResolver

UrlBasedViewResolver

Una simple implementación de la interfaz ViewResolver que efectúa una directa resolución del nombre lógico o simbólico de la vista asociada a una ruta (Path o View URL), sin una definición explícita.

Esto se aplica si el nombre simbólico de la vista coincide con el nombre de algún recursos de vista (archivos jsp) directamente, sin la necesidad de asignaciones dedicadas para definir cada vista.

El nombre simbólico o lógico es la única parte del nombre del archivo de recurso.

UrlBasedViewResolver

Solo es necesario configurar en el XML el prefijo y sufijo, donde el prefijo corresponde a la ruta o path donde se encuentran nuestros recursos/paginas JSP y el sufijo la extensión: prefix="/WEB-INF/jsp/", suffix=".jsp"

Supongamos que en el controlador retornamos como nombre de la vista "test", entonces el view resolver hará un forward del request (RequestDispatcher) enviando el request hacia el archivo JSP, cargando la vista /WEB-INF/jsp/test.jsp

UrlBasedViewResolver

Además como característica especial del UrlBasedViewResolver, redirigir direcciones URL se pueden especificar mediante el prefijo "redirect:".

Por ejemplo: "redirect: miAccion.do" desencadenará una redirección a la URL dada, en lugar de la resolución del nombre de la vista.

Esto se utiliza normalmente para redirigir hacia una URL de un controlador después de terminar algún proceso o tarea de un método handler del controlador.

```
<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>
```

InternalResourceViewResolver

Una subclase de UrlBasedViewResolver que soporta InternalResourceView (es decir, Servlets y JSP) y subclases como JstlView y TilesView.

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

BeanNameViewResolver

Aplicación simple de ViewResolver que interpreta el nombre de la vista como el nombre del vean (en el archivo XML del contexto web).

En otras palabras, provee funcionalidad para obtener las vistas desde un BeanFactory

En el siguiente ejemplo, después de retornar "bienvenida" como nombre lógico de la vista, el BeanNameViewResolver obtendrá el bean de la vista desde el contexto y usará ese bean para el render (desplegar) de la página JSP

```
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
   <bean id="bienvenida"
class="org.springframework.web.servlet.view.JstlView">
   <property name="url"><value>/WEB-
INF/jsp/bienvenida.jsp</value></property>
</bean>
```

ResourceBundleViewResolver

El resolutor ResourceBundleViewResolver lee y analiza un archivo .properties de recurso (ResourceBundle) identificado por su nombre base (basename), y para cada vista supuestamente a resolver, usa el valor de la propiedad [nombre_vista].(clase) como la clase de vista (clase que representa la vista) y en el valor de la propiedad [nombre_vista].url como la ruta o path o url hacia la vista.

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

resource bundle

```
# archivo views.properties ubicado en el classpath
welcomeRedirect.(class)=org.springframework.web.servlet.view.RedirectView
welcomeRedirect.url=welcome.jsp
reservationSuccessRedirect.(class)=org.springframework.web.servlet.view.RedirectView
reservationSuccessRedirect.url=reservationSuccess.jsp
```

ResourceBundleViewResolver

El resolutor ResourceBundleViewResolver lee y analiza un archivo .properties de recurso (ResourceBundle) identificado por su nombre base (basename), y para cada vista supuestamente a resolver, usa el valor de la propiedad [nombre_vista].(clase) como la clase de vista (clase que representa la vista) y en el valor de la propiedad [nombre_vista].url como la ruta o path o url hacia la vista.

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

resource bundle

```
# archivo views.properties ubicado en el classpath
welcomeRedirect.(class)=org.springframework.web.servlet.view.RedirectView
welcomeRedirect.url=welcome.jsp
reservationSuccessRedirect.(class)=org.springframework.web.servlet.view.RedirectView
reservationSuccessRedirect.url=reservationSuccess.jsp
```

ContentNegotiatingViewResolver

Este es un View Resolver especial, es decir no es un view resolver en si mismo, si no que delega a otro ViewResolver mas adecuado según un media type enviado en el request (XLS, PDF, XML, JSON etc)

```
<bean id="contentNegotiatingResolver"
class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
   <property name="order"
      value="#{T(org.springframework.core.Ordered).HIGHEST_PRECEDENCE}" />
   <property name="mediaTypes">
      <map>
         <entry key="html" value="text/html" />
         <entry key="pdf" value="application/pdf" />
         <entry key="xsl" value="application/vnd.ms-excel" />
         <entry key="xml" value="application/xml" />
         <entry key="json" value="application/json" />
      </map>
   </property>
</bean>
```

Orden en View Resolver

Podemos encadenar view resolvers agregando más de un view resolver en nuestro xml, si es necesario, podemos establecer el atributo "order" para especificar el orden en que serán llamados

- Cuanto mayor es el atributo "order", será posicionado al final de la cadena.
- InternalResourceViewResolver está posicionado de forma automática como el último ViewResolver (en caso de que no se le haya asignado el atributo "order") y se recomienda que siempre esté al final de la cadena.

Orden en View Resolver

Si un específico view resolver no resulta en una vista, Spring continúa con la inspección de los demás hasta que se resuelva una vista

En el siguiente ejemplo, la cadena de view resolvers consiste en dos resolvers, un InternalResourceViewResolver, el cual siempre y automáticamente es posicionado la final de la cadena, y un XmlViewResolver para definir vistas con Excel

```
<bean id="excelViewResolver"
class="org.springframework.web.servlet.view.XmlViewResolver">
   <property name="order" value="1"/>
   <property name="location" value="/WEB-INF/views.xml"/>
</bean>

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

Tipos de Conversión

Tipos de Conversión

Las conversiones (Type conversion) suceden de forma automática:

- El componente "ConversionService" es utilizado en los lugares donde se requiera la conversión de tipos
  - ✓ @RequestParam, JavaBean, @PathVariable
  - ✓ @RequestHeader, @CookieValue
- El componente "HttpMessageConverter" usado para
  - ✓ @RequestBody, @ResponseBody, HttpEntity, ResponseEntity

Tipos de Conversión

- Toda conversión importante se produce internamente y automáticamente por Spring (out of the box)
  - ✓ Primitive, String, Date, Collection, Map

- También podemos declarar anotaciones para reglas de conversión
  - ✓ @NumberFormat, @DateTimeFormat

Convención sobre Configuración

(Generación automática de llaves)

Generación automática de llaves/nombres para ModelMap

```
@Controller
@RequestMapping("/carro")
public class EjemploShoppingCartController {

   // Generación automática de llaves en atributos para ModelMap
   @RequestMapping(value = "/ejemplo", method = RequestMethod.GET)
   public String handleRequest(ModelMap modelMap) {

      // Tome nota de que carroCompras es un List del tipo Item
      List<Item> carroCompras = new ArrayList<Item>();
      carroCompras.add(new Item("Panasonic  Pantalla LCD", 259990.0));
      carroCompras.add(new Item("Sony  Cámara digital DSC-W320B", 123490.0));

      Usuario usuario = new Usuario("Andrés Guzmán");

      // Esto es lo mismo que modelMap.addAttribute("itemList", carroCompras);
      // "itemList" es generado automáticamente como una llave en el Map
      modelMap.addAttribute(carroCompras);

      // Esto es lo mismo que modelMap.addAttribute("usuario", usuario);
      // "usuario" es generado automáticamente como una llave en el Map
      modelMap.addAttribute(usuario);

      // verCarro es el nombre lógico de la vista
      return "verCarro";
   }
}
```

Generación automática de llaves/nombres para ModelAndView

```
@Controller
@RequestMapping("/carro")
public class EjemploShoppingCartController {

   // Generación automática de llaves en atributos para ModelMap
   @RequestMapping(value = "/ejemplo", method = RequestMethod.GET)
   public ModelAndView handleRequest() {

      // Tome nota de que carroCompras es un List del tipo Item
      List<Item> carroCompras = new ArrayList<Item>();
      carroCompras.add(new Item("Panasonic  Pantalla LCD", 259990.0));
      carroCompras.add(new Item("Sony  Cámara digital DSC-W320B", 123490.0));

      Usuario usuario = new Usuario("Andrés Guzmán");

      // verCarro es el nombre lógico de la vista
      ModelAndView mav = new ModelAndView("verCarro");

      // Esto es lo mismo que mav.addObject("itemList", carroCompras);
      // "itemList" es generado automáticamente como una llave en el Map
      mav.addObject(carroCompras);

      // Esto es lo mismo que mav.addObject("usuario", usuario);
      // "usuario" es generado automáticamente como una llave en el Map
      mav.addObject(usuario);

      return mav;
   }
}
```

Estrategia de Generación de llaves

Objetos escalares usamos el nombre de la clase (sin packages)	Objetos Collection
• x.y.Usuario la instancia agregada al ModelMap/ModelAndView tendrá el nombre de "usuario" • x.y.Registro la instancia agregada al ModelMap/ModelAndView tendrá el nombre de "registro"	• Un arreglo x.y.Usuario[] (con uno o mas elementos x.y.Usuario) agregado al ModelMap/ModelAndView se llamará "usuarioList" • Un arreglo x.y.Foo[] (con uno o mas elementos x.y.Foo) agregado al ModelMap/ModelAndView se llamará "fooList" • Una lista java.util.ArrayList<Usuario> (con uno o mas elementos x.y.Usuario) agregado al ModelMap/ModelAndView se llamará "usuarioList"

Convención sobre Configuración

(Generación del nombre de Vista lógico)

Handler sin definir nombre de vista

Un request URL, por ejemplo http://localhost/registro.html tiene como resultado el nombre lógico de la vista: "registro", resuelto por el componente DefaultRequestToViewNameTranslator que viene por defecto configurado en Spring

Una vez obtenido el nombre lógico de la vista mediante el request URL, se resuelve internamente el recurso de la vista (típicamente JSP) mediante el View Resolver InternalResourceViewResolver dando como resultado la vista JSP /WEB-INF/jsp/registro.jsp.

Handler sin definir nombre de vista

En el ejemplo observe que no hay declaración explícita de la vista (nombre lógico de la vista) ya que se resuelve de forma automática por convención de nombres en base a URL

```
@Controller
public class RegistroController {

   @RequestMapping(value = "/registro.html",  method = RequestMethod.GET)
   public ModelAndView registro(ModelAndView mav) {
      // NO se define ningún nombre de vista
      // agregamos datos al model para la vista...
      return mav;
   }
}
```

Configuración de Generación del Nombre de Vista lógica (Esto se hace automáticamente)

Este bean encargado de generar el nombre de vista por nosotros. Este es configurado automáticamente por Spring. En el ejemplo lo configuramos sólo para propósito de demostración

```
<beans>
   <!-- Este bean encargado de generar el nombre de vista por nosotros. Este es
configurado automáticamente por Spring. En el ejemplo lo configuramos sólo
para propósito de demostración. -->
   <bean id="viewNameTranslator"
class="org.springframework.web.servlet.view.DefaultRequestToViewNameTrans
lator"/>

   <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <property name="prefix" value="/WEB-INF/jsp/"/>
      <property name="suffix" value=".jsp"/>
   </bean>
</beans>
```

Tutorial sobre Namespaces XML

Necesidad de Namespace XML

Un documento XML podría tener conflictos de nombres	• Elementos que significan cosas diferentes podrían tener mismo nombre
Cambiar nombres de los elementos (para evitar conflictos) NO es buena opción	• Sobre todo si no somos los autores del XML
Algunos choques de nombres son inevitables	• Si ambos están dentro de la misma lengua y del vocablo estándar • SVG "set" vs MathML también "set"

Necesidad de Namespace XML

Agrupar nombres es una buena alternativa para evitar conflictos

- El procesador XSLT necesita saber cuales son las instrucciones XSLT y cuales son elementos que están dentro de una jerarquía o árbol

Declaración del Namespace

- Un prefijo (prefix) es asociado a una URI
- La asociación se define como un atributo dentro de un elemento:
  - ✓ xmlns:prefix
- xmlns el es Namespace y el prefijo es definido por el usuario, ejemplo:

```
<classes xmlns:XMLclass="http://www.brandeis.edu/rseg-0151-g">
  <XMLclass:syllabus>
    ...
  </XMLclass:syllabus>
</classes>
```

Namespace por defecto

- Declarado con el atributo xmlns pero sin prefijo
- Cualquier elemento sin prefijo pertenece al namespace por defecto

```
<classes xmlns:XMLclass="http://www.brandeis.edu/rseg-0151-g">
  <XMLclass:syllabus>
    ...
  </XMLclass:syllabus>
</classes>
```

Ejemplo: Namespace por defecto

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <head><title>Namespaces</title></head>
  <body>
    <h1 align="center">Una elipse y un rectángulo</h1>
    <svg xmlns="http://www.w3.org/2000/svg" width="12cm"
height="10cm">
      <ellipse rx="110" ry="130" />
      <rect x="4cm" y="1cm" width="3cm" height="6cm" />
    </svg>
    <p xlink:type="simple" xlink:href="elipses.html">
      Más sobre elipses
    </p>
    <p xlink:type="simple" xlink:href="rectangulos.html">
      Más sobre rectángulos
    </p>
    <hr/>
    <p>Última modificación 20 de septiembre, 2015</p>
  </body>
</html>
```

<html>, <head>, <title>, <body>, <h1>, <table>, , <p> etc pertenecen al Namespace por defecto

<svg>, <width>, <height>, <ellipse>, <rx>, <ry>, etc pertenecen al Namespace por defecto

Ejemplo Spring: Namespace por defecto

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/mvc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
   <annotation-driven />
   <resources mapping="/resources/**" location="/resources/" />

   <beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <beans:property name="prefix" value="/WEB-INF/views/" />
      <beans:property name="suffix" value=".jsp" />
   </beans:bean>

   <context:component-scan base-package="com.formacionbdi.ejemplos" />
</beans:beans>
```

<annotation-driven>, <resources> etc pertenecen al Namespace por defecto

Ejemplo 2 Spring: Namespace por defecto

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

   <mvc:annotation-driven />
   <mvc:resources mapping="/resources/**" location="/resources/" />

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

   <context:component-scan base-package="com.formacionbdi.ejemplos" />
<beans>
```

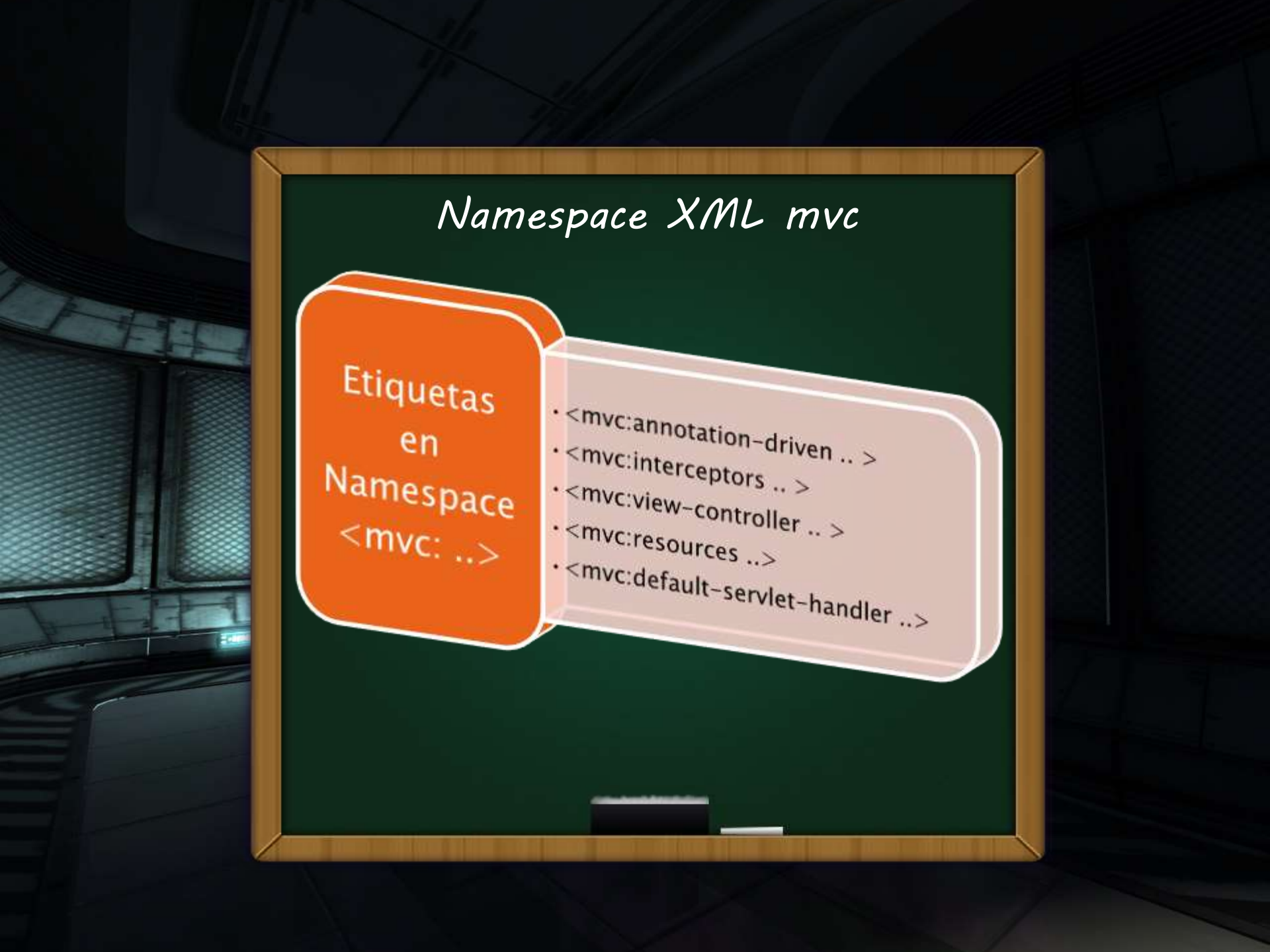
<bean> etc pertenecen al Namespace por defecto

Configuración <mvc: ··>

Namespace XML mvc

Spring 3 introduce el namespace XML **mvc** que simplifica la configuración de Spring MVC en aplicaciones web.

- En lugar de registrar los beans de bajo nivel tales como AnnotationMethodHandlerAdapter, simplemente usamos el namespace XML mvc y este se encraga de las construcciones por debajo (alto nivel).
- Es la configuración más recomendado, a menos que se quiera un control más preciso de la configuración a nivel de bean.


Namespace XML mvc
Etiquetas en Namespace <mvc: ..>
<mvc:annotation-driven .. >
<mvc:interceptors .. >
<mvc:view-controller .. >
<mvc:resources ..>
<mvc:default-servlet-handler ..>

<mvc:annotation-driven />

Esta etiqueta registra los beans DefaultAnnotationHandlerMapping y AnnotationMethodHandlerAdapter que son registrados por Spring MVC para despachar solicitudes web (dispatch requests) hacia los controladores anotados con @Controller

La etiqueta configura estos dos beans con parámetros por defecto basado sobre la configuración presente en nuestro classpath

<mvc:annotation-driven />

Soporte para tipos de conversion de objetos usando ConversionService, además de JavaBeans PropertyEditors durante el proceso de Data Binding (poblamiento de valores en objetos, desde y hacia formularios).

Soporte para el formato en los campos numéricos utilizando la anotación @NumberFormat

Soporte para el formato en los campos de fechas con Date, Calendar, Long, y Joda Time utilizando la anotación @DateTimeFormat.

Soporte para validar formarios en Controller con la anotación @Valid, utilizando el API de validación de Java JSR 303: Bean Validation. El soporte se detecta en el classpath y se habilita automáticamente

Soporte para lectura y escritura XML

Soporte para lectura y escritura JSON

<mvc:annotation-driven />

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd">

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

<mvc:view-controller>

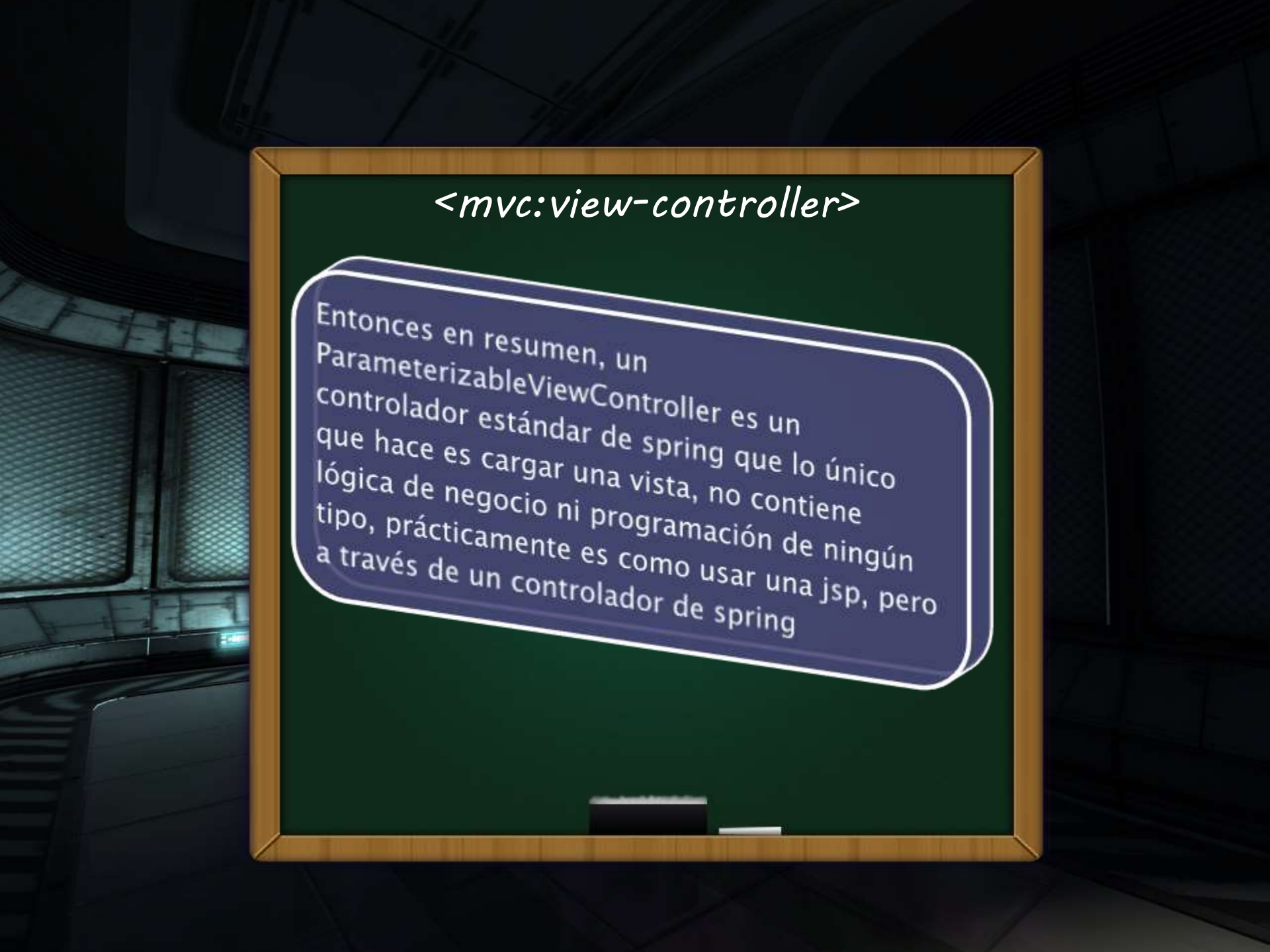
Esta etiqueta es un atajo para definir un ParameterizableViewController que inmediatamente mapea una url hacia una vista, sin necesidad de pasar por un controlador implementado

Generalmente se usa para casos con contenido estáticos, cuando no hay lógica de negocio asociada al controlador, donde p:viewName sería el nombre de la vista a cargar por el controlador de spring "ParameterizableViewController"

```
<!-- url "/bienvenidos" mapea hacia la vista "home" -->
<mvc:view-controller path="/bienvenidos" view-name="home"/>
<!-- url "/index.html" mapea hacia la vista "index" -->
<mvc:view-controller path="/index.html" view-name="index"/>
```

<mvc:view-controller>

ParameterizableViewController es un controller que viene ya creado en spring, listo para usarse, la idea es para cargar vistas jsp directamente sin la necesidad de crear una clase controller, ¿la razón? en vez de crear un controlador que no aporte nada en la lógica de negocio y sólo lo necesitamos para que cargue una vista jsp, pero no hay lógica de negocio, ni conexiones es decir nada salvo cargar la vista jsp, en esos casos para evitar crear el controller usamos directamente el ParameterizableViewController, que justamente hace nada más que crear la vista, por lo tanto se define con la url y el nombre de la vista a cargar


<mvc:view-controller>
Entonces en resumen, un ParameterizableViewController es un controlador estándar de spring que lo único que hace es cargar una vista, no contiene lógica de negocio ni programación de ningún tipo, prácticamente es como usar una jsp, pero a través de un controlador de spring

<mvc:resources>

- Proporciona una conveniente alternativa para incluir/insertar recursos estáticos desde lugares distintos a la raíz de la aplicación web (web root), incluyendo localización del classpath
- La propiedad cache-period puede ser usado para definir expiración de las cabeceras HTTP (headers)

```
<!-- location es la ruta o ubicación física donde se encuentran los archivos estáticos o de recurso (el directorio), mientras que mapping se refiere al acceso a ellos mediante ruta URL, mapeo url. Los mvc:resources se refieren a recursos estáticos, ejemplo js, css, pdf, xls, doc, imágenes, archivos html estaticos etc-->
<mvc:resources mapping="/resources/**" location="/public-resources/" cache-period="31556926"/>
```

<mvc:resources>

- Otros ejemplos

```
<!-- Peticiones HTTP GET para la url /resources/** sirve de manera eficiente para cargar los recursos estáticos en el directorio ${webappRoot}/recursos -->
<mvc:resources mapping="/resources/**" location="/recursos"/>

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

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

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

Logging

Logging

Para integrar log4j a nuestro proyecto con spring, se debe crear un archivo log4j.properties o log4j.xml en la ruta classpath del proyecto con las configuraciones

Ejemplo Logging en el controlador

```
@Controller
public class HelloWorldController {

   private static org.apache.log4j.Logger log =
Logger.getLogger(HelloWorldController.class);

   @RequestMapping(value = {"/index", "/helloWorld"})
   public ModelAndView helloWorld() {

      log.trace("Trace");
      log.debug("Debug");
      log.info("Info");
      log.warn("Warn");
      log.error("Error");
      log.fatal("Fatal");

      return new ModelAndView("helloWorld");
   }
}
```

Config Logging en archivo properties

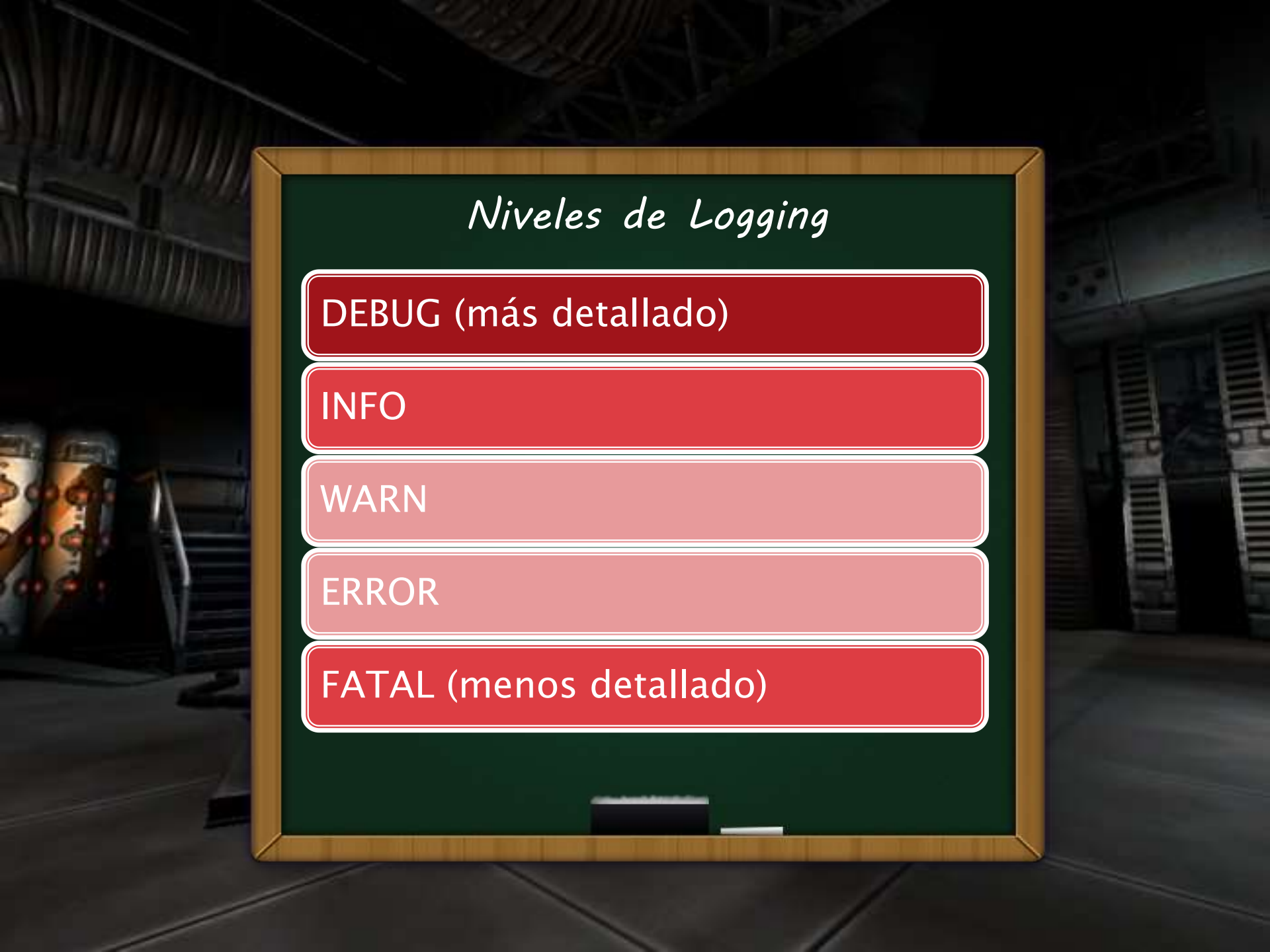
log4j.properties

```
# For JBoss: Avoid to setup Log4J outside
$JBOSS_HOME/server/default/deploy/log4j.xml!
# For all other servers: Comment out the Log4J listener in web.xml to activate
Log4J.
#log4j.rootLogger=INFO, stdout, logfile
log4j.rootLogger=ERROR, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - <%m>%n
#log4j.appender.logfile=org.apache.log4j.RollingFileAppender
#log4j.appender.logfile.File=${mvc_basics_form.root}/WEB-
INF/mvc_basics_form.log
#log4j.appender.logfile.MaxFileSize=512KB
# Keep three backup files.
#log4j.appender.logfile.MaxBackupIndex=3
# Pattern to output: date priority [category] - message
#log4j.appender.logfile.layout=org.apache.log4j.PatternLayout
#log4j.appender.logfile.layout.ConversionPattern=%d %p [%c] - %m%n
log4j.logger.org.springframework.web=DEBUG
```

Config Logging en archivo XML

log4j.xml

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
   <!-- Appenders -->
   <appender name="console" class="org.apache.log4j.ConsoleAppender">
      <param name="Target" value="System.out" />
      <layout class="org.apache.log4j.PatternLayout">
         <param name="ConversionPattern" value="%-5p: %c - %m%n" />
      </layout>
   </appender>
   <!-- 3rdparty Loggers -->
   <logger name="org.springframework.core">
      <level value="info" />
   </logger>
   <logger name="org.springframework.beans">
      <level value="info" />
   </logger>
   <logger name="org.springframework.context">
      <level value="info" />
   </logger>
   <logger name="org.springframework.web">
      <level value="debug" />
   </logger>
   <!-- Root Logger -->
   <root>
      <priority value="warn" />
      <appender-ref ref="console" />
   </root>
</log4j:configuration>
```


Niveles de Logging
DEBUG (más detallado)
INFO
WARN
ERROR
FATAL (menos detallado)

Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com