



Formación
BDI



Curso Spring Framework

Módulo 4 Formulario y validación

Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com





Temas

Manejo de Formulario en 2 fases

Resultado: Redirect o Render

Command/form objects

Anotación @ModelAttribute

Data binding (conversiones)

Etiquetas de formulario

Validación usando anotaciones

Manejo de Formulario en 2 fases

Fase 1: Manejo inicial del formulario

- Creamos e inicializamos el objeto comando (del formulario)
- Reference data/Poblar Datos de Referencia al form (si es necesario)
- Desplegar el formulario en la página

Fase 2: Procesar envío del formulario

- Data binding del objeto de formulario (conversión)
- Validación
- Manejo de la lógica de negocio
- Resultado: Redirect o Render



Fase 1: Manejo inicial del formulario

```
@Controller
@RequestMapping(value="/cuenta")
public class CuentaController {

    // Fase 1: Manejo inicial del formulario
    @RequestMapping(method=RequestMethod.GET)
    public String crearForm(Model model) {
        // (1.1) Creamos el objeto comando Cuenta
        // El objeto cuenta mapeado al formulario
        model.addAttribute("cuenta", new Cuenta());
        // (1.3) retornamos nombre lógico de la vista "cuenta/form",
        // el cual despliega/render la vista "cuenta/form.jsp"
        return "cuenta/form";
    }

    // (1.2) Poblamos datos de referencia (opcional)
    @ModelAttribute("tipoCuentaList")
    public List<String> poblarListaTipoCuenta() {
        // Datos de referencia para elemento radiobuttons
        List<String> tipoCuenta = new ArrayList<String>();
        tipoCuenta.add("Cuenta Corriente");
        tipoCuenta.add("Cuenta Vista");
        tipoCuenta.add("Cuenta RUT");
        return tipoCuenta;
    }

    ....
}
```

Accesible en la página
formulario

Accesible en la página
formulario

Fase 1: Desplegamos el formulario en la página form.jsp

```
<form:form modelAttribute="cuenta" action="cuenta" method="post">
  <p>
    <form:label for="nombre" path="nombre">Nombre:</form:label>
    <form:input path="nombre" />
    <form:errors path="nombre" />
  </p>
  <p>
    <form:label for="saldo" path="saldo">Saldo:</form:label>
    <form:input path="saldo" />
    <form:errors path="saldo" />
  </p>
  <p>
    <form:label for="nivelEndeudamiento" path="nivelEndeudamiento" >Nivel
    Endeudamiento:</form:label>
    <form:input path="nivelEndeudamiento" />
    <form:errors path="nivelEndeudamiento" />
  </p>
  <p>
    <form:label for="fechaRenovacion" path="fechaRenovacion">Fecha
    Renovación:</form:label>
    <form:input path="fechaRenovacion" />
    <form:errors path="fechaRenovacion" />
  </p>
  <p><input type="submit" value="Crear Cuenta!" /></p>
</form:form>
```

Fase 2: Enviar formulario (submitted)

```
<form:form modelAttribute="cuenta" action="cuenta" method="post">
  <p>
    <form:label for="nombre" path="nombre">Nombre:</form:label>
    <form:input path="nombre" />
    <form:errors path="nombre" />
  </p>
  <p>
    <form:label for="saldo" path="saldo">Saldo:</form:label>
    <form:input path="saldo" />
    <form:errors path="saldo" />
  </p>
  <p>
    <form:label for="nivelEndeudamiento" path="nivelEndeudamiento" >Nivel
    Endeudamiento:</form:label>
    <form:input path="nivelEndeudamiento" />
    <form:errors path="nivelEndeudamiento" />
  </p>
  <p>
    <form:label for="fechaRenovacion" path="fechaRenovacion">Fecha
    Renovación:</form:label>
    <form:input path="fechaRenovacion" />
    <form:errors path="fechaRenovacion" />
  </p>
  <p><input type="submit" value="Crear Cuenta!" /></p>
</form:form>
```

Fase 2: Procesar envío formulario



```
@Controller
@RequestMapping(value="/cuenta")
public class CuentaController {
...
    // Fase 2: Procesar envío formulario
    // (2.1) Data binding del objeto de formulario (conversión)
    // (2.2) Validación es ejecutada antes que el método sea llamado
    @RequestMapping(method=RequestMethod.POST)
    public String crearCuenta(@Valid Cuenta cuenta, BindingResult result) {

        if (result.hasErrors()) {
            return "cuenta/form";
        }

        (2.3) // Manejamos la lógica de negocio
        this.cuentas.put(cuenta.asignarId(), cuenta);


        (2.4) Redirigimos
        return "redirect:/cuenta/" + cuenta.getId();
    }
...
}
```

El method del request es POST

El objeto es mapeado e inyectado automáticamente, con @Valid validamos

Manejamos la lógica de negocio con el objeto inyectado

Retornamos el nombre de la vista a cargar (Render) o Redirigimos a otra página



Fase 2: Procesar envío formulario

El resultado o vista lo manejamos con el retorno del nombre de la vista (conocido como el nombre simbólico o lógico), para cargar una vista (Render) o bien redirigir hacia otro handler o controlador


Ejemplo en método crearCuenta()

Observamos también el tipo de envío del request:

RequestMethod.POST

```
@RequestMapping(method=RequestMethod.POST)
public String crearCuenta(@Valid Cuenta cuenta,
                          BindingResult result) {
    if (result.hasErrors()) {
        return "cuenta/form";
    }

    this.cuentas.put(cuenta.asignarId(), cuenta);
    return "redirect:/cuenta/" + cuenta.getId();
}
```

Fase 2: Procesar envío formulario

Ahora, si necesitamos que el resultado sea un Redirect HTTP, es decir para redirigir hacia otra página

Agregamos el PREFIJO: **redirect:**

```
return "redirect:/cuenta/" + cuenta.getId();
```

Redirect Form Submission



Produce una redirección HTTP hacia el cliente (navegador), antes de que la página o vista sea desplegada

Casos de Uso

Cuando desde una vista o página de formulario se ha enviado datos POST hacia un controlador/componente

Para eliminar la posibilidad de que el usuario envíe el formulario múltiples veces por medio del "refresh" o "actualizar" del navegador

Redirect

```
return "redirect:<nombre-lógico-vista>";
```

El view resolver `UrlBasedViewResolver` reconocerá como una especial indicación de que hay que hacer un redirect HTTP

El resto del nombre de la vista será tratado como la URL de redireccionamiento

Ejemplo Redirect

```
@RequestMapping(method=RequestMethod.POST)
public String guardar(@Valid Usuario usuario, BindingResult result) {

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

    // alguna lógica de negocio para guardar al usuario ...

    return "redirect:listado.htm";
}
```

*Objeto Comando
o de Formulario*



Objeto Command/Form



El objeto de formulario (command objects), es manejado por Spring, es decir cuando se envía el formulario automáticamente se hidrata el objeto con los datos del formulario



Es poblado (sus atributos) con los datos del formulario (HTTP Request parameters), es decir se inyecta cada valor del formulario en los atributos del componente



Con la conversión de tipos por defecto definido en la clase

Command/form objects

El objeto comando (o form) mapea campos del formulario hacia atributos de la clase (bean)

- Con la conversión de tipos por defecto (String, Integer, Float, etc...)
- O bien con la conversión de tipos personalizados a través de métodos anotados con `@InitBinder` y/o configuración `HandlerAdapter`

Command/form objects

El objeto mapeado al formulario es Bi-direccional, es pasado desde el controlador al formulario, y una vez enviado los datos es pasado de vuelta al método handler del controlador, entonces:

Es mapeado y pasado al controlador, vía argumento del método handler, de forma automática y se puede acceder a sus atributos con los datos del formulario como parte de la lógica de negocio una vez enviado el formulario

Puede ser pasado desde el controlador a la vista (form backing object) y es guardado en un atributo de la clase model (model attributes) para que se tenga acceso desde la vista del formulario

Command/form objects

El objeto comando o de formulario, junto al resultado de su validación serán expuestos por defecto como atributos del objeto model de la vista (model attributes), utilizando el nombre de la clase command (sin package e iniciando en minúscula)

- Por ejemplo, "orderAddress" como model attribute del tipo "mipackage.OrderAddress"
- O bien especificando a nivel del argumento (método handler) la anotación `@ModelAttribute` para definir un nombre más personalizado.

*El objeto command es creado en la fase inicial
(RequestMethod.GET)*

```
@Controller
@RequestMapping(value="/cuenta")
public class CuentaController {

    // Fase 1: Manejo inicial del formulario
    @RequestMapping(method=RequestMethod.GET)
    public String crearForm(Model model) {
        model.addAttribute("cuenta", new Cuenta());
        return "cuenta/form";
    }

    // Fase 2: Procesar envío formulario
    @RequestMapping(method=RequestMethod.POST)
    public String crearCuenta(@Valid Cuenta cuenta, BindingResult result) {
        if (result.hasErrors()) {
            return "cuenta/form";
        }
        this.cuentas.put(cuenta.asignarId(), cuenta);
        return "redirect:/cuenta/" + cuenta.getId();
    }
}
```


El Objeto command es usado cómo Model Attribute y es mostrado al usuario en la Fase 1

```
<form:form modelAttribute="cuenta" action="cuenta" method="post">
  <p>
    <form:label for="nombre" path="nombre">Nombre:</form:label>
    <form:input path="nombre" />
    <form:errors path="nombre" />
  </p>
  <p>
    <form:label for="saldo" path="saldo">Saldo:</form:label>
    <form:input path="saldo" />
    <form:errors path="saldo" />
  </p>
  <p>
    <form:label for="nivelEndeudamiento" path="nivelEndeudamiento" >Nivel
    Endeudamiento:</form:label>
    <form:input path="nivelEndeudamiento" />
    <form:errors path="nivelEndeudamiento" />
  </p>
  <p>
    <form:label for="fechaRenovacion" path="fechaRenovacion">Fecha
    Renovación:</form:label>
    <form:input path="fechaRenovacion" />
    <form:errors path="fechaRenovacion" />
  </p>
  <p><input type="submit" value="Crear Cuenta!" /></p>
</form:form>
```


El objeto command es enviado al método handler del controlador (RequestMethod.POST) que procesa el formulario en Fase 2

```
@Controller
@RequestMapping(value="/cuenta")
public class CuentaController {

    // Fase 1: Manejo inicial del formulario
    @RequestMapping(method=RequestMethod.GET)
    public String crearForm(Model model) {
        model.addAttribute("cuenta", new Cuenta());
        return "cuenta/form";
    }

    // Fase 2: Procesar envío formulario
    @RequestMapping(method=RequestMethod.POST)
    public String crearCuenta(@Valid Cuenta cuenta, BindingResult result) {
        if (result.hasErrors()) {
            return "cuenta/form";
        }
        this.cuentas.put(cuenta.asignarId(), cuenta);
        return "redirect:/cuenta/" + cuenta.getId();
    }
}
```

Anotación @ModelAttribute



@ModelAttribute

@ModelAttribute tiene dos escenarios de uso en los controladores:

Uso 1: Anotación en argumento del método handler

- @ModelAttribute mapea un atributo del objeto model (que es un objeto comando) hacia un específico argumento/parámetro del método handler del controlador.

Uso 2: Anotación sobre métodos (reference data)

- @ModelAttribute provee datos de referencia para el objeto Model, típicamente para llenar controles del form
- Métodos anotados con @ModelAttribute se ejecutan antes que el método handler seleccionado
- @RequestMapping

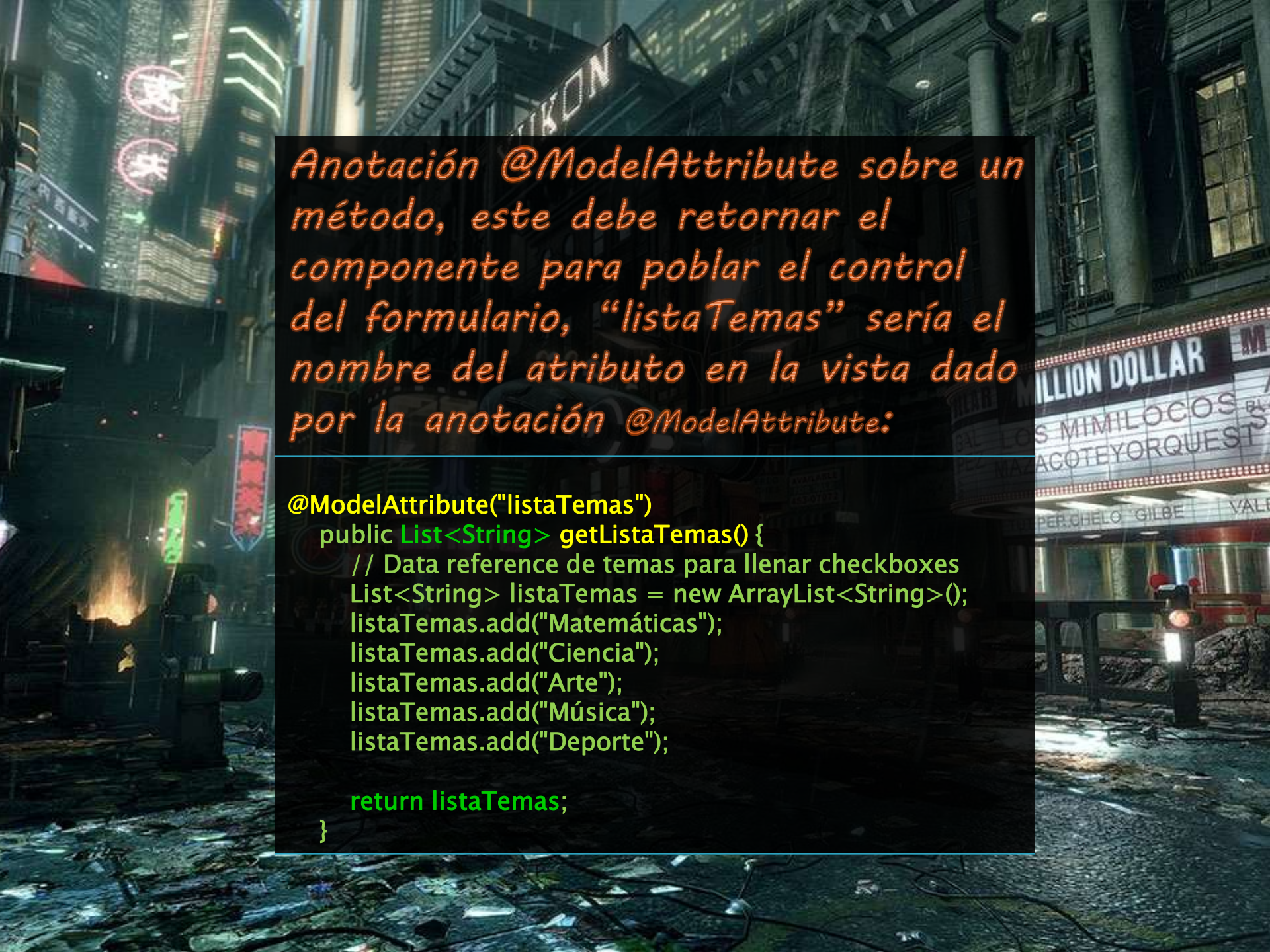
Datos de referencia con @ModelAttribute



Utilizamos la anotación `@ModelAttribute` para crear nuevos componentes que serán utilizados en controles del formulario (select, checkboxes, radiobutton etc)

Llenan/pueblan con datos a específicos atributos del objeto model view (`ModelAttribute`), típicamente desde una base de datos

Estos serán creados como componentes manejado por el contexto Spring y pasados hacia el objeto Model de la vista para que estén disponibles en la página del formulario



Anotación `@ModelAttribute` sobre un método, este debe retornar el componente para poblar el control del formulario, "listaTemas" sería el nombre del atributo en la vista dado por la anotación `@ModelAttribute`:

```
@ModelAttribute("listaTemas")
public List<String> getListaTemas() {
    // Data reference de temas para llenar checkboxes
    List<String> listaTemas = new ArrayList<String>();
    listaTemas.add("Matemáticas");
    listaTemas.add("Ciencia");
    listaTemas.add("Arte");
    listaTemas.add("Música");
    listaTemas.add("Deporte");

    return listaTemas;
}
```

@ModelAttribute

En la vista

```
<tr>
<td>Temas favoritos:</td>
<td>
<form:checkboxes items="${listaTemas}" path="temas" />
</td>
<td><form:errors path="temas" cssClass="error" /></td>
</tr>
```

Formulario Estudiante

User name:

Fecha Nacimiento:

Dirección:

Password:

Suscribirse al newsletter: ☐

Temas favoritos: ☒ Matemáticas ☐ Ciencia ☐ Arte ☐ Música ☐ Deporte

Genero: ☒ Hombre ☐ Mujer

Seleccione un número: ☐ Numero 1 ☐ Numero 2 ☐ Numero 3 ☐ Numero 4 ☐ Numero 5 ☐ Numero 6 ☐ Numero 7

País:

Habilidades de Java EE7:
Java EE JSF2
Ejb3
Hibernate
JPA



Data Binding (Conversiones)

Customizar parámetros del Request

Hay dos formas para personalizar parámetros del request usando siempre una clase PropertyEditors mediante el componente de spring WebDataBinder (Data binding)

- Forma #1: Método en el controlador anotado con `@InitBinder` (Simple y recomendada)
- Forma #2: Externalizar la configuración a otra clase/bean `WebBindingInitializer`

Forma #1: Usando @InitBinder

Anotando un método del controlador con @InitBinder, nos permite convertir valores del formulario (web data binding) en objetos o datos relacionados

@InitBinder define métodos que inicializan el WebDataBinder, el cual nos permite poblar, modificar y customizar atributos del objeto command/form con datos/objetos relacionados, antes de que el objeto command/form sea pasado por argumento en el método handler del controlador

Por ejemplo podemos asignar al objeto comando producto una categoría, o bien podríamos convertir a mayúscula el valor de un atributo string, ejemplo nombre o apellido del objeto persona

Forma #1: Usando @InitBinder

Los métodos anotados con @InitBinder soportan los mismos tipos de argumentos soportados por los métodos handler @RequestMapping, a excepción del objeto command/form y el objeto result de validación

Métodos @InitBinder NO debe tener un valor de retorno, son declarados void

Típicamente incluyen argumentos como WebDataBinder en conjunto con WebRequest o java.util.Locale, permitiendo al código registrar editores (convertidores) dentro del mismo contexto

Además nos permite registra clases validadoras de formularios, alternativa a validar con anotaciones.

Ejemplo @InitBinder

@Controller

```
public class EjemploFormController {
```

```
    // Configuramos editor/convertidor CustomDateEditor  
    // para las fechas del formulario
```

@InitBinder

```
public void initBinder(WebDataBinder binder) {
```

```
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
    dateFormat.setLenient(false);
```

```
    // Registramos la clase editor CustomDateEditor (incluida en Spring)  
    binder.registerCustomEditor(Date.class,  
                                new CustomDateEditor(dateFormat, false));
```

```
    // Registramos una clase editor personalizada  
    binder.registerCustomEditor(Categoria.class,  
                                new CategoriaEditor(this.dao));
```

```
    }  
    // ...  
}
```

Personalizada clase Binder/Editor

```
public class CategoriaEditor extends PropertyEditorSupport {  
  
    private IProductoDao dao;  
  
    public CategoriaEditor(IProductoDao dao) {  
        this.dao = dao;  
    }  
  
    @Override  
    public void setAsText(String text) throws IllegalArgumentException {  
        for (Categoria categoria : dao.getCategorias()) {  
            if (categoria.getNombre().equals(text)) {  
                setValue(categoria);  
            }  
        }  
    }  
}
```


Forma #2: WebBindingInitializer

Permite una personalizada implementación de la interface `WebBindingInitializer`

Provee un personalizada configuración (XML) del bean `AnnotationMethodHandlerAdapter`, dejando sin efecto la configuración por defecto

Forma #2: *WebBindingInitializer*

```
<bean  
class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHa  
ndlerAdapter">  
  <property name="cacheSeconds" value="0" />  
  <property name="webBindingInitializer">  
    <bean  
class="com.formacionbdi.ejemplos.catalogo.web.CatalogoBindingInitializer" />  
  </property>  
</bean>
```

Forma #2: *WebBindingInitializer*

```
public class CatalogoBindingInitializer implements WebBindingInitializer {  
  
    @Autowired  
    private IProductoDao dao;  
  
    public void initBinder(WebDataBinder binder, WebRequest request) {  
  
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
        dateFormat.setLenient(false);  
  
        // Registramos la clase editor CustomDateEditor (incluida en Spring)  
        binder.registerCustomEditor(Date.class,  
                                    new CustomDateEditor(dateFormat, false));  
  
        // Registramos una clase editor personalizada  
        binder.registerCustomEditor(Categoria.class,  
                                    new CategoriaEditor(this.dao));  
    }  
}
```



Etiquetas de formulario

Usando Etiquetas Spring Form

Spring provee un
amplio conjunto de
etiquetas para los
elementos del
formulario usando
JSP y Spring Web
MVC

Cada etiqueta
soporta el conjunto
de atributos de su
contraparte en HTML,
haciéndolas familiar
y simples de usar

Las etiquetas
generadas HTML
son compatible
con HTML
4.01 / XHTML 1.0

Como ya hemos
visto, las etiquetas
están mapeadas
hacia el objeto de
comando o
formulario

Por lo tanto se tiene
acceso a los
atributos del objeto
del formulario

Librerías de etiquetas e integración con Spring MVC

A diferencia de otras librerías de etiquetas para formulario (form/input), las etiquetas form de Spring se integra con Spring Web MVC, dando a las etiquetas acceso total al objeto comando (command object) y a los datos de referencia (reference data) provenientes del controlador

Librerías de etiquetas e integración con Spring MVC

- En las vistas utilizamos los tags de spring, para lo cual, es necesario la directiva taglib:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<!DOCTYPE html>
<html>
  <head>
    <title>Forms</title>
  </head>
  <body>
    <form:form method="post" commandName="usuario">
      <span>Nombre:</span>
      <div><form:input path="nombre" /></div>
      ... etc...
    </table>
  </form:form>
</body>
</html>
```


Etiqueta form de Spring

Supongamos que tenemos un objeto de dominio (o negocio) llamado "command". Este es un JavaBean con atributos tales como nombre y apellido y métodos getters/setters.

Por defecto si no se define en la etiqueta form el nombre del objeto comando, spring asume por defecto que el nombre será 'command'.

```
<form:form>
  <table>
    <tr>
      <td>Nombres:</td>
      <td><form:input path="nombre" /></td>
    </tr>
    <tr>
      <td>Apellidos:</td>
      <td><form:input path="apellido" /></td>
    </tr>
    <tr>
      <td colspan="2">
        <input type="submit" value="Guardar Cambios" />
      </td>
    </tr>
  </table>
</form:form>
```


Etiqueta form de Spring

Cómo ya hemos visto el objeto command o de formulario es mapeado al formulario de spring, éste es pasado a la vista desde el controlador cuando retornamos la vista (form.jsp), cada propiedad del objeto command se mapea a cada elemento del formulario mediante el atributo "path".

Etiqueta form de Spring

Sin embargo, si en el objeto model hemos guardado al objeto command con otro nombre (definitivamente mejor práctica), podemos usar el atributo `commandName` o `modelAttribute` (etiqueta form) para asignar el nombre del objeto command en el form

```
<form:form commandName="usuario">
  <table>
    <tr>
      <td>Nombres:</td>
      <td><form:input path="nombre" /></td>
    </tr>
    <tr>
      <td>Apellidos:</td>
      <td><form:input path="apellido" /></td>
    </tr>
    <tr>
      <td colspan="2">
        <input type="submit" value="Guardar Cambios" />
      </td>
    </tr>
  </table>
</form:form>
```

Etiqueta "input"

Supongamos que tenemos un objeto de formulario, un command llamado "usuario" con propiedades tales como nombre y apellido.

- Esta etiqueta dibuja una etiqueta HTML 'input' con type 'text'

```
<form:form commandName="usuario">
  <table>
    <tr>
      <td>Nombres:</td>
      <td><form:input path="nombre" /></td>
    </tr>
    <tr>
      <td>Apellidos:</td>
      <td><form:input path="apellido" /></td>
    </tr>
    <tr>
      <td colspan="2"><input type="submit" value="Guardar " /></td>
    </tr>
  </table>
</form:form>
```

Etiqueta "password"

Esta etiqueta dibuja una etiqueta HTML 'input' con type 'password'

```
<form:password path="password" />
```


Etiqueta "select"

Esta etiqueta
dibuja un
elemento select

Soporta tipos de
array, Map y
java.util.Collection

```
<form:select path="pais" items="${listaPaises}"/>
```

Etiqueta “checkboxes”

Esta etiqueta dibuja multiples etiquetas HTML 'input' con type 'checkbox'

Soporta tipos de array y java.util.Collection

```
<form:checkboxes path="temas" items="${listaTemas}"/>
```

Etiqueta "radiobuttons"

Esta etiqueta
dibuja multiples
etiquetas HTML
'input' con type
'radio'



Soporta tipos de
array y
java.util.Collection

```
<form:radiobuttons path="numeroFavorito" items="${listaNumeros}"/>
```



Validación de formulario

Validación en Spring

Desde la versión 3 de Spring, Spring MVC tiene la habilidad de validar automáticamente los campos del formulario (objeto comando o de formulario) en los `@Controllers`

En versiones anteriores era responsabilidad del desarrollador invocar manualmente la lógica de validación

Actualmente, para gatillar la validación en el @Controller, simplemente anotamos al argumento objeto comando con @Valid (from JSR 303)

```
@Controller  
public class CatalogoController {  
  
    @RequestMapping(method=RequestMethod.POST)  
    public void processForm (@Valid Producto producto,  
                             BindingResult result)  
    {  
  
    }  
}
```

Configurando un Validador

Hay dos opciones de validar cuando existe un argumento del método handler anotado con `@Valid`:

> Opción #1 – Invocar `binder.setValidator(Validator)` dentro del método anotado con `@InitBinder` en el `@Controller`. Permite configurar y registrar un validador en el controlador

> Opción #2 – Validación usando anotaciones en la clase Comando (JSR-303 Validator)

Opción #1 - Validación vía @Controller

@Controller

public class CatalogoController {

@InitBinder

```
protected void initBinder(WebDataBinder binder) {  
    binder.setValidator(new ProductoValidator());  
}
```

@RequestMapping("/guardar", method=RequestMethod.POST)

public void processForm(@Valid Producto producto) { ... }

}

Opción #1 - Validación vía @Controller

```
import org.springframework.util.StringUtils;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

public class ProductoValidator implements Validator {

    @Override
    public boolean supports(Class clazz) {
        return Producto.class.equals(clazz);
    }

    @Override
    public void validate(Object obj, Errors errors) {
        Producto prod = (Producto) obj;
        if (!StringUtils.hasLength(prod.getNombre())) {
            errors.rejectValue("nombre", "required", "campo nombre requerido");
        }

        if (null == prod.getPrecio()) {
            errors.rejectValue("precio", "required", "precio no puede ser vacio");
        }
    }
}
```


Opción #2 -Validación Bean JSR-303

Spring proporciona soporte para la validación de formularios mediante la especificación JSR 303: Bean Validation de la Plataforma Java

Validación



La especificación JSR-303 es una instancia de `javax.validation.Validator`, se encarga de validar todos los objetos del modelo que declaran restricciones mediante uso de anotaciones.

Opción #2 -Validación Bean JSR-303

Para configurar el API Bean Validator JSR-303 con Spring MVC

- Simplemente agregamos el proveedor JSR-303 en nuestro classpath, por ejemplo Hibernate Validator. Spring MVC detectará y habilitará automáticamente el soporte para JSR-303 en todos nuestros @Controllers.
- Además de tener configurado
`<mvc:annotation-driven/>`

Opción #2 -Validación Bean JSR-303

El componente `javax.validation.Validator` (JSR-303) se encarga de validar a todos los objetos comando que tienen definidas las reglas de validación vía anotaciones JSR-303

Cualquier violación de las reglas de validación automáticamente gatillará en errores en el `BindingResult` mostrando los mensajes de error mediante la etiqueta form “errors” (Spring MVC form taglib):

```
<form:errors path="nombre" cssClass="error"/>
```

Clase de comando con anotaciones de validación

```
public class Cuenta {  
  
    private Long id;  
  
    @NotEmpty  
    @Size(min = 4, max = 12)  
    private String nombre;  
  
    @NotNull()  
    private BigDecimal saldo = new BigDecimal("5500");  
  
    @NotNull  
    @DecimalMax(value = "1.00")  
    @Digits(fraction = 2, integer = 1)  
    private BigDecimal nivelEndeudamiento = new BigDecimal(".05");  
  
    @Email  
    private String email;  
  
    @DateTimeFormat(style="S-")  
    @Future  
    private Date fechaRenovacion = new Date(new Date().getTime() + 31536000000L);  
}
```


Clase @Controller

```
@Controller
@RequestMapping(value = "/cuenta")
public class CuentaController {

    private Map<Long, Cuenta> cuentas = new ConcurrentHashMap<Long, Cuenta>();

    // Metodo handler formulario, para crear la cuenta
    @RequestMapping(method = RequestMethod.GET)
    public String crearCuentaForm(Model model) {
        model.addAttribute("cuenta", new Cuenta());
        return "cuenta/crearForm";
    }

    // Metodo handler que procesa el envio de datos del form
    @RequestMapping(method = RequestMethod.POST)
    public String crearCuenta(@Valid Cuenta cuenta, BindingResult result) {
        if (result.hasErrors()) {
            return "cuenta/crearForm";
        }
        this.cuentas.put(cuenta.asignarId(), cuenta);
        return "redirect:/cuenta/" + cuenta.getId();
    }
}
```

Configuración y dependencia Maven

Para configurar JSR-303-backed Validator

- Simplemente agregamos las dependencia Maven JSR-303 Provider, tales Hibernate Validator. El proyecto detectará esta y las habilitará de forma automática

```
<!-- Bean Validation (JSR-303) -->
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
  <version>1.1.0.Final</version>
</dependency>
<!-- Hibernate Bean Validation (JSR-303) -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>5.2.1.Final</version>
</dependency>
```



GRACIAS!

Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com