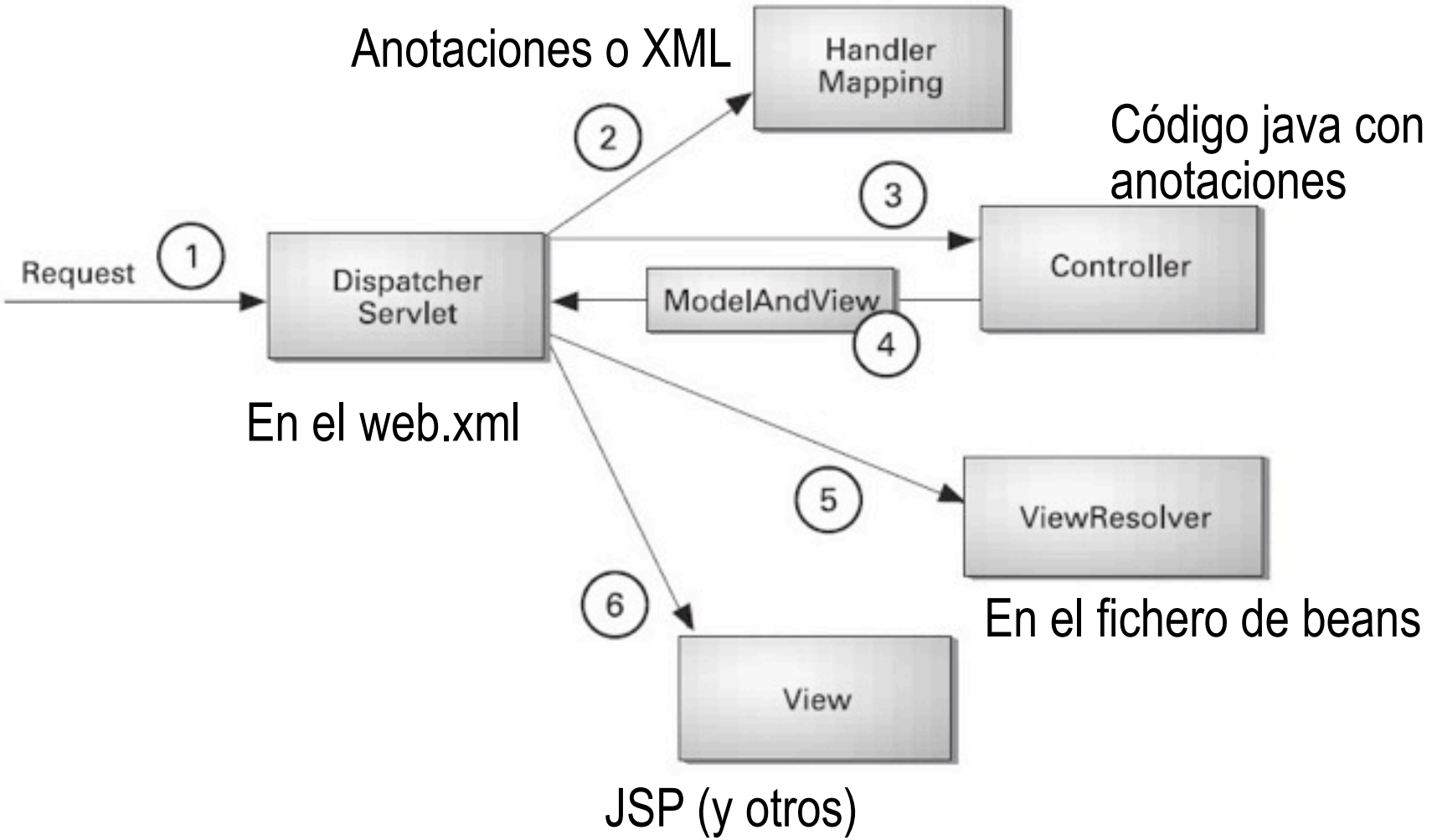
Spring

Sesión 3: Spring MVC

Indice

- 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



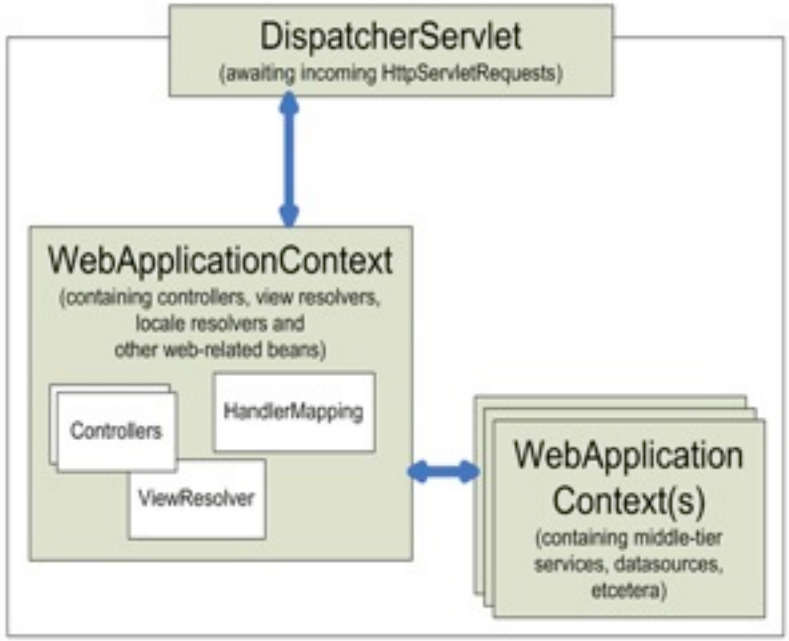
Indice

- 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.DispatcherSer
   </servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>dispatcher</servlet-name>
   <url-pattern>*.do</url-pattern>
</servlet-mapping>
```



- En el archivo *nombre_del_servlet*-servlet.xml (aquí **dispatcher-servlet.xml**) definiremos los beans de la capa web.
- Los de negocio y DAOs (**los que tengamos en otros ficheros de beans**) son accesibles desde él automáticamente

Enfoque que vamos a seguir

- Spring ofrece muchas posibilidades distintas
- 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

- 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 X”

```
public class Oferta {
    private BigDecimal precio;
    private Date fechaLimite;
    private TipoHabitacion tipoHab;
    private int minNoches;

    //..aquí vendrían los getters y setters
}
```

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

El Controller

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

```
package es.ua.jtech.spring.mvc;
import es.ua.jtech.spring.negocio.IGestorOfertas;

@Controller
@RequestMapping("/listaOfertas.do")
public class ListaOfertasController  {
    @Autowired
    private IGestorOfertas miGestor;
    ...
}
```

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

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
  - El interface **Model** 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*:

Asociar petición con método a ejecutar

- Se puede **mapear la URL con la clase** (todos los métodos responderán a la misma) o bien mapear cada método por separado a distintas URL

```
@Controller
@RequestMapping("/listaOfertas.do")
public class ListaOfertasController {
    @Autowired
    private IGestorOfertas miGestor;

    @RequestMapping(method=RequestMethod.GET)
    public String procesar(HttpServletRequest req) {
        int mes = Integer.parseInt(req.getParameter("mes"));
        ...
    }
}
```

El trabajo del *controller*:

Asociar petición con método a ejecutar (II)

- Se puede mapear la URL con la clase (todos los métodos responderán a la misma) o bien **mapear cada método por separado a distintas URL**

```
@Controller
public class OfertasController {
   @Autowired
   private IGestorOfertas miGestor;

   @RequestMapping("/listaOfertas.do”)
   public String listar(HttpServletRequest req) {
      int mes = Integer.parseInt(req.getParameter("mes"));
      ...
   }

   @RequestMapping("/getOferta.do”)
   public String getOferta(HttpServletRequest req) {
      int id = Integer.parseInt(req.getParameter("id"));
      ...
   }

}
```

El trabajo del *controller*:

1. obtener parámetros HTTP

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

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

  - Si el parámetro no existe generará un error (HTTP *status* 400)

- Para hacerlo opcional

```
public String procesar(@RequestParam(value="mes",required=false) int mes) {
    ...
}
```

El trabajo del *controller*:

2 y 3. Disparar la lógica y colocar el resultado

- El interface **Model** permite compartir objetos con la vista, asignándoles un nombre
- "Inyección implícita": cuando pasamos un parámetro de tipo Model, se asocia automáticamente con el de la aplicación

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

Más “inyección automática”

- También pasa con otros tipos, como HttpServletRequest, HttpSession, … consultar la documentación

```
@Controller
@RequestMapping("/logout")
public class LogoutController
   public String logout(HttpSession sesion) {
    sesion.invalidate();
    return "index"
   }
}
```

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” ⇨/jsp/result.jsp”

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

En el `dispatcher-servlet.xml`

La vista (JSP)

- No tiene por qué tener nada especial de Spring
  - Aunque hay etiquetas propias, útiles sobre todo para formularios

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core"prefix="c"%>
...
<c:forEach items="${ofertas}" var="o">
    Habitación ${o.tipoHab} un mínimo de
        ${o.minNoches} noches por solo ${o.precio}eur./noche
</c:forEach>
```

Indice

- 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
- Hay 2 acciones
  - Mostrar formulario con valores por defecto
  - Validar datos, disparar lógica de negocio y saltar al resultado
- Posibilidades de implementación:
  - Un controller por cada acción
  - El mismo *controller* para las dos. Mapear un método para GET y otro para POST

JavaBean para almacenar datos de formulario

- En Spring se conoce como *command*. No tiene nada de especial, es simplemente un JavaBean estándar

```
package es.ua.jtech.spring.mvc;
import 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

Previo: 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(Model modelo) {
        ...
    }

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

1. Mostrar el formulario

- Crear un “actionform” (un javabean) 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(Model modelo) {
    modelo.addAttribute("bo", new BusquedaOfertas());
    return "busquedaOfertas";
}
```

La vista con el formulario

- Por el momento usaremos HTML convencional
  - Los nombres de los campos deben coincidir con las propiedades del javabean

```
<html>
    <head><title>Esto es busquedaOfertas.jsp</title></head>
    <body>
        <form action="busquedaOfertas.do">
            <input type="text" name="precioMax"/> <br/>
            <select name="tipoHab" size=1>
                <option>individual</option>
                <option>doble</option>
            <select>
            <input type="submit" value="buscar"/>
        </form>
    </body>
</html>
```

2. Procesar el formulario

- Por el momento no validamos los datos
- Al definir un parámetro con el tipo del javabean automáticamente se rellenan las propiedades con los parámetros de la petición HTTP (los campos del formulario)

```
@RequestMapping(method=RequestMethod.POST)
public String procesaForm(BusquedaOfertas bo, Model modelo) {
   //buscamos las ofertas deseadas
   modelo.addAttribute("ofertas", miGestor.BuscaOfertas(bo));
   //y saltamos a la vista que muestra los resultados
   return "listaOfertas";
 }
```

Validación de datos

- Si hay errores de conversión de tipos, Spring los encapsulará en un objeto BindingResult
  - Podemos acceder a él sin más que definir un parámetro del tipo adecuado
  - En la siguiente sesión veremos cómo validar otro tipo de errores (p.ej. Números en un rango, etc)
- Hay que usar etiquetas de Spring en el formulario para que si falla la validación se vuelvan a mostrar los datos

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>
```

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 javabean)**
  - **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

Código para validación de datos

Cuidado: esto es validación manual, mejor usar JSR303

```
@RequestMapping(method=RequestMethod.POST)
public String procesaForm(@ModelAttribute("bo") BusquedaOfertas bo,
                          BindingResult result,
                          Model modelo) {

   //El precio no puede ser negativo
   if (bo.getPrecioMax()<0)
       result.rejectValue("precioMax", "precNoVal");
   //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";
 }
```

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" cssClass="rojo"/>
...
```

- Averiguar si hay errores

```
<%@taglib prefix="s" uri="http://www.springframework.org/tags" %>
<s:hasBindErrors name="bo">
     <form:errors path="precioMax"/>
</s:hasBindErrors>
```

Los mensajes de error

- ¿Dónde está el .properties?

```
<!-- esto estaría en el dispatcher-servlet.xml. El id del bean debe ser
     messageSource -->
<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)

```
precNoVal = precio no válido
#typeMismatch significa que no se puede convertir el valor al tipo deseado
typeMismatch.precioMax = el precio no es un número
```

¿Preguntas...?