Spring MVC y WebFlow

INDICE

1. INTRODUCCIÓN FRAMEWORKS WEB

Con el término framework, nos estamos refiriendo a una estructura software compuesta de componentes personalizables e intercambiables para el desarrollo de una aplicación. En otras palabras, un framework se puede considerar como una aplicación genérica incompleta y configurable a la que podemos añadirle las últimas piezas para construir una aplicación concreta.

Los objetivos principales que persigue un framework son:

- Acelerar el proceso de desarrollo
- Reutilizar código ya existente
- Promover buenas prácticas de desarrollo como el uso de patrones.

Un framework Web, por tanto, podemos definirlo como un conjunto de componentes (por ejemplo clases en java y descriptores y archivos de configuración en XML) que componen un diseño reutilizable que facilita y agiliza el desarrollo de sistemas Web.

PATRON MVC

El patrón Modelo-Vista-Controlador es una guía para el diseño de arquitecturas de aplicaciones que ofrezcan una fuerte interactividad con usuarios. Este patrón organiza la aplicación en tres modelos separados, el primero es un modelo que representa los datos de la aplicación y sus reglas de negocio, el segundo es un conjunto de vistas que representa los formularios de entrada y salida de información, el tercero es un conjunto de controladores que procesa las peticiones de los usuarios y controla el flujo de ejecución del sistema.

La mayoría, por no decir todos, de los frameworks para Web implementan este patrón.

FRAMEWORKS WEB MAS CONOCIDOS

- Cocoon
- Java Server Faces
- JetSpeed
- Maverick
- Struts
- Tapestry
- Wicket
- Spring MVC

2. SPRING MVC

Spring Mvc es una alternativa de framework basado en el patrón modelo-vista-controlador. Es un modulo más dentro del framework Spring.

A partir de la versión permite trabajar con anotaciones que suplen al código xml.

CONFIGURACION DEL FRAMEWORK

Como todos los frameworks web necesitamos mapear en el web.xml el servlet dispatcher que nos proporciona Spring

```
<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>/</url-pattern>
</servlet-mapping>
```

Gráfico 1. Configuración del servlet DispatcherServlet

Una vez que el contenedor reconoce el servlet a partir de aquí puede cargar el fichero xml donde irán mapeadas todas las peticiones.

Toda la configuración de la capa web irá en este fichero.

El nombre de este archivo se resuelve de la siguiente forma, se toma el nombre del servlet <servlet-name> y se le añade -servlet.xml tal como se muestra en la imagen siguiente.

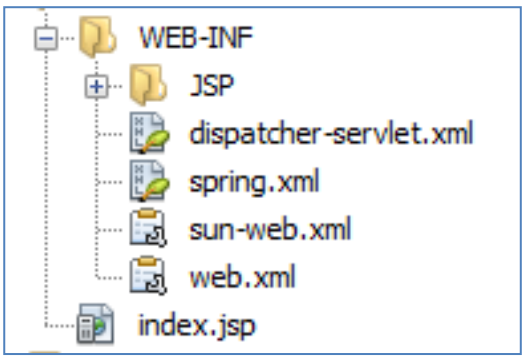

Gráfico 2. Archivos de configuración

También necesitaremos el archivo de configuración de spring para poder declarar beans como por ejemplo un DataSource, el Dao, ...etc.

En nuestro ejemplo utilizamos Hibernate por lo cual aquí declaramos el bean de tipo SessionFactory.

Se precisa de un listener de tipo ContextLoaderListener y de un parametro de contexto con la ruta del archivo spring.xml tal como vemos en la imagen siguiente:

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring.xml</param-value>
</context-param>
```

Gráfico 3. Configuración del archivo spring.xml

CONTROLLERS

La lógica de negocio asociada a cada petición la resolverá un controlador asociado a esta. Estos controladores se denominan Controllers.

Existen dos formas de manejar las peticiones:

- Configurando el path y el controller a través de xml
- Configurando el path y el controller a través de anotaciones

PETICIONES SOBRE XML

Vamos a realizar como ejemplo la gestión de una tienda de productos.

Nuestra primera petición consistirá en solicitar todos los productos. Para ello debemos crear un link desde la página index.jsp

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <a href="todos">Ver todos los productos</a><br>
    </body>
</html>
```

Gráfico 4. Petición para ver todos los productos

A continuación creamos el controller que procesará esta petición.

```
public class VerTodosController extends AbstractController{

    private ProductoDao dao;

    @Override
    public ModelAndView handleRequestInternal(HttpServletRequest request,
                                              HttpServletResponse response)
                                              throws Exception {

        // Establecer la vista donde mostrara todos los productos
        ModelAndView vista = new ModelAndView("mostrarTodos");

        // guardar la lista de productos como atributo
        vista.addObject("lista", dao.listarProductos());

        return vista;
    }
```

Gráfico 5. Controller que atiende la petición mostrar todos los productos.

Por último mapeamos la petición en el archivo dispatcher-servlet.xml

```
<bean name="/todos" class="app.controlers.VerTodosController">
    <property name="dao" ref="beanDao" />
</bean>
```

Gráfico 6. Mapeo de la petición mostrar todos

El código de este ejemplo lo encontrareis en **Ejemplo1_Spring_MVC.zip**

PETICIONES SOBRE ANOTACIONES

Podemos utilizar anotaciones para mapear el path con su controller correspondiente. Ahora sobre el propio controller utilizamos la anotación RequestMapping para establecer el path de la aplicación.

Sobre el método verTodos indicamos que se ejecutará para las peticiones por el método GET.

Podemos comprobar que el controller no hereda de la clase AbstractController como ocurría anteriormente.

```
@RequestMapping("/todos")
public class VerTodosController {

    private ProductoDao dao;

    @RequestMapping(method = RequestMethod.GET)
    public String verTodos(Model modelo) {
        modelo.addAttribute("lista",  dao.listarProductos());
        return "mostrarTodos";
    }
```

Grafico 7. Controller con anotaciones

Debemos añadir el siguiente elemento para que pueda procesar las anotaciones. Este se declara en el dispatcher-servlet.xml

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

Grafico 8. Configuración con anotaciones

Por último solo declaramos el bean del controller en el dispatcher-servlet.xml

```
<bean class="app.controlers.VerTodosController">
    <property name="dao" ref="beanDao" />
</bean>
```

Grafico 9. Declaración del bean VerTodosController

El código de este ejemplo lo encontrareis en **Ejemplo2_Spring_MVC_Anotaciones.zip**

SOLUCIONADOR DE VISTAS

En nuestro proyecto todas las paginas que actuarán de vistas estarán ubicadas dentro de una carpeta JSP que definiremos dentro de la carpeta WEB-INF.

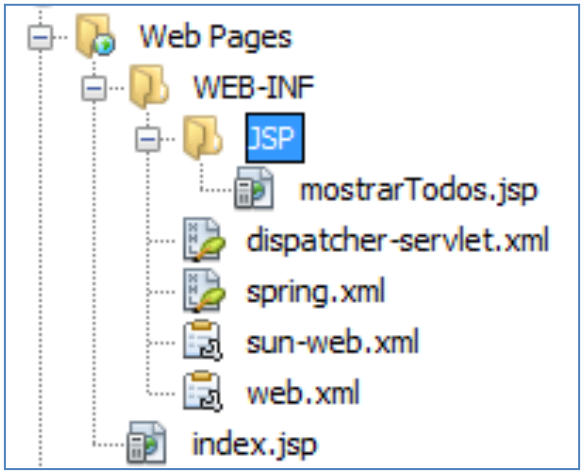

Gráfico 10. Ubicación de las vistas

Como hemos visto en cualquiera de los dos controllers anteriores no utilizamos la ruta ni la extensión de la vista que mostrará la respuesta de la petición, en nuestro ejemplo la tabla con todos los productos.

Vamos a utilizar un solucionador de vistas para que automáticamente se establezca un prefijo y un sufijo. De esta forma el framework encontrará la pagina.

En la imagen siguiente vemos como se declara este bean en el dispatcher-servlet.xml

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

Gráfico 11. Declaración del bean solucionador de vistas

UN CONTROLLER PARA VARIAS PETICIONES

Podemos utilizar el mismo controller para atender varias peticiones en este caso en vez de poner el path a nivel de clase lo pondremos a nivel de método.

```
@RequestMapping
public class BuscarController {

    private ProductoDao dao;

    @RequestMapping(method = RequestMethod.GET, value = "/uno")
    public String buscarProductoPorId(@RequestParam("id") int codigo,
            Model modelo) {
        Producto encontrado = dao.buscarPorId(codigo);
        modelo.addAttribute("otro", encontrado);
        return "mostrarUno";
    }

    @RequestMapping(method = RequestMethod.GET, value = "/buscarNombre")
    public String buscarProductoPorNombre(@RequestParam("nombre") String nombre,
            Model modelo) {
        Producto encontrado = dao.buscarPorNombre(nombre);
        modelo.addAttribute("otro", encontrado);
        return "mostrarUno";
    }
```

Gráfico 12. Crear un controller para varias peticiones

Una vez creado el controller el siguiente paso es declarar un bean de él en el dispatcher-servlet.xml

```
<bean class="app.controlers.BuscarController">
    <property name="dao" ref="beanDao" />
</bean>
```

Gráfico 13. Declaración del bean BuscarController

Ahora ya podremos emitir peticiones contra este controller.

```
<a href="uno?id=3">Buscar el producto con id 3</a><br>

<a href="buscarNombre?nombre=Teclado">Buscar teclado</a><br>
```

Gráfico 14. Peticiones en index.jsp

El código de este ejemplo lo encontrareis en **Ejemplo3_Spring_MVC_Controller.zip**

En este ejemplo quizás os resulte incoherente el hecho de buscar siempre los mismos productos. La solución es utilizar formularios de entrada de datos para que el usuario pueda introducir el id o el nombre del producto a buscar.

FORMULARIOS DE ENTRADA DE DATOS

Vamos a crear un controller que permite capturar dos peticiones:

- La primera de ellas será por el método GET y su misión es facilitar una instancia de producto al formulario para que esta sea rellenada con los datos introducidos por el usuario.
- La segunda petición será por el método POST y será la encargada de dar de alta el producto.

```java
@RequestMapping("/alta")
public class AltaController {

    private ProductoDao dao;

    @RequestMapping(method = RequestMethod.GET)
    public String enviarFormulario(Model modelo) {
        modelo.addAttribute("producto", new Producto());
        return "formularioAlta";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String procesarDatos(@Valid Producto producto,
            BindingResult resultadoValidacion) {

        if (resultadoValidacion.hasErrors()) {
            return "formularioAlta";
        } else {
            dao.alta(producto);
            return "confirmacion";
        }
    }
```

Gráfico 15. Creación del controller para el alta de productos

Una vez creado el controller se debe declarar un bean que lo instancie en el dispatcher-servlet.xml

```xml
<bean class="app.controlers.AltaController">
    <property name="dao" ref="beanDao" />
</bean>
```

Gráfico 16. Declaración del bean AltaController

A continuación vemos el formulario que se facilitará al usuario. Spring MVC proporciona una serie de etiquetas que hacen más fácil esta labor.

Como vemos se declara el formulario con su action, method y modelAttribute. Este último es el encargado de volver a enviar al controller la instancia de producto, esta vez con las propiedades establecidas con los datos introducidos por el usuario.

```
<sf:form action="alta" method="POST" modelAttribute="producto">
    <sf:label path="id">Introduce ID:</sf:label>
    <sf:input path="id" />

    <sf:label path="nombre">Introduce nombre:</sf:label>
    <sf:input path="nombre" />

    <sf:label path="precio">Introduce precio:</sf:label>
    <sf:input path="precio" />

    <input type="submit" value="ALTA" />
</sf:form>
```

Gráfico 17. Formulario de entrada de datos

Si el alta se ha ejecutado correctamente, el usuario verá la pagina de confirmación.

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>El producto se ha dado de alta correctamente</h1>
    </body>
</html>
```

Gráfico 18. Pagina de confirmación de alta

El código de este ejemplo lo encontrareis en **Ejemplo4_Spring_MVC_Formularios.zip**

VALIDACION DE DATOS

Las validaciones en Spring MVC se llevan a cabo a través de la especificación JSR-303 que incluye reglas de validación con anotaciones.

Estas anotaciones las incluimos sobre las propiedades de nuestro modelo, en nuestro ejemplo será la clase Producto.

```java
@NotNull
@Digits(integer=4, fraction=0, message="Debe tener 4 digitos como maximo")
private int id;

@Size(min=3, max=15, message="Debe tener entre 3 y 15 caracteres")
private String nombre;

@Digits(integer=8, fraction=2, message="Debe tener 10 digitos como maximo")
@DecimalMin(value="1", message="El precio minimo ha de ser 1")
private double precio;
```

Gráfico 19. Validaciones en la clase Producto

En el formulario de entrada agregamos una nueva etiqueta por cada campo para mostrar el mensaje en caso de que no se cumpla la validación.

```jsp
<sf:form action="alta" method="POST" modelAttribute="producto">
    <sf:label path="id">Introduce ID:</sf:label>
    <sf:input path="id" />
    <sf:errors path="id" /><br>

    <sf:label path="nombre">Introduce nombre:</sf:label>
    <sf:input path="nombre" />
    <sf:errors path="nombre" /><br>

    <sf:label path="precio">Introduce precio:</sf:label>
    <sf:input path="precio" />
    <sf:errors path="precio" /><br>

    <input type="submit" value="ALTA" />
</sf:form>
```

Gráfico 20. Formulario de entrad de datos

A continuación listamos las anotaciones con las principales reglas de validación:

@AssertFalse: El campo booleano tiene que ser false.

@AssertTrue: El campo booleano tiene que ser true.

@DecimalMax: El campo tiene que ser un numero cuyo valor sea menor o igual a un máximo especificado.

@DecimalMin: El campo tiene que ser un numero cuyo valor sea mayor o igual a un mínimo especificado.

@Digits: El campo tiene que ser un número cuyo valor se encuentre en un rango definido.

@Future: El campo tiene que ser una fecha futura.

@Max: El campo tiene que ser un numero cuyo valor sea menor o igual a un máximo especificado.

@Min: El campo tiene que ser un numero cuyo valor sea mayor o igual a un mínimo especificado.

@NotNull: El campo no puede tener valor null.

@Null: El campo debe tener valor null.

@Past: El campo debe tener una fecha ya pasada.

@Pattern: La cadena tiene que coincidir con el patrón de expresión regular especificado.

@Size: El tamaño del campo tiene que estar dentro de un límite especificado inclusive.

El código de este ejemplo lo encontrareis en **Ejemplo5_Spring_MVC_Validaciones.zip**

INTERNACIONALIZACION

La internacionalización de tu aplicación consiste en hacerla accesible no solo para las personas localizadas en un país(con un determinado lenguaje, como el español para nosotros), sino tambien para aquellas personas en otros países(como Inglaterra, Francia ó Italia, que leen inglés, francés, italiano) La internacionalización consiste básicamente en que el texto que muestra tu aplicación en un determinado lenguaje pueda ser visto en otro lenguaje si cambiar nada de código, es decir si desde Italia accedo a tu aplicación entonces todo el texto tiene que estar en italiano, si desde Inglaterra accedo a tu aplicación entonces todo el texto tiene que estar en inglés.

Para internacionalizar una aplicación debemos extraer todo el texto estático de nuestras páginas y llevarlos a un archivo de propiedades.

Estos archivos se duplican en función de los distintos idiomas que queremos ofrecer a nuestros usuarios.

Estos archivos se deben situar en el context-path de la aplicación:

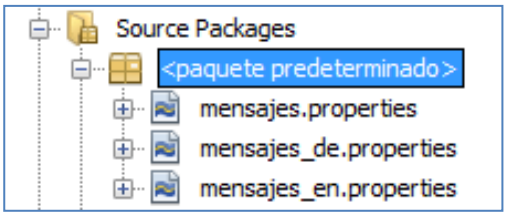

Gráfico 21. Ubicación de archivos properties

```
id_Txt=Introduce ID:
id_Nombre=Introduce Nombre:
id_Precio=Introduce Precio:
boton Alta=ALTA
```

Gráfico 22. Mensajes en español mensajes.properties

```
id_Txt=Enter ID:
id_Nombre=Enter Name:
id_Precio=Enter Price:
boton_Alta=INSERT
```

Gráfico 23. Mensajes en ingles mensajes_en.properties

```
id_Txt=Geben ID:
id_Nombre=Geben Sie Name:
id_Precio=Geben Preis:
boton_Alta=HOCH
```

Gráfico 24. Mensajes en alemán mensajes_de.properties

Cuando la aplicación sea invocada esta va a detectar el lenguaje del navegador y va a utilizar el archivo de propiedades apropiado.

En Spring MVC debemos declarar un bean en dispatcher-servlet.xml para que pueda resolver la internacionalización. En este bean debemos especificar como propiedad el nombre de nuestro archivo properties.

```
<!-- Bean para los mensajes internacionalizados  -->
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename">
        <value>mensajes</value>
    </property>
</bean>
```

Gráfico 25. Declaración del bean para resolver internacionalización

Hemos modificado nuestro formulario de alta para que se muestre internacionalizado.

```
<sf:form action="alta" method="POST" modelAttribute="producto">
    <spring:message code="id_Txt"/>
    <sf:input path="id" />
    <sf:errors path="id" /><br>

    <spring:message code="id_Nombre"/>
    <sf:input path="nombre" />
    <sf:errors path="nombre" /><br>

    <spring:message code="id_Precio"/>
    <sf:input path="precio" />
    <sf:errors path="precio" /><br>

    <input type="submit" value='<spring:message code="boton_Alta"/>'>
</sf:form>
```

Gráfico 26. Formulario con etiquetas internacionalizadas

El código de este ejemplo lo encontrareis en **Ejemplo6_Spring_MVC_Internacionalizacion.zip**

3. SPRING WEBFLOW

Spring Web Flow es un módulo del frameworkSpring dirigido a definir y gestionar los flujos de páginas dentro de una aplicación web.

El flujo de páginas de una aplicación web consiste en la secuencia de páginas por las que pasa dicha aplicación en función de la conversación que mantenga con el usuario. Dependiendo de las opciones que escoja el usuario, resultados de las operaciones de proceso…etc, la aplicación seguirá una ruta de páginas especifica u otra.

Los aspectos fundamentales que potencia Spring Web Flow son:

- Encapsulación de la lógica de los flujos de páginas como un módulo autónomo que puede reutilizarse en diferentes situaciones. Hay aplicaciones web específicas que suelen utilizar la misma lógica de flujo de páginas, como por ejemplo, aplicaciones tipo “carrito de la compra”, correo web, etc.
- Proporcionar un motor capaz de capturar los flujos de páginas de una aplicación, integrándolo con algunos frameworks de uso habitual como Spring, Spring MVC o JSF.

TIPOS DE ESTADOS

Para implementar el flujo de páginas con Spring Web Flow creamos un fichero xml dónde se incluirá la declaración del flujo de páginas de la aplicación en forma de estados.

Se necesitarán los siguientes tipos de estados:

View-State: Son estados en los que se activa algún recurso con tareas de presentación en la aplicación. Por ejemplo, en aplicaciones web corresponde habitualmente con la activación de Java Server Pages (jsp´s).

Action-State: Son estados en los que la aplicación realiza alguna labor de la lógica de negocio. Por ejemplo, acciones típicas sobre bases de datos, proceso de datos, etc. La transición de este tipo de estados a otros suele estar condicionada por el resultado de la operación específica que se realice en el mismo. Un poco más adelante se comentará como gestiona esto Spring Web Flow.

Start-State: Es un estado en el que se indica que comienza el flujo de la aplicación. Al igual que los View-State, suele llevar asociada la activación de un recurso de presentación (jsp, etc).

End-State: Son estados en los que el flujo de la aplicación finaliza y ya no avanza hacia ningún otro. Al igual que los View-State, suele llevar asociada la activación de un recurso de presentación (jsp, etc).

EJEMPLO

Vamos a crear un ejemplo muy simple en el cual tan solo con pulsar un link en la página index.jsp accederemos al flujo que nos llevará a la página inicio.jsp, luego a medio,jsp y finalmente a final.jsp.

Utilizamos este ejemplo para explicar todos los pasos a seguir en el desarrollo de flujos.

DECLARAR EL FLUJO

En el archivo principal-flow.xml declaramos el flujo tal como vemos en la imagen.

```
<view-state id="entrada" view="inicio">
      <transition on="ok1" to="intermedio" />
</view-state>

<view-state id="intermedio" view="medio">
      <transition on="ok2" to="fin" />
</view-state>

<end-state id="fin" view="final" />
```

Gráfico 27. Declarar el flujo

Como nuestro flujo se basa únicamente en mostrar páginas web por eso utilizamos estados de tipo <view-state> y un estado final <end-state>.

Con el elemento <transition> en función del evento recogido en la página pasaremos al siguiente estado.

CONFIGURAR EL FLUJO

Los siguientes elementos deben ser declarados como beans en el dispatcher-servlet.xml

viewResolver

Es el solucionador de vistas igual que lo vimos en Spring MVC

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

Gráfico 28. Declaración del bean viewResolver

viewFactoryCreator

Crea factorías de vista apoyándose en Spring MVC

```
<bean id="viewFactoryCreator"
      class="org.springframework.webflow.mvc.builder.MvcViewFactoryCreator">
    <property name="viewResolvers">
        <list>
            <ref bean="viewResolver" />
        </list>
    </property>
</bean>
```

Gráfico 29. Declaración del bean viewFactoryCreator

flowBuilderServices

Contenedor para los servicios necesarios durante la construcción del flujo, asi como la creación de factorias de vista del ViewFactoryCreator

```
<webflow:flow-builder-services id="flowBuilderServices"
                               view-factory-creator="viewFactoryCreator"/>
```

Gráfico 30. Declaración del bean flowBuilderServices

flowRegistry

El registro de flujos se encarga de buscar los flujos .xml en la ruta especificada

```
<webflow:flow-registry id="flowRegistry"
                       flow-builder-services="flowBuilderServices">
    <webflow:flow-location id="inicio"
                           path="/WEB-INF/flujos/principal-flow.xml" />
</webflow:flow-registry>
```

Gráfico 31. Declaración del bean flowRegistry

flowExecutor

El ejecutor de flujos se encarga de crear y ejecutar una instancia de flujo para cada usuario

```
<webflow:flow-executor id="flowExecutor" flow-registry="flowRegistry" />
```

Gráfico 32. Declaración del bean flowExecutor

FlowHandlerMapping

FlowHandlerMapping dirige las peticiones desde el DispatcherServlet al flujo

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
    <property name="flowRegistry" ref="flowRegistry" />
</bean>
```

Gráfico 33. Declaración del bean FlowHandlerMapping

FlowHandlerAdapter

FlowHandlerAdapter responde a las peticiones

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
    <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

Gráfico 34. Declaración del bean FlowHandlerAdapter

FlowController

FlowController es el controller para manejar flujos

```
<bean id="flowController"
      class="org.springframework.webflow.mvc.servlet.FlowController">
    <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

Gráfico 35. Declaración del bean FlowController

SimpleUrlHandlerMapping

SimpleUrlHandlerMapping permite que a través de una url entremos en el flujo

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <value>inicio.flow=flowController</value>
    </property>
    <property name="alwaysUseFullPath" value="true"/>
</bean>
```

Gráfico 36. Declaración del bean SimpleUrlHandlerMapping

ENTRAR AL FLUJO

Con el siguiente link en la página index.jsp conseguiremos entrar al flujo.

```
<a href="inicio.flow">Entrar al flujo</a>
```

Gráfico 37. Entrar al flujo

PAGINA INICIO

El flujo nos llevará a la primera página inicio.jsp.

Esta página contiene un link donde se envía el evento "ok1" para pasar al siguiente estado.

```
<h1>Pagina Inicio</h1>

<a href="${flowExecutionUrl}&_eventId=ok1">
    Pasar al siguiente estado
</a>
```

Gráfico 38. Página inicio.jsp

PAGINA MEDIO

Esta página contiene un link donde se envía el evento "ok2" para pasar al siguiente estado.

```
<h1>Pagina Intermedio</h1>

<a href="${flowExecutionUrl}&_eventId=ok2">
    Pasar al siguiente estado
</a>
```

Gráfico 39. Página medio.jsp

PAGINA FINAL

```
<h1>Fin del flujo</h1>
```

Gráfico 40. Página final.jsp

El código de este ejemplo lo encontrareis en **Ejemplo1_Spring_MVC_WebFlow.zip**

EJEMPLO CON ESTADOS DE ACCION Y DECISION

El siguiente ejemplo se trata de presentar al usuario un formulario para introducir un numero mediante un estado de vista.

Con un estado de acción almacenaremos el numero en una variable con ámbito de flujo.

Comprobaremos si el numero es par o impar mediante un estado de decision. Si es par iremos al estado de vista par y si no lo es al impar.

CREAR LA CLASE VALIDADOR

En una clase muy sencilla implementamos la logica de negocio para comprobar si el numero es par o impar.

```
public class Validador {
    public boolean comprobar(int numero){
        if (numero % 2 == 0){
            return true;
        }else{
            return false;
        }
    }
}
```

Gráfico 41. Clase Validador

DECLARAR UN BEAN DE LA CLASE VALIDADOR

En el dispatcher-servlet.xml declaramos un bean de tipo Validador.

```
<bean id="validador" class="app.negocio.Validador" />
```

Gráfico 42. Declara un bean validador

DECLARAR EL FLUJO

```
<view-state id="entrada" view="inicio">
      <transition on="ok1" to="guardarNumero" />
</view-state>

<action-state id="guardarNumero" >
    <evaluate expression="requestParameters.num"
              result-type="int" result="flowScope.numero" />
    <transition to="comprobarNumero" />
</action-state>

<decision-state id="comprobarNumero">
      <if test="validador.comprobar(numero)"
          then="numeroPar" else="numeroImpar" />
</decision-state>

<view-state id="numeroPar" view="mostrarPar">
      <transition on="terminar" to="fin" />
</view-state>

<view-state id="numeroImpar" view="mostrarImpar">
      <transition on="terminar" to="fin" />
</view-state>

<end-state id="fin" view="final" />
```

Gráfico 43. Declarar el flujo

El código de este ejemplo lo encontrareis en **Ejemplo2_Spring_MVC_WebFlow_ParImpar.zip**

SUBFLUJOS

Un subflujo es un nuevo flujo que se inicia dentro de otro. Cada flujo o subflujo se declara en un archivo xml aparte.

FLUJO PRINCIPAL

```
<view-state id="entrada" view="inicio">
      <transition on="ok1" to="entradaSubflujo" />
</view-state>

<subflow-state id="entradaSubflujo" subflow="subflujo">
    <output name="numero" />
    <transition to="comprobarNumero" />
</subflow-state>

<decision-state id="comprobarNumero">
      <if test="validador.comprobar(numero)"
          then="numeroPar" else="numeroImpar" />
</decision-state>

<view-state id="numeroPar" view="mostrarPar">
      <transition on="terminar" to="fin" />
</view-state>

<view-state id="numeroImpar" view="mostrarImpar">
      <transition on="terminar" to="fin" />
</view-state>

<end-state id="fin" view="final" />
```

Gráfico 44. Flujo principal

SUBFLUJO

```
<action-state id="guardarNumero" >
    <evaluate expression="requestParameters.num"
              result-type="int" result="flowScope.numero" />
    <transition to="mostrarMsg" />
</action-state>

<view-state id="mostrarMsg" view="numGuardado">
      <transition on="ok2" to="finSubflujo" />
</view-state>

<end-state id="finSubflujo">
    <output name="numero" />
</end-state>
```

Gráfico 45. Subflujo

El código de este ejemplo lo encontrareis en **Ejemplo3_Spring_MVC_WebFlow_ParImpar_Subflujos.zip**

INDICE DE GRÁFICOS