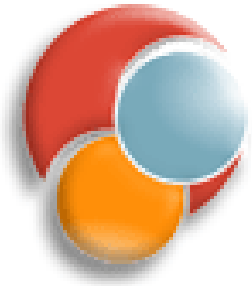


Spring

Sesión 3: Spring MVC

Indice

- **Spring MVC vs. Struts**
- Procesamiento de una petición
- Configuración básica
- Caso 1: petición sin entrada de datos
- Caso 2: petición con datos de entrada y validación

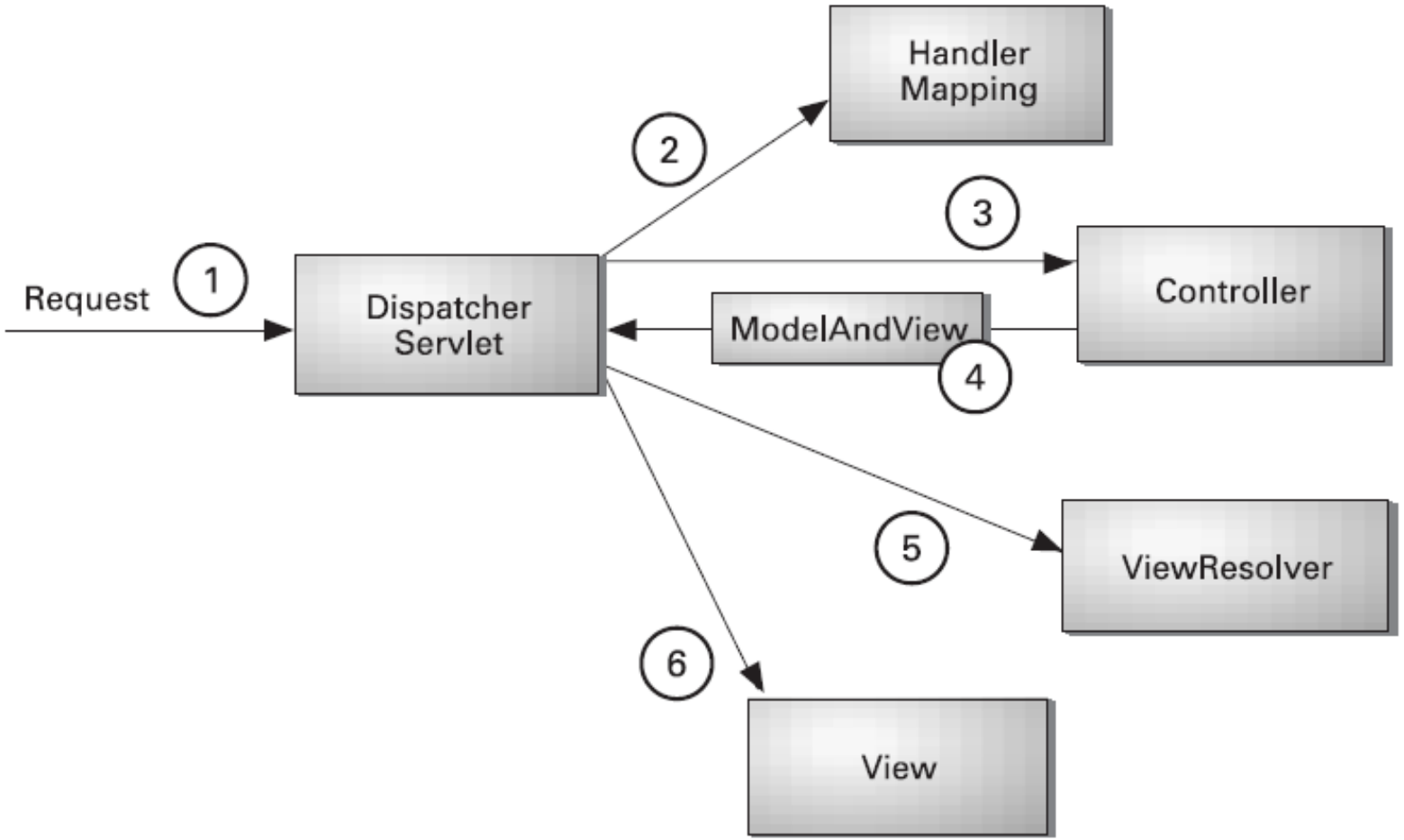
Spring MVC vs. Struts

- En general los dos frameworks ofrecen cosas similares
  - “Controllers” de Spring vs. Acciones de Struts
  - “Commands” de Spring vs. ActionForms de Struts
  - “Validators” de Spring vs. ValidatorForms de Struts
  - Taglibs de Spring vs. las de Struts
  - …
- Spring tiene una arquitectura mejor diseñada, más completa y configurable… lógico, ya que es mucho más moderno que Struts 1.X.

Indice

- Spring MVC vs. Struts
- **Procesamiento de una petición**
- Configuración básica
- Caso 1: petición sin entrada de datos
- Caso 2: petición con datos de entrada y validación

Procesamiento de una petición



Indice

- Spring MVC vs. Struts
- Procesamiento de una petición
- **Configuración básica**
- Caso 1: petición sin entrada de datos
- Caso 2: petición con datos de entrada y validación

Configuración básica

- En el web.xml

```
<servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.
                                servlet.DispatcherServlet </servlet-class>
</servlet>
<servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

- En el *nombre_del_servlet*-servlet.xml (aquí **dispatcher-servlet.xml**) definiremos los beans de la capa web.
  - Los de negocio (**applicationContext.xml**) son accesibles desde él automáticamente

Alternativas de implementación en Spring 2.5

- Controller: el equivalente al Action de Struts
  - Es decir, lo que tenemos que implementar
- 1. **Anotaciones:**
  - Nos ahorra mucho XML en el dispatcher-servlet.xml
  - Nos permite que los controller sean POJOS y sus métodos de signatura arbitraria
- 2.”Clásico”:
  - El controller debe heredar de cierta clase y sus métodos deben tener cierta signatura

Enfoque que vamos a seguir

- En Spring MVC se puede hacer lo mismo de muchas maneras distintas, usemos o no anotaciones
- Mejor que ver las cosas sistemáticamente (imposible en 2 horas), vamos a ver dos casos particulares y sin embargo muy típicos
  - **1: Petición HTTP para recuperar datos** (ej: datos de un pedido, lista de todos los clientes, todos los libros,…)
  - **2: Petición HTTP con entrada de datos y obtención de resultados** (vale, en realidad eso son 2 peticiones, pero en Spring se hace todo “en el mismo código”).

Indice

- Spring MVC vs. Struts
- Procesamiento de una petición
- Configuración básica
- **Caso 1: petición sin entrada de datos**
- Caso 2: petición con datos de entrada y validación

Caso 1: petición sin formulario

- Ver “ofertas del mes XX”

```
public class Oferta {
   private BigDecimal precio;
   private TipoHabitacion tipoHab;
   private int minNoches;
   ...
}
```

```
public enum TipoHabitacion {
        individual,
        doble
}
```

```
@Service
public class GestorOfertas {
   public List<Oferta> getOfertasDelMes(int mes) {
      ...
   }
   public List<Oferta> buscarOfertas(int precMax, TipoHabitacion t) {
      ...
   }
}
```

El Controller (≡ Action de Struts)

- Es un POJO con anotaciones
  - @Controller: indica que la clase es un controlador
  - @RequestMapping: mapeo URL -> controlador

```
@Controller
@RequestMapping("/listaOfertas.do")
public class ListaOfertasController  {
   //Accesible aunque está definido en applicationContext.xml
   @Autowired
   private GestorOfertas miGestor;
   ...
}
```

- **También** en el **dispatcher-servlet.xml h**ay que decirle a Spring que autodetecte las anotaciones

```
<context:component-scan base-package="es.ua.jtech.spring.mvc"/>
```

El trabajo del controller

Cierto método (¿cuál?) del controller deberá:

- Obtener los parámetros HTTP (si los hay)
  - Mediante anotaciones podemos asociar parámetros java a parámetros HTTP
- Disparar la lógica de negocio
- Colocar el resultado en un ámbito accesible a la vista y cederle el control
  - La clase **MapModel** permite almacenar varios objetos asignándole a cada uno un nombre, accesibles a la vista
  - Si devolvemos un String eso es el nombre lógico de la vista (como en JSF)

El trabajo del controller:

0. Asociar petición con método a ejecutar

- 2 posibilidades
  - Mapear el método con el GET a la URL

```
@Controller
@RequestMapping("/listaOfertas.do")
public class ListaOfertasController  {
   @RequestMapping(method=RequestMethod.GET)
   public String procesar(...) {
      ...
 }
}
```

  - Mapear directamente el método a la URL (GET, POST,...)

```
@Controller
public class ListaOfertasController  {
   @RequestMapping("/listaOfertas.do")
   public String procesar(...) {
      ...
 }
}
```

El trabajo del controller:

1. obtener parámetros HTTP

- **@RequestParam** asocia y convierte un parámetro HTTP a un parámetro java

```
@RequestMapping(method=RequestMethod.GET)
 public String procesar(@RequestParam("mes") int mes) {
      ...
 }
```

El trabajo del controller:

2 y 3. Disparar la lógica y colocar el resultado

- La clase **ModelMap** permite compartir objetos con la vista, asignándoles un nombre
- "Magia": cuando pasamos un parámetro java ModelMap, se asocia automáticamente con el de la aplicación
  - También pasa con otros tipos, como HttpServletRequest

```
@RequestMapping(method=RequestMethod.GET)
public String procesar(@RequestParam("mes") int mes,
                       ModelMap modelo) {
  List<Oferta> = miGestor.getOfertasDelMes(mes);
  modelo.addAttribute("ofertas");
  return "listaOfertas";
}
```

Resolver el nombre de la vista

- El String retornado es el nombre lógico, no el físico
- El encargado de mapearlo es un ViewResolver, de los que hay varias implementaciones en Spring. Uno de los más sencillos es el InternalResourceViewResolver
- En este caso, por ejemplo, "result" ⇨❾/jsp/result.jsp"

(en el dispatcher-servlet.xml)

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

La vista (JSP)

- No tiene nada especial de Spring

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core"prefix="c"%>
<html>
<head> <title>Ejemplo de vista JSP</title> </head>
<body>
      <h1>Superofertas de la semana</h1>
      <c:forEach items="${ofertas}" var="o">
            Habitación ${o.tipoHab}
                  un mínimo de ${o.minNoches} noches
                  por solo ${o.precio} &euro; la noche<br/>
      </c:forEach>
</body>
</html>
```

Indice

- Spring MVC vs. Struts
- Procesamiento de una petición
- Configuración básica
- Caso 1: petición sin entrada de datos
- **Caso 2: petición con datos de entrada y validación**

Caso 2: petición con procesamiento de datos de entrada

- Ejemplo: buscar ofertas por precio y tipo de habitación
- Esto en Struts normalmente lo haríamos con 2 acciones
  - Mostrar formulario con valores por defecto
  - Validar datos, disparar lógica de negocio
- **En Spring se suele hacer con un solo controller**
- Hay un paralelismo Spring-Struts
  - ActionForm ⇔ POJO con datos de formulario
  - Método validate() ⇔ POJO o clase que implementa Validator
  - Ambos usan ficheros .properties para los mensajes de error

Equivalente al ActionForm de Struts

- **Diferencia:** no necesita heredar de nada ni implementar nada especial

```
package es.ua.jtech.spring.mvc;
es.ua.jtech.spring.dominio.TipoHabitacion;

public class BusquedaOfertas {
	private int precioMax;
	private TipoHabitacion tipoHab;
	//..ahora vendrían los getters y setters
}
```

El trabajo del controller ahora es doble

1. Mostrar el formulario
   - Rellenarlo con datos por defecto
2. Procesar el formulario

- Validar los datos. Si no pasan la validación, volver a mostrar el formulario
- Disparar la lógica de negocio
- Colocar el resultado en un ámbito accesible a la vista y cederle el control

0. Asociar petición con método

- GET: ver formulario, POST: ya se ha rellenado, procesar

```
@Controller
@RequestMapping("/busquedaOfertas.do")
public class BusquedaOfertasController {
    @Autowired
    private GestorOfertas miGestor;

    @RequestMapping(method=RequestMethod.GET)
    public String preparaForm(...) {
        ...
    }

    @RequestMapping(method=RequestMethod.POST)
    public String procesaForm(...) {
        ...
    }
}
```

1. Mostrar el formulario

- Crear un “actionform” y colocarlo en el modelo
  - Aquí lo rellenaríamos con los valores por defecto, si queremos
- Devolver el nombre de la vista que contiene el formulario

```
@RequestMapping(method=RequestMethod.GET)
public String preparaForm(ModelMap modelo) {
    modelo.addAttribute("bo", new BusquedaOfertas());
    return "busquedaOfertas";
}
```

“1.5” La vista con el formulario

- Se usan tags de Spring

```
<%@ taglib prefix="form"
           uri="http://www.springframework.org/tags/form" %>
<html>
  <head><title>Esto es busquedaOfertas.jsp</title></head>
  <body>
    <form:form modelAttribute="bo">
      <form:input path="precioMax"/> <br/>
      <form:select path="tipoHab">
        <form:option value="individual"/>
        <form:option value="doble"/>
      </form:select>
      <input type="submit" value="buscar"/>
    </form:form>
  </body>
</html>
```

2. Procesar el formulario

- Obtener el “ActionForm”
  - **@ModelAttribute(“nombre”)** asociado a un parámetro
- Validar
  - Ver si Spring ha detectado errores: parámetro de tipo **BindingResults** que debe ser el siguiente al asociado al “ActionForm”
  - Hacer nuestra propia validación
  - Cualquiera de los dos falla: volver a la vista de formulario
- Disparar la lógica de negocio y actualizar el modelo
  - Parámetro de tipo ModelMap asociado
- Devolver el nombre de la vista deseada

2. Procesar el formulario

```
@RequestMapping(method=RequestMethod.POST)
public String procesaForm(@ModelAttribute("bo") BusquedaOfertas bo,
                          BindingResult result, ModelMap modelo) {
  //validamos los datos
  if (bo.getPrecioMax()<0)
      result.rejectValue("precioMax", "noEsPositivo");
  //si Spring o nosotros hemos detectado error, volvemos al formulario
  if (result.hasErrors()) {
      return "busquedaOfertas";
  }
  //si no, realizamos la operación
  modelo.addAttribute("ofertas", miGestor.BuscaOfertas(bo));
  //y saltamos a la vista que muestra los resultados
  return "listaOfertas";
}
```

Validación de datos

- El parámetro de tipo **BindingResult** contiene el resultado de la validación hecha por Spring (**CUIDADO: este parámetro debe seguir al parámetro asociado al "actionform"**)
  - **hasErrors()** ¿hay errores?
  - **getFieldError("precioMax")** obtener el error asociado al campo "precioMax"
  - **rejectValue("precioMax", "noEsPositivo")**: rechazar el valor del campo "precioMax". El mensaje de error se supone almacenado bajo la clave "noEsPositivo" en un fichero .properties

Los mensajes de error

- ¿Dónde está el .properties?

(en el dispatcher-servlet.xml)

```
<bean id="messageSource"
        class="org.springframework.context.
                support.ResourceBundleMessageSource">
    <property name="basename"
                value="es/ua/jtech/spring/mvc/mensajesWeb"/>
</bean>
```

- El .properties (en es.ua.jtech.spring.mvc)

```
noEsPositivo = el precio debe ser un número positivo
#Cuando Spring no puede convertir, la clave es
#typeMismatch.nombre-del-campo
typeMismatch.precioMax = el precio no es un número
```

Mostrar los mensajes de error

- Etiqueta <form:errors path=”campo”/>: muestra los errores asociados al campo

```
...
<form:form modelAttribute="bo">
      <form:input path="precioMax"/>
      <form:errors path="precioMax"/> <br/>
...
```

¿Preguntas...?