# MANUAL DE ESTÁNDARES DE DESARROLLO WEB

## MANUAL DE DESARROLLO EN JSP

Junta de Castilla y León

Coordinador de la Información: Área de Desarrollo y Mantenimiento
**Dirección General de Telecomunicaciones y Transportes.**

ÍNDICE DE CONTENIDOS

CAPITULO

# 1 Normas Generales de Codificación JSP

La intención del documento es la de presentar una serie de normas y recomendaciones acerca de la codificación JSP. No consistirá en un catálogo de los tags propios del lenguaje JSP, y no pretende por tanto servir como referencia de la sintaxis del lenguaje como tal.

La mayoría de los desarrolladores JSP deberán seguir la misma regla a la hora de escribir su código :

**"Una página JSP se deberá parecer a una página HTML con poco contenido dinámico y no a un programa Java con alguna marca HTML".**

Hay dos reglas más que parten de esta primera premisa. Todo ello nos llevará a una clara separación de la presentación y contenido que es lo que se pretende:

- Asegurarse de que la lógica de negocio está donde tiene que estar: fuera de la JSP y de cualquier elemento ligado a visualización. Los scripts JSP, es decir, el código Java embebido en la página JSP no sólo obscurece la página, sino que no es reutilizable y no sigue las reglas de una modelo orientado a objetos.

- Asegurarse de que la presentación está donde tiene que estar: en páginas JSP y siempre con código HTML explícito, nunca generando código HTML desde un script JSP. La presentación forma parte de las JSPs, pero hay que evitar generar HTML con código Java.

# Introducción

JavaServer Pages™ (JSP™) combinan HTML con fragmentos de Java para producir páginas web dinámicas. Cada página es automáticamente compilada a servlet por el motor de JSP , en primer lugar es recogida y a continuación ejecutada.

JSP tiene gran variedad de formas para comunicarse con las clases de Java, servlets, applets y el servidor web; por esto se puede aplicar una funcionalidad a nuestra web a base de componentes.

Una página JSP es archivo de texto simple que consiste en contenido HTML o XML con elementos JSP. Cuando un cliente pide una página JSP del sitio web y no se ha ejecutado antes, la página es inicialmente pasada al motor de JSP, el cual compila la página convirtiéndola en Servlet, la ejecuta y devuelve el contenido de los resultados al cliente.

Es posible ver el código del servlet generado, este código debe estar en el directorio o directorios que el servidor de aplicaciones tiene al efecto.

Nos encontramos también tres métodos:

- JspInit()
- JspDestroy()
- _jspService(HttpServletRequest request, HttpServletResponse response)

Los dos primeros métodos pueden ser definidos por el autor de la página JSP, pero el tercer método es una versión compilada de la página JSP, y su creación es responsabilidad del motor de JSP.

El código fuente de una página JSP incluye:

1. Directivas: Dan información global de la página, por ejemplo, importación de sentencias, página que maneja los errores o cuando la página forma parte de una sesión, en el ejemplo anterior informamos del tipo de script de Java.

2. Declaraciones: Sirven para declarar métodos y variables.

3. Scripts de JSP, también llamados *scriptlets*: Es el código Java embebido en la página.

4. Expresiones de JSP: Formatea las expresiones como cadenas para incluirlas en la página de salida. Estos elementos siguen una sintaxis como XML, así se obtiene un significado con una presentación totalmente separada de la lógica. Un buen ejemplo es `<jsp:useBean.../>` el cual busca o crea una instancia de un bean. Con el mecanismo de extensiones de tag se tiene la posibilidad de definir tags con acciones similares y poner la funcionalidad en una librería de tags.

# Ventajas ofrecidas por JSP

Los beneficios ofrecidos por JSP como alternativa a la generación de contenido dinámico para la web se resumen en los siguientes apartados.

## Mejoras en el rendimiento

- Utilización de procesos ligeros (hilos Java) para el manejo de las peticiones.
- Manejo de múltiples peticiones sobre una página .jsp en un instante dado.
- El contenedor servlet puede ser ejecutado como parte del servidor web.
- Facilidad para compartir recursos entre peticiones (hilos con el mismo padre: servlet container).

## Soporte de componentes reutilizables

- Creación, utilización y modificación de JavaBeans del servidor.
- Los JavaBeans utilizados en páginas .jsp pueden ser utilizados en servlets, applets o aplicaciones Java.

## Separación entre código de presentación y código de implementación

- Cambios realizados en el código HTML relativos a cómo son mostrados los datos, no interfieren en la lógica de programación y viceversa.

## División del trabajo

- Los diseñadores de páginas pueden centrarse en el código HTML y los programadores en la lógica del programa.
- Los desarrollos pueden hacerse independientemente.
- Las frecuentes modificaciones de una página se realizan más eficientemente.

# Reglas generales de sintaxis

## Elementos JSP

La sintaxis JSP está basada en XML. Todos los elementos basados en la sintaxis XML tienen un tag de inicio que incluye el nombre del elemento (posiblemente con atributos), un cuerpo opcional, y su correspondiente tag de fin. Ejemplo:

```
<mytag attr1="valor atributo" ... >
cuerpo
</mytag>
```

ó

```
<mytag attr1="valor atributo" ... />
```

Los tags JSP serán sensibles a mayúsculas y minúsculas al igual que XML. Los scripts JSP y directivas seguirán la sintaxis: `<% .... %>`.

## Inicio y fin de tags

Los elementos que tienen inicio y fin de tag (con cuerpo) deberán empezar y terminar en el mismo fichero. No se puede empezar un tag en un fichero y terminarlo en otro.

Esto se aplica también a los elementos con sintaxis alternativa, Por ejemplo, los fragmentos de código `<% scriptlet %>`, los caracteres de apertura `<%` y cierre `%>` deberán estar en el mismo fichero.

## Elementos vacíos

Siguiendo las especificaciones XML, un elemento descrito por un tag vacío es indistinguible de uno que use tag de inicio, fin y cuerpo.

## Valores de los atributos

Siguiendo las especificaciones XML, los valores de los atributos deberán siempre aparecer entre comillas. Se puede usar tanto comillas simples como dobles. Las entidades `&apos;` y `&quot;` son permitidas para describir comillas simples y dobles.

## Espacios en blanco

En HTML y XML los espacios en blanco no suelen ser significativos, con algunas excepciones. Una excepción es que un fichero XML debe empezar con `<?xml` sin poder dejar ningún espacio en blanco.

La especificación al respecto seguirá el comportamiento definido para XML: todos los espacios en blanco dentro del cuerpo del documento no serán significativos pero se reservarán.

Por ejemplo, como las directivas no generan datos, ocurriría lo siguiente:

Entrada:

| | |
|---|---|
| | `<?xml versión 1.0 ?>` |
| | `<%@ page buffer=”8k” %>` |
| | `Resto del documento` |

Salida:

| | |
|---|---|
| | `<?xml versión 1.0 ?>` |
| `Línea vacía` | |
| | `Resto del documento` |

# Manejo de errores

Hay dos fases en el proceso de generación de un JSP:

- Traducción (o compilación) de la página JSP.
- Procesamiento de la página.

Los errores se pueden dar en cualquiera de las dos fases anteriores. Se pretende en este apartado describir cómo tratar estos errores.

## Errores en tiempo de compilación

Puede haber fallos en la traducción de una página JSP en su correspondiente clase de implementación. Como resultado de esto, el desarrollador se encontrará con errores de compilación que le serán reportados a través del navegador.

Aunque la forma exacta de este aviso varía de una implementación a otra, la mayoría de los contenedores JSP reportan un error que se basa en un código en el servlet JSP. Por ejemplo, para protocolos http, aparecerá el código de error 500 en la mayoría de los casos.

Sin embargo cuando se trata de errores de compilación por mala escritura del código, este error se mostrará de forma bastante clara en la navegador pudiendo el desarrollador encontrar el error fácilmente.

## Errores en el procesamiento

Durante el procesamiento de la petición del cliente, pueden ocurrir errores arbitrarios en tiempo de ejecución. Estos errores utilizan el mecanismo de excepciones del lenguaje Java. Estas excepciones se pueden capturar y tratar en el cuerpo de la JSP.

Sin embargo, cualquier excepción no capturada se transformará en una redirección a la errorPage definida en la directiva `<@page...errorPage=""..>`.

El `java.lang.Throwable` que describe la excepción se almacenará en `el javax.ServletRequest` usando el método `putAttribute()`, con el nombre `javax.servlet.jspException`. Los nombres que empiezan por `"java"` o `"javax"` están reservados por las diferentes especificaciones de la plataforma Java, el prefijo `"javax.servlet"` es usado por las especificaciones JSP/Servlet.

## Cómo añadir páginas de error

Aunque las llamemos páginas de error, las páginas especializadas JSP que describimos aquí realmente muestran información sobre excepciones. Para añadir páginas de error que muestren información de excepciones a una aplicación web, se deberán seguir estos pasos:

1. Incluir en una JSP una directiva “errorpage” de forma que en caso que se produzca una excepción, dicha directiva indique cuál será la página JSP que atenderá el error.

2. Crear una JSP encargada de atender el error detectado en la JSP del punto 1. Debe contener la directiva page con `isErrorPage="true"`.

3. En el fichero de la página de error, usa el objeto exception para obtener información sobre la excepción.

4. Usamos mensajes informativos, en nuestra página de error o incluida desde otros ficheros, para darle al usuario un mensaje informativo sobre lo que el usuario estaba haciendo cuando se lanzó la excepción.

# Comentarios

Todos los desarrolladores experimentados usan Javadoc, pero las JSP no ofrecen esta herramienta.

El primer paso para una buena documentación JSP es ubicar el código Java en beans y manejadores de tags en vez de en las JSP. Esto permitirá usar el javadoc para generar la documentación de la mayoría de la lógica.

En cualquier caso, habrá dos tipos de comentarios en una página JSP, comentarios para la página JSP en sí misma, comentando lo que la página está haciendo, y comentarios que se pretenda que aparezcan en el documento resultado que se manda al cliente.

## Comentarios de salida al cliente

La sintaxis es la siguiente:

```
<!-- comentarios... -->
```

El contenedor JSP no interpretará estos comentarios, serán incluidos en la salida de la página .jsp, y por lo tanto serán visibles en el código fuente que le llegue al cliente.

Es el tipo estándar de comentario para HTML y XML. Se puede incluir contenido dinámico en el comentario (expresiones JSP), y el valor de la expresión aparecerá como parte del comentario en el cliente:

```
<!-- comentarios <%= expresión %> más comentarios ... -->
```

## Comentarios JSP

Estos comentarios sólo pueden ser vistos examinando la página .jsp original. Son de la forma:

```
<%-- comentarios... --%>
```

Una forma alternativa de introducir un comentario en JSP es hacerlo como dictan las normas Java para ello:

```
<% // Esto es un comentario %>

<% /* Esto es un comentario */ %>
```

# Convenciones de escritura para caracteres especiales

## Scripts

Un literal `%>` se escribiría como `%\>`

## Texto

Un literal `<%` se escribiría como `<\%`

## Atributos

- Comilla simple ‘ se escribiría como \’.
- Comilla doble “ se escribiría como \”.
- Un `\` se escribiría como `\\`.
- Un `%>` se escribiría como `%\>`.
- Un `<%` se escribiría como `<\%`.

# Uso de includes

Hay un peligro de duplicación de código en las páginas JSP. La forma más eficaz de evitarlo es utilizar los `includes` para introducir fragmentos JSP que puedan ser reutilizables en muchas páginas. Hay dos formas de incluir datos:

| Sintaxis | Qué es | Fase | Objeto | Descripción |
|---|---|---|---|---|
| `<%@ include file=... %>` | directiva | Compilación | estático | El contenido es incluido en tiempo de compilación. |
| `<jsp:include page=... />` | acción | Request-time | dinámico | El contenido es incluido sin ser parseado |

### include estático

Se trata de una directiva que nos permite incluir ficheros en el momento en que la página JSP es traducida a un servlet. Ejemplo:

```
<%@ include file="list.jsp %>
```

La URL especificada normalmente se interpreta como relativa a la página JSP a la que se refiere, pero, al igual que las URLs relativas en general, podemos decirle al sistema que interpreta la URL relativa al directorio home del servidor Web empezando la URL con una barra invertida. Los contenidos del fichero incluido son analizados como texto normal JSP, y así pueden incluir HTML estático, elementos de script, directivas y acciones.

### include dinámico

Se trata de una acción que nos permite insertar ficheros en una página que está siendo generada. Ejemplo:

```
<jsp:include file="include.jsp"  flush="true" />
```

Al contrario que la directiva include, que inserta el fichero en el momento de la conversión de la página JSP a un Servlet, esta acción inserta el fichero en el momento en que la página es solicitada. Esto se paga un poco en la eficiencia, e imposiblita a la página incluida de contener código JSP general (no puede seleccionar cabeceras HTTP, por ejemplo), pero se obtiene una significativa flexibilidad.

También se podrá utilizar como una subtag del <jsp:include> el tag <jsp:param>, para pasarle parámetros a la página incluida.

Ejemplo:

```
<jsp:include file="include.jsp  flush="true" %>
    <jsp:param name="Param1" value="param1" />
    <jsp:param name="Param2" value="param2" />
</jsp:include>
```

# Uso de Tag Extensions

El uso de tag Extensions es una característica muy potente del JSP 1.1. Incrementa la habilidad de mantener la lógica en Java y no son mas que clases java que implementan o heredan de determinadas clases.

## Ventajas de su uso

- Potencian la reutilización de código usado frecuentemente.
- Permiten limpiar de código las JSP sustituyendo la mayoría del código embebido por etiquetas. Esto mejora el mantenimiento y la legibilidad de las páginas.
- Permiten separar más los roles en el desarrollo de JSP. Esto permite un mayor paralelismo en el trabajo.
- Cuando ya se disponga de una buena biblioteca de taglibs se acelerará el desarrollo de páginas.
- Debido a que están basadas en XSL permiten, además de simplemente añadir código a la página, transformarla con gran potencia. Un ejemplo de las ventajas de que proporciona el hecho de que sean hojas de transformación XSL es que podemos definir etiquetas para invocar de forma sencilla métodos con signaturas complejas. Si preparamos valores por defecto para cada argumento podemos comprobar que atributos se nos pasan en la etiqueta y para los que falten colocar el valor por defecto.

## Directiva taglib

La directiva **`taglib`** se utilizará para notificar al contenedor JSP que la página utilizará una o más librerías de etiquetas personalizadas. Una librería de etiquetas es una colección de etiquetas personalizadas, que pueden ser utilizadas para extender la funcionalidad básica de JSP:

```
<%@ taglib uri="URIdeLaLibreriaDeEtiquetas" prefix="prefijoEtiquetas" %>
```

El valor del atributo **uri** indica la localización del TLD (Tag Library Descriptor), mientras que el atributo **prefix** especifica el identificador del espacio de nombres en XML que será anexado a todas la etiquetas de esa librería que se utilicen en la página .jsp.

Etiquetas personalizadas:

```
<prefijoEtiqueta:nombre atributo="valor"+ ... />
```

```
<prefijoEtiqueta:nombre atributo="valor"+ ... >
  otras etiquetas
</prefijoEtiqueta:nombre>
```

Por razones de seguridad, la especificación JSP permite utilizar solamente TLD's del sistema de archivos local.

## Definición de un Tag

Una etiqueta es una clase Java que implementa un interface especializado. Se usa para encapsular la funcionalidad desde una página JSP. Una etiqueta puede o no tener cuerpo. Para definir una sencilla etiqueta sin cuerpo, nuestra clase debe implementar el interface **`Tag`**.

A continuación se muestra un ejemplo del código fuente del interface Tag que hay que implementar:

```
public interface Tag {
   public final static int SKIP_BODY = 0;
   public final static int EVAL_BODY_INCLUDE = 1;
   public final static int SKIP_PAGE = 5;
   public final static int EVAL_PAGE = 6;

   void setPageContext(PageContext pageContext);
   void setParent(Tag parent);
   Tag getParent();
   int doStartTag() throws JspException;
   int doEndTag() throws JspException;
   void release();
}
```

Todas las etiquetas deben implementar el interface Tag (o uno de sus interfaces) como si definiera todos los métodos que el motor de ejecución JSP llama para ejecutar una etiqueta.

| Método | Descripción |
|---|---|
| **setPageContext (PageContext pc)** | A este método lo llama el motor de ejecución JSP antes de doStartTag, para seleccionar el contexto de la página. |
| **setParent(Tag parent)** | Invocado por el motor de ejecución JSP antes de doStartTag, para pasar una referencia a un controlador de etiqueta a su etiqueta padre. |
| **getParent** | Devuelve un ejemplar Tag que es el padre de esta etiqueta. |
| **doStartTag** | Invocado por el motor de ejecución JSP para pedir al controlador de etiqueta que procese el inicio de etiqueta de este ejemplar. |
| **doEndTag** | Invocado por el motor de ejecución JSP después de retornar de doStartTag. El cuerpo de la acción podría no haber sido evaluado, dependiendo del valor de retorno de doStartTag. |
| **release** | Invocada por el motor de ejecución JSP para indicar al controlador de etiqueta que realice cualquier limpieza necesaria. |

Ahora, veamos una etiqueta de ejemplo que cuando se le invoca imprime un mensaje en el cliente.

Hay unos pocos pasos implicados en el desarrollo de una etiqueta personalizada. Estos pasos son los siguientes:

1. Desarrollar el controlador de etiqueta
2. Crear un descriptor de librería de etiqueta
3. Probar la etiqueta

***Desarrollar el controlador de etiqueta***

Un controlador de etiqueta es un objeto llamado por el motor de ejecución JSP para evaluar la etiqueta personalizada durante la ejecución de una página JSP que referencia la etiqueta. Los métodos del controlador de etiqueta son llamados por la clase de implementación en varios momentos durante la evaluación de la etiqueta. Cada controlador de etiqueta debe implementar un interface especializado.

***Descriptor de librería de etiqueta***

El siguiente paso es especificar cómo ejecutará la etiqueta el motor de ejecución JSP . Esto puede hacerse creando un "Tag Library Descriptor" (TLD), que es un documento XML.

***Probar la etiqueta***

El paso final es probar la etiqueta que hemos desarrollado. Para usar la etiqueta, tenemos que referenciarla, y esto se puede hacer de tres formas:

1. Referenciar el descriptor de la librería de etiquetas de una librería de etiquetas desempaquetada. Por ejemplo:

```
<@ taglib uri="/WEB-INF/jsp/mytaglib.tld" prefix="first" %>
```

2. Referenciar un fichero JAR que contiene una librería de etiquetas. Por ejemplo:

```
<@ taglib uri="/WEB-INF/myJARfile.jar" prefix='first" %>
```

3. Definir una referencia a un descriptor de la librería de etiquetas desde el descriptor de aplicaciones web (web.xml) y definir un nombre corto para referenciar la librería de etiquetas desde al JSP. Para hacer esto, abrimos el fichero: c:\tomcat\webapps\examples\web-inf\web.xml y añadimos las siguientes líneas antes de la última línea, que es <web-app>:

```
   <taglib>
<taglib-uri>mytags</taglib-uri>
<taglib-location>/WEB-INF/jsp/
mytaglib.tld</taglib-location>
   </taglib>
```

## JDeveloper Data Tag Library

JDeveloper, a partir de la versión 3.2, ofrece una facilidad de manejo de las JSP gracias a la librería de tags para el acceso de datos que incluye (DataTags.tld).

El acceso a los datos asociados a los Business Components, aportan una manera muy rápida y fácil de construir aplicaciones JSP, de forma que se consigue tener un control completo sobre el diseño HTML y las operaciones de acceso a base de datos.

Además Oracle JDeveloper también ofrece un conjunto de wizards para trabajar de una forma más eficiente y rápida si cabe con los BC4J JSP Data Tags. Aparece también el DataPage Wizard que ofrecerá además la posibilidad de generar páginas JSP basadas en los BC4J Views.

Oracle JDeveloper BC4J Data Tag Library, por lo tanto, permitirá:

- Un manejo eficiente de los Application Modules.
- Presentación, navegación y edición de los BC4J.
- Construcción de formularios HTML asociados a datos de una forma muy rápida y eficaz.

# Uso de JSP Beans

La manera más sencilla de reducir el código Java en una página JSP es usar uno o más JSP Beans.

La acción `<jsp:useBean>` nos permite cargar y utilizar un JavaBean en la página JSP. Esto nos facilita (modelo MVC) la separación entre lógica de la aplicación y presentación, encapsulando el código en el JavaBean y reutilizando las clases Java ya construidas, mediante invocación a sus métodos desde las páginas JSP.

```
<jsp:useBean id="name" class="package.class" />
```

Esto normalmente significa "instancia un objeto de la clase especificada por class, y lo asocia a una variable con el nombre especificado por id". Sin embargo, también podemos especificar un atributo scope que hace que ese Bean se asocie con más de una sóla página. En este caso, es útil obtener referencias a los beans existentes, y la acción jsp:useBean especifica que se instanciará un nuevo objeto si no existe uno con el mismo nombre y ámbito.

Ahora, una vez que tenemos un bean, podemos modificar sus propiedades mediante `<jsp:setProperty>`, o usando un scriptlet y llamando a un método explícitamente sobre el objeto con el nombre de la variable especificada anteriormente mediante el atributo id. Con los beans, cuando decimos "este bean tiene una propiedad del tipo X llamada foo", realmente queremos decir "Esta clase tiene un método getFoo que devuelve algo del tipo X, y otro método llamado setFoo que toma un X como un argumento". La acción `<jsp:setProperty>` nos permitirá suministrar un valor explícito, dando un atributo param para decir que el valor está derivado del parámetro de la petición nombrado, o sólo lista las propiedades para indicar que el valor debería derivarse de los parámetros de la petición con el mismo nombre que la propiedad. Leemos las propiedades existentes en una expresión o scriptlet JSP llamando al método `getXxx`, o más comúnmente, usando la acción `jsp:getProperty`.

### Page Beans

Estos beans están íntimamente relacionados con la JSP que los usan. Sólo se deberán usar en varias páginas si éstas representan el mismo contenido en distinto formato.

Normalmente estos beans definirán accesores (getters) para cada contenido a visualizar en la página. Estos getters devolverán valores primitivos u objetos más complejos, y la página JSP será la responsable de mostrar estos objetos (si son complejos, normalmente se hará a través de tags personalizados).

Estos beans podrán tratar los parámetros del request, pero no deberán tratar lógica de negocio compleja. (por ejemplo, no deberían diseñarse para manejar redirecciones).

### Session Beans

Estos beans ya no están ligados a la representación de la página, sino que podrán hacer referencia a recursos de lógica de negocio o a cualquier información, permitiendo al usuario mantener en la sesión.

Por ejemplo, un session bean se podrá utilizar para mantener el nombre y email del usuario, y así poder hacer referencia a esta información desde cualquier otro bean para sacar más datos.

Los session beans nunca se deberán utilizar para procesamiento del request.

### Application Beans

Estos beans mantienen el estado para todos los usuarios de una aplicación. Se usan menos que los dos anteriores, pero pueden ser útiles para minimizar el consumo de memoria y mejorar el rendimiento, especialmente cuando muchos usuarios necesiten acceder a los mismos datos.

Application Beans se suelen usar para sólo lectura de datos. Cuando se quieren utilizar para lectura-escritura, se tendrá que hacer un tratamiento de sincronización y bloqueos.

### Ventajas y desventajas

La regla más importante con beans es la de que no deberán generar HTML. Tampoco deberán procesar lógica de negocio que no sea realmente necesaria para la representación JSP.

También es importante ser precavido con el uso de la sesión puesto que hoy en día es muy usual la implementación de clustering de servidor, con lo que se perderá la sesión sino hacemos nada para evitarlo.

# Objetos implícitos

En una página JSP se tiene acceso a algunos objetos que están siempre disponibles sin necesidad de haberlos declarado anteriormente. Estos son:

| Objeto | Del tipo | Lo que representa | Scope |
|---|---|---|---|
| request | Subtipo de javax.servlet.ServletRequest Ejemplo: Javax.servlet.HttpServletRequest | Petición del cliente | request |
| response | Subtipo de javax.servlet.ServletResponse Ejemplo: javax.servlet.HttpServletResponse | Página JSP de respuesta | page |
| pageContext | javax.servlet.http.PageContext | El motor de JSP utiliza la clase Factory para devolver la implementación de clase PageContext del servidor. Esta clase PageContext es inicializada con los objetos response y request y algunos atributos de la directiva de la página (erropage, session,buffer and autoflush) y facilita los otros objetos implícitos para la pagina de petición. | page |
| session | javax.servlet.http.HttpSession | El objeto se sesión HTTP asociado a la petición del cliente | session |
| application | javax.servlet.ServletContext | El contexto servlet obtenido de la configuración del servlet (getServletConfig(). getServletContext()) | application |
| out | javax.servlet.jsp.JspWriter | Objeto que representa la salida de texto por pantalla. | page |
| config | javax.servlet.ServletConfig | El objeto ServletConfig de la página JSP | page |
| page | java.lang.Object | Es la forma que tiene la página para referirse a si misma. Se usa como alternativa al objeto this. | page |

Además, en una página de error, se podrá acceder al objeto `exception` descrito a continuación:

| Objeto | Del tipo | Lo que representa | Scope |
|---|---|---|---|
| exception | java.lang.Throwable | La excepción que no fue capturada en la página que invocó la página de error | page |

Estos objetos tienen la funcionalidad común de establecer y recuperar valores de atributos como mecanismo de intercambio de información. Los métodos estándares para manejo de atributos son los que se muestran a continuación:

| Método | Descripción |
|---|---|
| setAttribute(key, value) | Asocia un atributo con su valor correspondiente. |
| getAttributeNames() | Retorna los nombres de todos los atributos asociados con la sesión. |
| getAttribute(key) | Retorna el valor del atributo asociado con key. |
| removeAttribute(key) | Borra el atributo asociado con key |

# Elementos de script

## Declaraciones

Se usan para declarar variables o métodos válidos en Java. Se inserta fuera del método `service()` del servlet que se genera a partir de la página JSP.

Un uso importante de la declaración de métodos es el manejo de eventos relacionados con la inicialización y destrucción de la página .jsp. La inicialización ocurre la primera vez que el contenedor JSP recibe una petición de una página .jsp concreta. La destrucción ocurre cuando el contenedor JSP descarga la clase de memoria o cuando el contenedor JSP se reinicia.

Sintaxis:

```
<%! Esto es una declaración %>
```

El espacio en blanco después de "`<%!`" y antes de "`%>`" es opcional.

Ejemplo:

```
<%! int i = 0; %>
<%! public String f(int i) { if (i<3) return ("..."); .. %>
```

## Scriplets

Los scriptlets JSP nos permiten embeber segmentos de código java dentro de una página JSP. El código embebido se inserta directamente en el servlet generado que se ejecuta cuando se pide la página. Este scriptlet usa las variables declaradas en las directivas descritas arriba. Los Scriptlets van encerradas entre etiquetas <% y %>.

Debemos terminar una sentencia de lenguaje con un punto y coma si el lenguaje los requiere. Cuando escribamos un scriptlet podemos usar cualquiera de los objetos o clases implícitos de JSP importados por la directiva page, declarada en una declaración, o nombrada en una etiqueta <jsp:useBean> .

Sintaxis:

```
<% Esto es un scriplet %>
```

El espacio en blanco después de "`<%`" y antes de "`%>`" es opcional.

Ejemplo:

```
<% if(Calendar.getInstance().get(Calendar.AM_PM) == Calendar.AM) { %>
```

```
    Buenos días
<% } else {%>
    Buenas tardes
<% } %>
```

## Expresiones

La expresiones JSP nos permiten recuperar dinámicamente o calcular valores a insertar directamente en la página JSP.

Todo resultado de una expresión es convertido a un String antes de ser añadido a la salida de la página. Las expresiones no terminan con un punto y coma (;).

Sintaxis:

```
<%= Esto es una expresión %>
```

El espacio en blanco después de "<%=" y antes de "%>" es opcional.

Ejemplo:

```
<% new java.util.Date()).toLocaleString() %>
```

# Recomendaciones generales

- Todas las páginas JSP deberán ser de sólo lectura de datos, con un bean asociado que sea el que de el modelo. La manera de representar los datos al usuario puede cambiar muchas veces, sin embargo la estructura del contenido cambiará mucho menos.

- Pensar verticalmente. Primero hay que identificar los datos que van a ser representados en la JSP y después diseñar el bean que proveerá del modelo de datos a la página. JSPs y JSP Beans se han de diseñar conjuntamente.

- Para páginas más complejas, hay que pensar y diseñar en términos de grupos de JSPs que colaboren a ofrecer pequeños fragmentos de funcionalidad cada una de ellas, en vez de pensar en una JSP como algo individual.

- La lógica condicional ha de situarse en páginas JSP que hagan de controladores en vez de en la propia página JSP (es decir, en la vista). Por ejemplo, si una parte substancial del JSP varía dependiendo de una condición, este JSP será un candidato para rehacerse y convertirse en un controlador, redireccionando éste a dos páginas separadas.

- Será conveniente seguir una convención para el nombrado de páginas JSP, dependiendo de qué tipo de ficheros estemos hablando:

    - Controlador JSP: `xxxxController.jsp`
    - JSP incluida: `_nombreDescriptivoDelFragmento.jsp`
    - Beans de página: `<pagename>Bean`
    - Beans de sesión: `xxxxSessionBean`
    - Manejadores de tags y clases relacionadas:

      `xxxxTag, xxxxTagExtraInfo`

- Las directivas de import deberán evitarse. El uso de imports provoca que se instancien otras clases a parte de los JSP beans. Por lo tanto, en vez de usar una directiva como ésta:

```
<%@ page import="com.java.util.*" %>
```

  será mejor referirse a cada clase escribiendo el nombre completo del paquete, por ejemplo:

```
<%@ java.util.List lista = new java.util.LinkedList(); %>
```

- Las páginas JSP no deberán acceder directamente a los parámetros del request como norma general. Para ello, deberán utilizarse controladores o beans. Esto es porque el acceso a los parámetros del request requieren normalmente chequeo de errores que complicaría la página JSP.

- Las páginas JSP no deberían acceder a los ficheros de propiedades o usar JNDI. Son los beans los que deberán hacerlo (por ejemplo, para proveer al sistema de internacionalización).

- Evitar el diseño de páginas que muestran un formulario y procesan al mismo tiempo el resultado. Utilizar para el resultado otra JSP.

- Evitar duplicación de código en un JSP. Esto significará que el diseño ha sido pobre. Este código duplicado se deberá transformar en un include, bean o tag extensión para hacerle reutilizable en toda la aplicación. Se recomienda utilizar includes estáticos frente a los dinámicos.

- Los JSP beans nunca han de generar código HTML. Precisamente JSP fue diseñada para evitar este tipo de técnicas.

- El uso de `out.println()` se deberá evitar en las JSP. Este uso provoca que se piense en las páginas JSP como programas en vez de páginas web, y esto es bastante confuso.

- La capa JSP nunca deberá acceder directamente a los datos, es decir, accesos JDBC y EJB, sino que los beans de las JSPs deberán acceder a objetos de la sesión que correspondan a la capa de la lógica de negocio.

- Los scriptlets nunca deberán superar las 5 líneas de longitud.

- Los JSP beans de la sesión no deberán contener una cantidad excesiva de datos.

- Si se utiliza `<jsp:forward>` o `<jsp:include>` y se quieren pasar valores a otra página externa, se deberá usar siempre que sea posible la directiva `<jsp:param>`. Si esto no fuera posible se deberá usar el método `setAttribute()` en el primer caso y un include estático en el segundo.

- Se deberán usar tag extensions para eliminar lógica de las páginas JSP.

- La directiva `<jsp:forward>` deberá usarse con cuidado. En realidad equivale a un `goto`, y hará que se pierda la idea de JSP como vista. Se deberá usar en ese caso un controlador o un bean.

- En general, las declaraciones y los scriptlets no se deberían usar para crear variables a las que se haga referencia dentro de la misma página.

- Los métodos `jspInit()` y `jspDestroy()` deberán usarse sólo si es estrictamente necesario. Usar estos métodos hace que la página JSP parezca más un servlet que una página web.

- Evitar manejo de excepciones en las JSP. Cada JSP deberá usar una página de error para manejar las excepciones. Las excepciones se deberán capturar en la capa JSP pero no por la página JSP en sí misma si es posible.

# Referencias


- JavaServer Pages Specification 1.2.
- JavaServer Pages. Dynamically generated web content (http://java.sun.com/products/jsp/index.html).
- Web Application Development with JSPTM and XML, Part III: Developing JSP Custom Tags, Java Developer ConnectionSM, August 2001
- Java Software Tutorial.
- Core Servlets and JavaServer Pages Tutorial (PDF).
- Jakarta-Taglibs.
- The binary and source distributions have a tutorial in the directory jakarta-taglibs/doc.
- Orion Server Tutorial

CAPITULO

# 2 Guía rápida de Codificación JSP

En el capítulo anterior se ha repasado la codificación para JSP y se han discutido algunos de sus usos ofreciendo en cada caso unas normas o recomendaciones para el correcto manejo del lenguaje.

En este capítulo se presenta una guía rápida con la intención de repasar todas las directivas presentando un ejemplo de su uso en cada una de ellas.

# Directivas JSP

## Directiva page

Los atributos correspondientes a esta directiva son los siguientes:

| Atributo | Descripción | Valor por defecto |
|---|---|---|
| language | Define el lenguaje de scripting que se va a utilizar. Preparado para usos futuros si el contenedor de JSPs permite otros lenguajes. | "Java" |
| extends | Esto indica la superclase del servlet que se va a generar. Debemos usarla con extrema precaución, ya que el servidor podría utilizar una superclase personalizada. | Se omite por defecto |
| import | Separados por coma aparecerán los paquetes o clases, de igual manera que se hace en una clase Java. | Se omite por defecto |
| session | Especifica si la página participa en un sesión http. Cuando es `"true"` el objeto implícito `session` está accesible y puede ser usado para acceder a la sesión actual. Si es `"false"` el objeto implícito `session` no estará accesible. | "true" |
| buffer | Esto especifica el tamaño del buffer para el `JspWriter out`. | Es específico del servidor, debería ser de al menos 8kb. |
| autoFlush | Un valor de `"true"` (por defecto) indica que el buffer debería desacargase cuando esté lleno. Un valor de `"false"`, raramente utilizado, indica que se debe lanzar una excepción cuando el buffer se sobrecargue. Un valor de `"false"` es ilegal cuando usamos `buffer="none"`. | "true" |
| isThreadSafe | Define el nivel de seguridad a nivel de thread implementado por la página. Si es `"true"` el motor de JSP podrá mandar múltiples request a la página al mismo tiempo. Si es `"false"` el procesador JSP irá encolando las peticiones de cliente y procesará sólo una al tiempo, en el orden en que las haya recibido, Es lo mismo que implementar el interface `javax.servlet.SingleThreadModel` en un servlet. | "true" |
| info | Define una información de la página que podrá ser accedida desde | Se omite por defecto |

| | | |
|---|---|---|
| | `Servlet.getServletInfo()`. | |
| errorPage | Define una URL a otra página JSP, que será invocada cuando se lance una excepción no tratada. La implementación de la página captura la instancia del objeto `Throwable` y se lo pasa a la página de error para su procesamiento. | Se omite por defecto |
| isErrorPage | Indica si la página actual es una página de error. Si es `"true"` la variable implícita `exception` estará accesible y hará referencia a una instancia de `java.lang.Throwable`. | "false" |
| contentType | Define el set de caracteres y el tipo MIME para la respuesta de la página. | El valor por defecto para `MIMETYPE` es `text/html`; el valor por defecto para `CHARSET` es `ISO-8859-1`. |

**Ejemplo**: `pageDirective.jsp`

```
<%page language="java" import="java.util.*, java.io.*"
  session="true" buffer="12kb" autoFlush="true"
  info="Mi directiva page" errorPage="errorPage.jsp"
  isErrorPage="false" isThreadSafe="false" %>

<html>
  <head>
    <title> Página de ejemplo para la directiva page</title>
    <body>
      <h1> Página de ejemplo para la directiva page </h1>

      Página de prueba para probar la directiva page

    </body>
  </head>
</html>
```

### Directiva `include`

El fichero a incluir puede ser estático (como un fichero html) o dinámico (otra JSP).

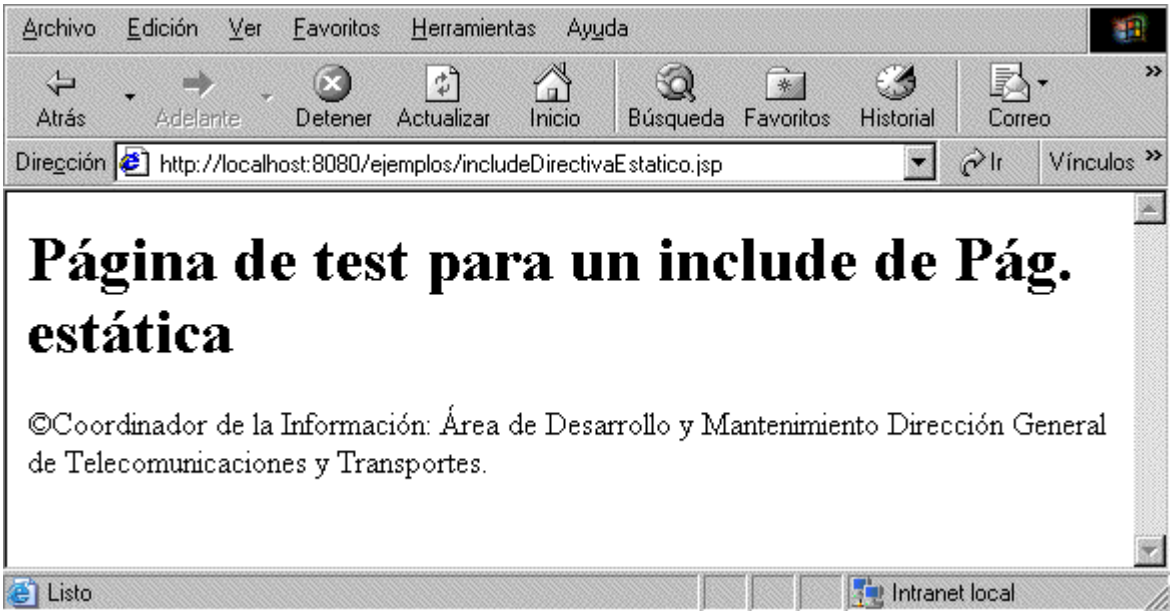
**Ejemplo**: `includeDirectivaEstatico.jsp`

```
<html>
  <head>
    <title> Página de test para un include de Pág. estática</title>
  </head>
  <body>
    <h1> Página de test para un include de Pág. estática </h1>

    <%@include file="/copyrigtht.html" %>

  </body>
</html>
```

El fichero `copyrigtht.html` contiene lo siguiente:

```
<p>&copy; Coordinador de la Información: Área de Desarrollo y
Mantenimiento
Dirección General de Telecomunicaciones y Transportes. </p>
```

El resultado sería el siguiente:



El siguiente ejemplo mostrará cómo incluir un fichero dinámico.

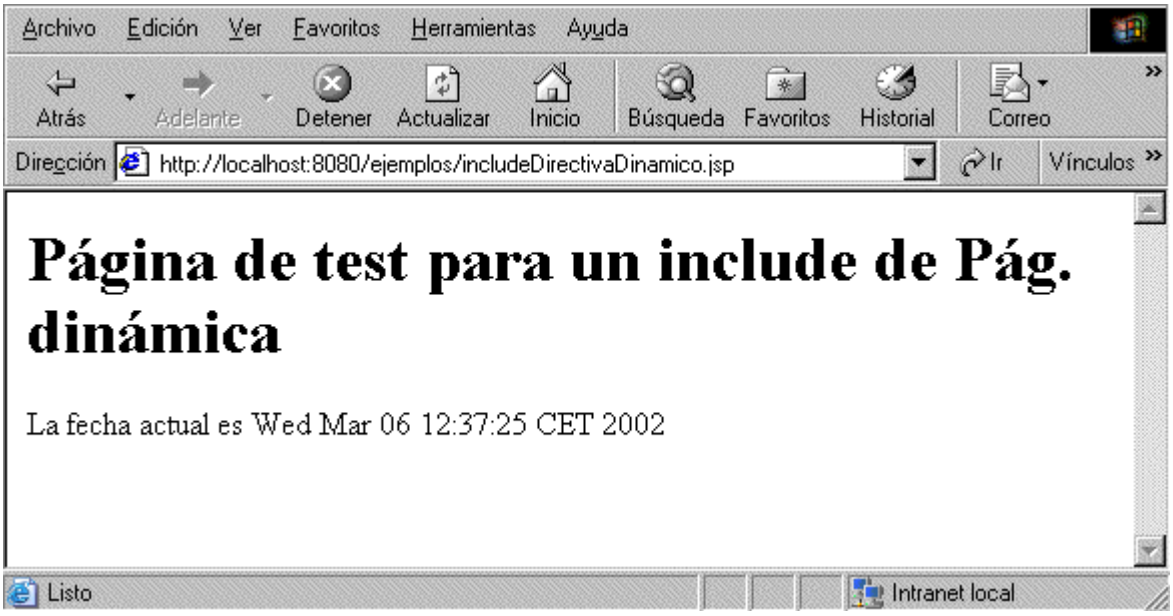
**Ejemplo**: `includeDirectivaDinamico.jsp`

```
<html>
  <head>
    <title>Página de test para un include de Pág. dinámica </title>
  </head>
  <body>
    <h1> Página de test para un include de Pág. dinámica </h1>

    <%@include file="included.jsp" %>

  </body>
</html>
```

El fichero `included.jsp` contiene lo siguiente:

```
<%@ page import="java.util.Date" %>
<%= "La fecha actual es " + new Date() %>
```

El resultado sería el siguiente:



### Directiva `taglib`

La directiva taglib permite a la página usar tag extensions. Los atributos de esta directiva son los siguientes:

| Atributo | Descripción | Valor por defecto |
|---|---|---|
| uri | Este atributo (Uniform Resource Identifier) identifica el descriptor de la librería de tags que es usado para el nombrado de forma única de los tags y para decirle al contenedor qué hacer con los tags específicos. | Si no se especifica un valor provoca un error de compilación. |
| tagPrefix | Define el prefijo que se usará para definir los tags. Los prefijos ya reservados son: jsp, JSPs, java, javax, servlet, sun y sunw. | Si no se especifica un valor provoca un error de compilación. |

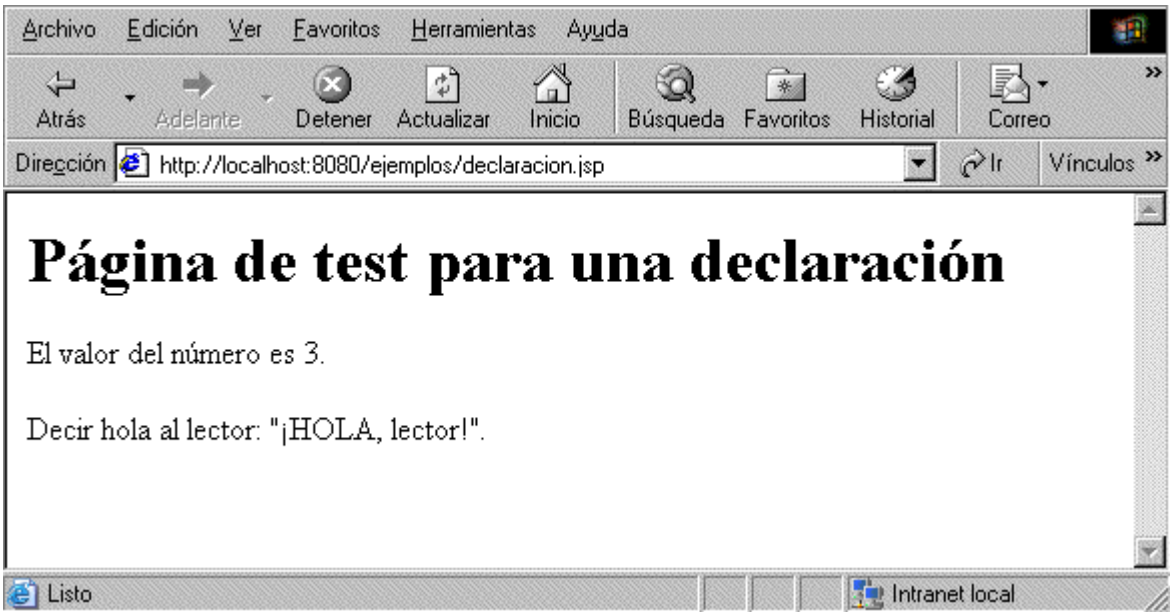
# Elementos de Scripting

## Declaraciones

**Ejemplo**: `declaracion.jsp`

```
<%!
  int numero =  3;
  public String decirHola(String nombre) {
    return "¡HOLA, " + nombre + "!";
  }
%>
<html>
  <head>
    <title>Página de test para una declaración</title>
  </head>
  <body>
    <h1> Página de test para una declaración </h1>

    <p>El valor del número es <%= numero %>.</p>
    <p>Decir hola al lector: "<%= decirHola("lector") %>".</p>

  </body>
</html>
```

El resultado sería el siguiente:

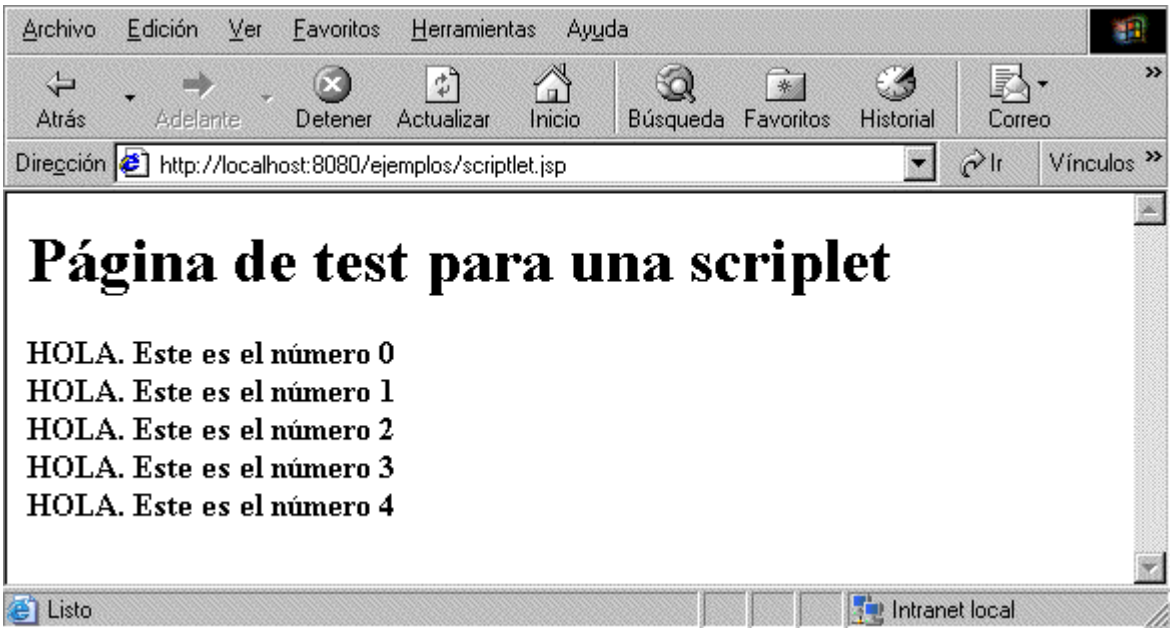


## Scriptlets

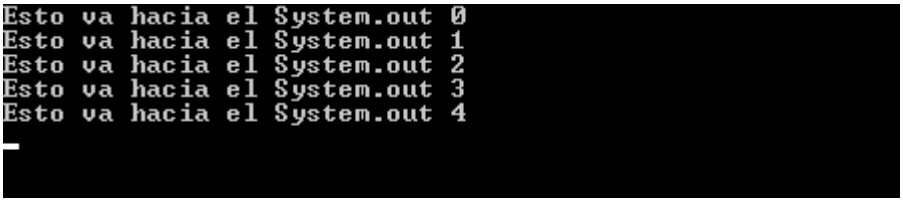
**Ejemplo**: `scriptlet.jsp`

```
<html>
  <head>
    <title>Página de test para un scriplet</title>
  </head>
  <body>
    <h1> Página de test para una scriplet </h1>
    <%
    for(int i=0; i<5; i++) {
      out.println("<b>HOLA. Este es el número " + i + "<b><br>");
      System.out.println("Esto va hacia el System.out " + i);
    }
    %>

  </body>
</html>
```

El resultado sería el siguiente:




y en la consola del Tomcat aparecerá lo siguiente:

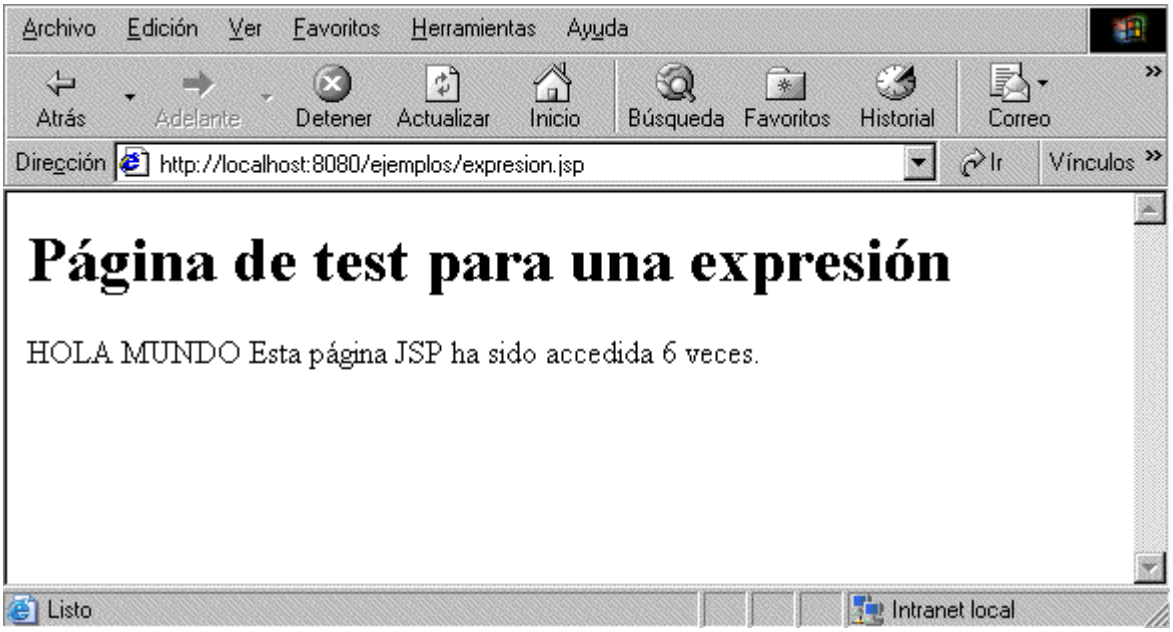


## Expresiones

**Ejemplo**: `expresion.jsp`

```
<html>
  <head>
    <title>Página de test para un expresión</title>
  </head>
  <body>
    <h1> Página de test para una expresión </h1>
    <%! int i = 0 ; %>
    <%
      i++;
    %>

    HOLA MUNDO
    <%= "Esta página JSP ha sido accedida " + i + " veces". %>

  </body>
</html>
```

El resultado, después de acceder a la página varias veces, sería el siguiente:



# Acciones Estándar

### <jsp:useBean>

Los atributos que componen esta acción son los siguientes

| Atributo | Descripción | Valor por defecto |
|---|---|---|
| id | Es el nombre que se usará para identificar la instancia del objeto. | No tiene ningún valor por defecto |
| scope | El ámbito en el que la instancia es disponible. Los valores posibles son "`page`", "`request`", "`session`" y "`application`". | "page" |
| class | El nombre completo de la clase. | No tiene ningún valor por defecto |
| beanName | El nombre del bean que se quiere instanciar. | No tiene ningún valor por defecto |
| type | Define el tipo de la variable que va a ser creada, siguiendo las reglas de casting estándares para Java. | El valor del atributo de la clase |

Veremos un ejemplo después de hacer el repaso de las acciones `getProperty` y `setProperty`, para que éste sea lo más completo posible.

### <jsp:setProperty>

Los atributos que componen esta acción son los siguientes

| Atributo | Descripción |
|---|---|
| name | Este atributo requerido designa el bean cuya propiedad va a ser seleccionada. El elemento `jsp:useBean` debe aparecer antes del elemento `jsp:setProperty`. |
| property | Este atributo requerido indica la propiedad que queremos seleccionar. Sin embargo, hay un caso especial: un valor de "*" significa que todos los parámetros de la petición cuyos nombres correspondan con nombres de propiedades del Bean serán pasados a los métodos de selección apropiados. |
| param | Este parámetro opcional designa el parámetro de la petición del que se debería derivar la propiedad. Si la petición actual no tiene dicho |

| | |
|---|---|
| | parámetro, no se hace nada: el sistema no pasa null al método seleccionador de la propiedad. Así, podemos dejar que el bean suministre los valores por defecto, sobrescribiéndolos sólo cuando el parámetro dice que lo haga.<br><br>Si se omite tanto `value` como `param`, es lo mismo que si suministramos un nombre de parámetro que corresponde con el nombre de una propiedad. Podremos tomar esta idea de automaticidad usando el parámetro de la petición cuyo nombre corresponde con la propiedad suministrada un nombre de propiedad de "*" y omitir tanto `value` como `param`. En este caso, el servidor itera sobre las propiedades disponibles y los parámetros de la petición, correspondiendo aquellas con nombres idénticos. |
| value | Este atributo opcional especifica el valor para la propiedad. Los valores `string` son convertidos automáticamente a números, boolean, Boolean, byte, `Byte`, `char`, y `Character` mediante el método estándar `valueOf` en la fuente o la clase envolvente. No podemos usar `value` y param juntos, pero si está permitido no usar ninguna. |

### <jsp:getProperty>

Los atributos que componen esta acción son lo siguientes:

| Atributo | Descripción |
|---|---|
| name | Nombre de un bean referenciado anteriormente mediante `jsp:useBean` |
| property | Propiedad cuyo valor debería ser insertado |

**Ejemplo**: El ejemplo preguntará al usuario su nombre y su lenguaje de programación favorito y mostrará el resultado de la elección.

**`Beans.html`** – Esta página html contendrá el formulario de elección.

```
<html>
  <head>
    <title>Página de test para la acción useBean</title>
  </head>
  <body>
    <h1> Página de test para la acción useBean </h1>
    <form method=”post” action=”beans.jsp”>
      <p> Por favor, introduzca su nombre:
      <input type=”text” name=”nombre”>
      <br> ¿Cuál es tu lenguaje de programación favorito?
      <select name=”lenguaje”>
        <option value=”Java”> Java
        <option value=”C++”> C++
        <option value=”Perl”> Perl
      </select>
      </p>
      <p><input type=”submit” value=”Aceptar”>
    </form>

  </body>
</html>
```

Esto enviará un request de tipo POST a la página beans.jsp con dos parámetros, el nombre y el lenguaje.

**`Beans.jsp`** :

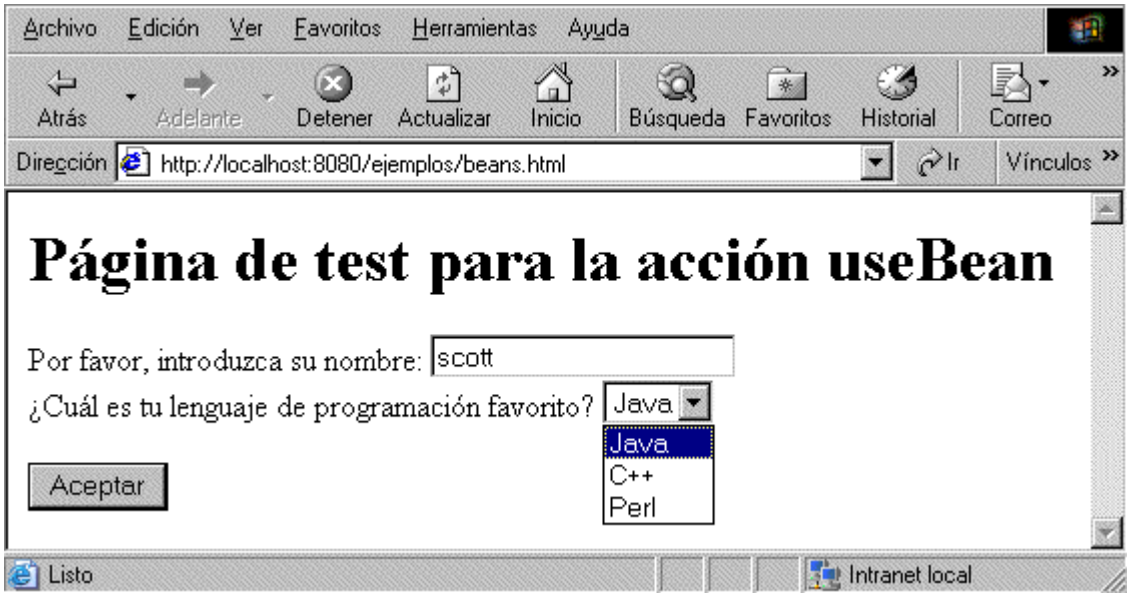
```
<jsp:useBean id=”lenguajeBean” scope=”page” class=”LenguajeBean”>
  <jsp:setProperty name=”lenguajeBean” property=”*” />
</jsp:useBean>

<html>
  <head>
    <title>Página de resultado del testeo del useBean</title>
  </head>
  <body>
    <h1> Página de resultado del testeo del useBean </h1>
    <p>Hola, <jsp:getProperty name=”lenguajeBean”
             property=”nombre” />.</p>
    <p>Su lenguaje favorito es
    <jsp:getProperty name=”lenguajeBean” property=”lenguaje” /></p>
    <p>Comentarios respecto al lenguaje elegido:</p>
    <p><jsp:getProperty name=”lenguajeBean”
        property=”lenguajeComentarios” /></p>
  </body>
</html>
```

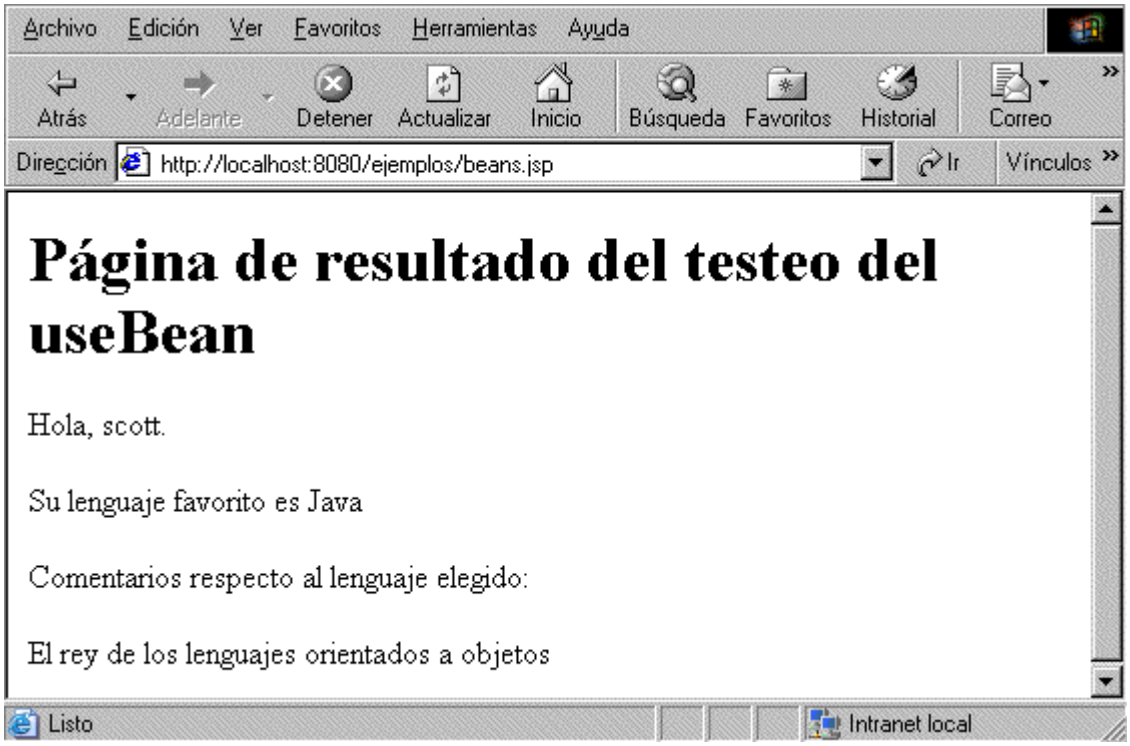
La página jsp es muy sencilla puesto que como vamos a ver a continuación, todo tratamiento Java se hace dentro del bean (LenguajeBean.java).

**`LenguajeBean.java`** :

```
public class LenguajeBean {

  private String nombre;
  private String lenguaje;

  public LenguajeBean () {}

  public void setNombre (String nombre) {
    this.nombre = nombre;
  }

  public String getNombre () {
    return nombre;
  }

  public void setLenguaje (String lenguaje) {
    this. lenguaje = lenguaje;
  }

  public String getLenguaje () {
    return lenguaje;
  }

  public String getLenguajeComentarios () {
    if(lenguaje.equals("Java")) {
      return "El rey de los lenguajes orientados a objetos";
    } else if(lenguaje.equals("C++")) {
      return "Demasiado complejo para algunos desarrollos";
    }else if(lenguaje.equals("Perl")) {
      return "Está bien si te gusta el código incomprensible";
    }else {
      return "Lo siento. Lenguaje desconocido";
    }
  }
}
```

El resultado sería el siguiente:






## <jsp:include>

| Atributo | Descripción |
|---|---|
| filename | El recurso a incluir. |
| flush | En JSP 1.1 este valor debe ser "`true`" porque el valor "`false`" no es soportado en esta versión. Si el valor es "`true`", el buffer en el `stream` de salida es xxx... |

**Ejemplo**: includeAccion.jsp Se mostrará un ejemplo de cómo utilizar tanto la acción como la directiva para ver la diferencia:

```
<html>
  <head>
    <title>Página de test para la acción include </title>
  </head>
  <body>
    <h1> Página de test para la acción include </h1>

    <h2>USANDO LA DIRECTIVA include </h2>
    <%@ include file="include.html" %>
    <%@ include file="include2.jsp" %>
<br>
<%
   java.util.Calendar gTime = java.util.Calendar.getInstance();
   gTime.setTime(new java.util.Date());
   String pagina = "include"+gTime.get(gTime.DAY_OF_WEEK)+".jsp";

   // la acción jsp:include no puede compilar la página que incluye
   // hasta el momento mismo de la ejecución,
   // puesto que no sabe ni siquiera qué página será incluida

   // en este caso, cada día se incluirá una página distinta
   // desde include1.jsp (domingo) hasta include7.jsp (sábado)
%>

    <h2>USANDO LA ACCIÓN include </h2>
    <jsp:include page="include.html" flush="true" />
    <jsp:include page="<%= pagina %>" flush="true" />


  </body>
</html>
```

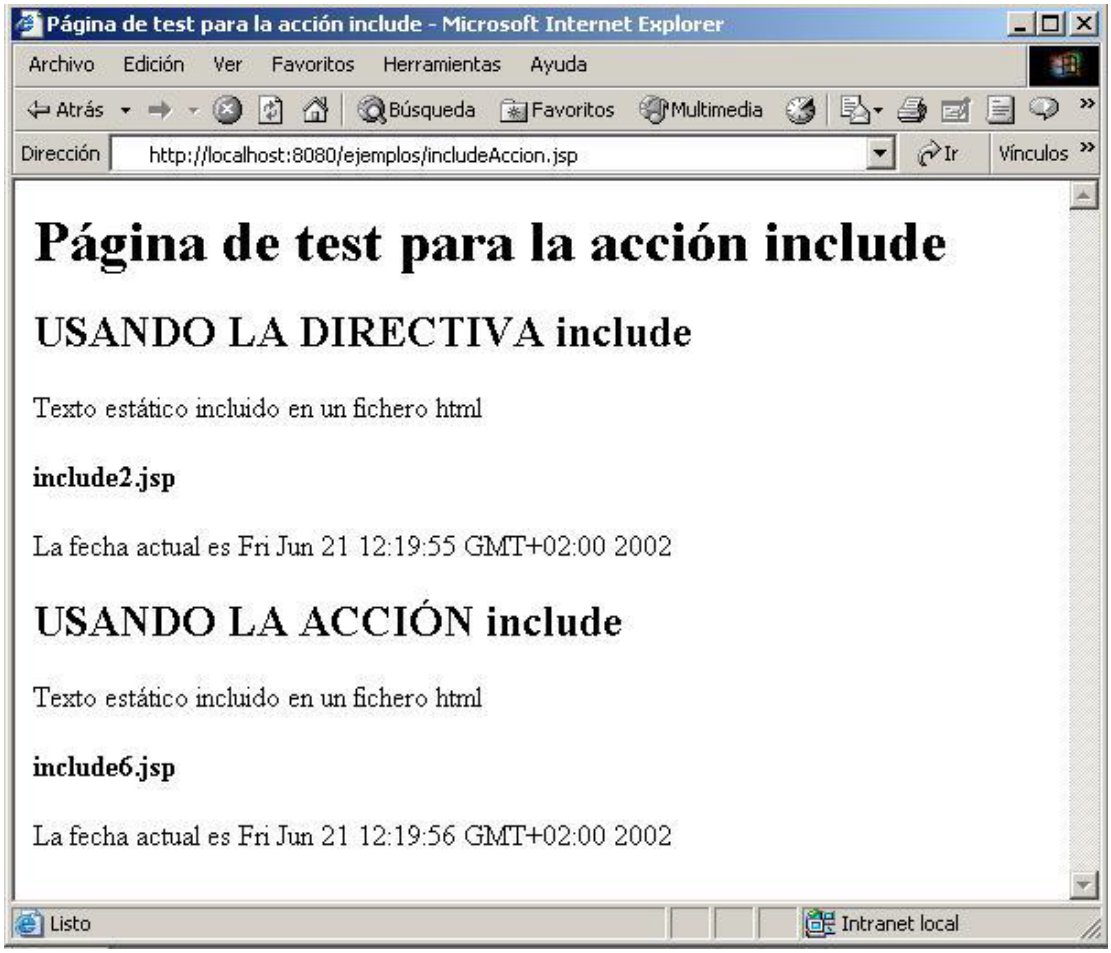
**include.html:**

```
<p> Texto estático incluido en un fichero html </p>
```

**includeX.jsp:**

```
<h4>includeX.jsp</h4>
<%@ page import="java.util.Date" %>
<%= "La fecha actual es " + new Date() %>
```

El resultado sería el siguiente:




## <jsp:forward>

El siguiente ejemplo muestra una pantalla de login que recoge los datos de nombre de usuario y password. Si estos datos son correctos se redirecciona a una página de bienvenida y si no se vuelve a cargar la misma página.

**Ejemplo**: `forward.html`

```
<html>
  <head>
    <title>Página de test para la acción forward </title>
  </head>
  <body>
    <h1> Página de test para la acción forward </h1>

    <form method="post" action="forward.jsp">
    <p>Por favor, introduzca su nombre:
    <input type="text" name="nombre">
    <br> y password:
    <input type="password" name="password"> </p>
    <p><input type="submit" value="Log in">
    </form>
  </body>
</html>
```

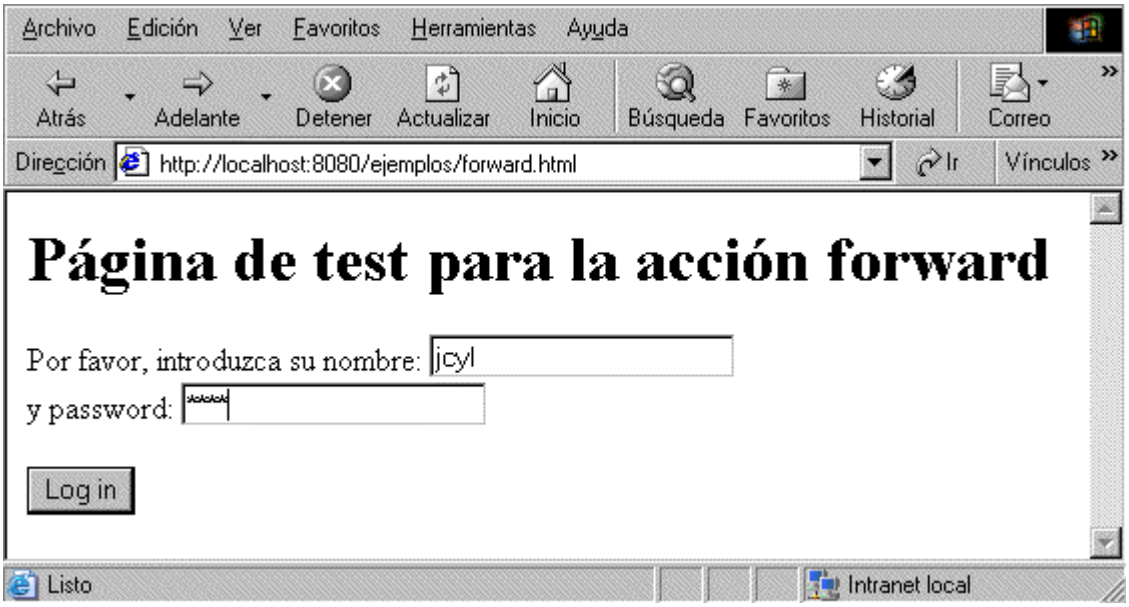
**forward.jsp:**

```
<%
  if((request.getParameter("nombre").equals("jcyl")) &&
     (request.getParameter("password").equals("jcyl"))) {
%>

<jsp:forward page="forward2.jsp" />

<% } else { %>

<%@include file="forward.html" %>

<% } %>
```

**forward2.jsp:**

```
<html>
  <head>
    <title> ¡Login correcto! </title>
  </head>
  <body>
   <h1> ¡Login correcto! </h1>
   <p>Bienvenido, <%= request.getParameter("nombre") %></p>

  </body>
</html>
```

El resultado sería el siguiente:




Archivo Edición Ver Favoritos Herramientas Ayuda
Atrás Adelante Detener Actualizar Inicio Búsqueda Favoritos Historial Correo
Dirección http://localhost:8080/ejemplos/forward.jsp
Ir Vínculos
¡Login correcto!
Bienvenido, jcyl
Listo
Intranet local

## <jsp:plugin>

**Ejemplo**:

```
<body bgcolor="#FFFFFF">
<center>
<hr><br>
<h1>
<font size="+3" color="#CC0066">Duke's</font>
<img src="duke.books.gif">
<font size="+3" color="black">Bookstore</font>
</h1>
<jsp:plugin type="applet" code="data.DigitalClock.class"
codebase="." jreversion="1.3" align="center" height="25"
width="300"
nspluginurl="http://java.sun.com/products/plugin/1.3.0_01/plugin-
install.html"
iepluginurl="http://java.sun.com/products/plugin/1.3.0_01/jinstall-
130_01-win32.cab#Version=1,3,0,1" >
<jsp:params>
<jsp:param name="language"
value="<%=request.getLocale().getLanguage()%>" />
<jsp:param name="country"
value="<%=request.getLocale().getCountry()%>" />
<jsp:param name="bgcolor" value="FFFFFF" />
<jsp:param name="fgcolor" value="CC0066" />
</jsp:params>
<jsp:fallback>
<p>Unable to start plugin.</p>
</jsp:fallback>
</jsp:plugin>
</center>
<br>
<hr>
<br>
```

# Arquitectura Request Controller

El diseño JSP se complica cuando aparece la necesidad conjunta de procesar el request y mostrar los datos. Estas tareas pueden ser muy diferentes para la mayoría de las páginas y el procesamiento del request puede llegar a ser complicado. Además, existen otros problemas añadidos, por ejemplo, algunas JSPs necesitan manipular el objeto session o application y otras necesitarán examinar los valores del request para ver hacia donde tienen que redireccionar. Y todo esto no se soluciona simplemente mapeando las propiedades del request en un bean, especialmente si se necesita redireccionamiento.

Por lo tanto se propone una solución mucho más elegante que es lo que se conoce como "`Arquitectura Request Controller`". Consiste en tener una JSP o servlet como un único punto de entrada para un grupo de páginas JSP. Este punto de entrada no genera una salida por sí misma sino que procesará el request, opcionalmente manipulará la sesión o el entorno de aplicación, y redireccionará a la página JSP apropiada. Las aplicaciones complejas estarán formadas por grupos de controladores.

Las implementaciones de los controladores normalmente incluyen un parámetro que se suele nombrar como `action`. El valor de este parámetro es examinado por el controlador para decidir cómo procesar el request.

Esta arquitectura se deberá apoyar de clases Java que contengan la lógica, nunca deberá estar esta lógica en el controlador JSP.

Por lo tanto la arquitectura se basará en tres puntos:

- Controlador JSP. Será una jsp sencilla que instanciará la clase que procesa el request que usará para encontrar la vista JSP a la que tiene que redireccionar. Como ya hemos visto en las normas de codificación hay una regla de nombrado para los controladores JSP que es `xxxxController.jsp`, donde `xxxx` es un nombre que identifique el papel que desempeña este controlador dentro de la aplicación web.

- Clase Java para el procesamiento del request, opcionalmente otras clases de ayuda a ésta. Esta clase como mínimo tendrá que decidir a qué página JSP redireccionar dependiendo del valor del request. Además, en muchos casos, antes además de decidir la vista JSP a la que redireccionar, se necesitará tratamiento del request. . Como ya hemos visto en las normas de codificación hay una regla de nombrado para estas clases que es `xxxRequestController` ó `xxxxRequestProcessor`.

- Múltiples vistas JSP. Estas páginas JSP serán muy sencillas y no necesitarán ningún procesamiento del request.

**Ejemplo**: `jcylController.jsp`

```
<%--
  Ejemplo de un controlador jsp.
  Esta página no genera ninguna salida, simplemente
  redireccionará cada request a la página JSP apropiada.
  La lógica para la toma de decisión es suministrada por
  el bean de sesión jcylRequestController.
--%>

<%@page session="true" errorPage="error.jsp" %>


<jsp:useBean id="controller" scope="session"
             class="jcyl.jcylRequestController" />


<%--
  Cada petición a esta página se redireccionará a la vista jsp
  que  determine el controlador. Esta clase deberá hacer un set
  del estado de la sesión antes de devolver la url de la página
  a la que se va a redireccionar.
--%>

<jsp:forward page="<%=controller.getNextPage(pageContext, request)%>" />
```

**`jcylRequestController.java`**

Se necesitará pasar unos parámetros al `RequestController` para que accediendo al request sea capaz de saber lo que tiene que hacer. Por ejemplo, el método getNextPage podría ser algo así:

```
public String getNextPage(PageContext pageContext,
           HttpServletRequest request)
                        throws BrowseException {

  String action = request.getParameter("action");
  // Si no hay ninguna acción definida, el usuario
  // deberá volver a la página de log in
  if(action == null || acion.equals("")) {
    return LOGIN_PAGE;
  } else {
    // Hacer lo que sea dependiendo del valor de la acción
  }
}
```

# Taglibs

Se muestra el ejemplo de un tag simple, es decir sin atributos ni cuerpo, que tenga como salida HTML. Se añadirá también algún tipo de contenido dinámico para probar que el tag es dinámico. Lo que se quiere conseguir es la siguiente salida html:

```
Junta de Castilla y León.
HOLA. Mi nombre es <tag handler class> y ahora mismo son las
<fecha y hora>
```

Llamaremos al tag `hola` y a continuación se mostrará cómo se debe usar este tag en una página jsp:

**hola.jsp:**

```
<%@ taglib uri="/hola" prefix="jcyl" %>

<html>
  <head>
    <title>Ejemplo de un tag personalizado </title>
  </head>
  <body>

    <p>Esta es una salida con contenido estático. </p>
    <i>
      <jcyl:hola></jcyl:hola>
    </i>
    <p>Esta es otra salida con contenido estático. </p>

  </body>
</html>
```

Lo que se tendrá que implementar es el controlador del tag (`HolaTag`) y crear el descriptor de librería de etiqueta.

Los dos métodos importantes a observar en el controlador del tag son `doStartTag` y `doEndTag`. El primero es invocado cuando se encuentra la etiqueta de inicio y el segundo será invocado cuando se encuentre la etiqueta de fin. Es importante señalar que el tag puede acceder al objeto `PageContext` de cada JSP que lo utiliza.

**HolaTag.jsp:**

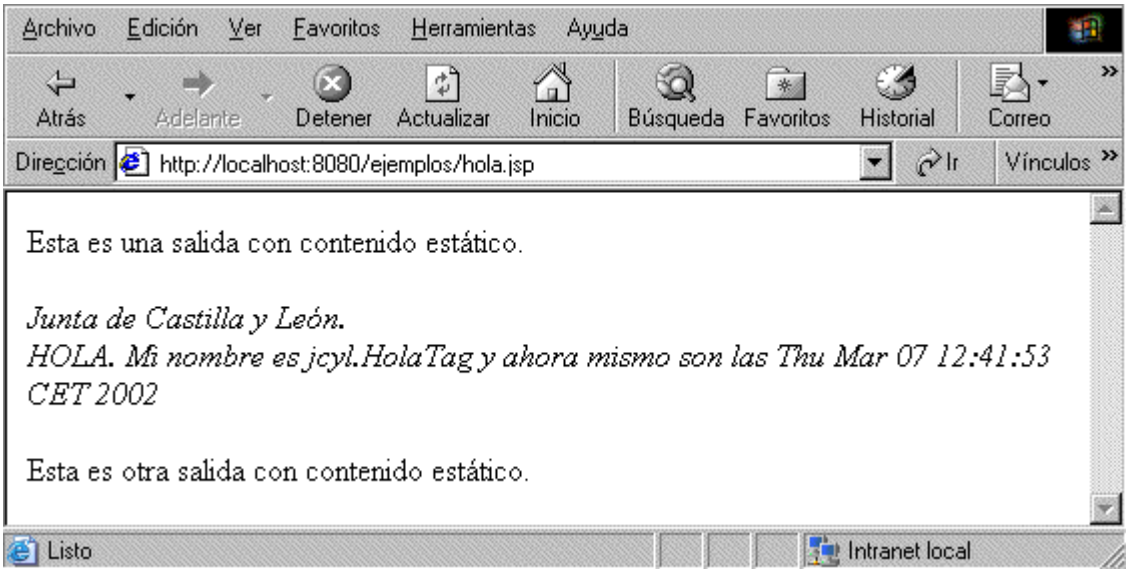
```
package jcyl;

import java.io.IOException;
import java.util.Date;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HolaTag extends TagSupport {

  public int doStartTag() throws JspTagException {
    // Este valor de retorno significa que el motor JSP deberá
    // evaluar el contenido y cualquier tag hijo de este tag
    return EVAL_BODY_INCLUDE;
  }

  public int doEndTag() throws JspTagException {
    String dateString = new Date().toString();
    try {
      pageContext.getOut().write("Junta de Castilla y
                                  León.<br/>");
      pageContext.getOut().write("HOLA. Mi nombre es " +
                                  getClass().getName() +
                                  " y ahora mismo son las " +
                                  dateString + "<p/>");

    } catch (IOException ex) {
      throw new JspTagException
            ("Error. EL tag hola no ha podido escribir la salida");
    }

    // Este valor de retorno  significa que el motor JSP
    // deberá continuar evaluando el resto de la página
    return EVAL_PAGE;
  }

}
```

El descriptor de librería de etiqueta se llamará `hola.tld`. Se tratará de un documento xml conforme a un DTD que viene especificado por Sun Microsystems.

**hola.tld:**

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>jcyl</shortname>

  <info>Ejemplo de una librería de tags</info>

  <tag>
    <name>hola</name>
    <tagclass>jcyl.HolaTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>Un ejemplo simple</info>
  </tag>
</taglib>
```

El resultado sería el siguiente:




El ejemplo se ha desplegado de la siguiente manera:

```
hola.jsp
WEB-INF/
        web.xml
        classes/
                jcyl/
                     HolaTag.class
                tlds/
                     hola.tld
```

Y el **`web.xml`** se ha de modificar de la siguiente manera:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>

  <display-name>jcyl</display-name>
  <description>Ejemplo de Tag Extensions</description>

  <session_config>
    <session-timeout>0</session-timeout>
  </session_config>

  <taglib>
    <taglib-uri>/hola</taglib-uri>
    <taglib-location>/WEB-INF/tlds/hola.tld</taglib-location>
  </taglib>

</web-app>
```

La mejor forma de desplegar el ejemplo en un servidor tomcat sería mediante la creación de un fichero war de la siguiente manera:

```
jar –cvf hola.war WEB-INF/classes/jcyl/*.class WEB-
INF/tlds/hola.tld WEB-INF/web.xml *.jsp
```