Curso de Java Server Pages Nivel Avanzado

Manual del alumno

<JSP>

Detodoprogramación.com

Índice

1 Introducción al curso

1.1 Objetivo de este curso

En este curso vamos a aprender el lenguaje JSP que nos permitirá crear páginas web dinámicas.

1.2 Manual del alumno

Este manual del alumno es una ayuda para el alumno, para tenga un recuerdo del curso. Este manual contiene un resumen de las materias que se van a estudiar durante el curso, pero el alumno debería de tomar notas personales para completas este manual.

1.3 Ejercicios prácticos

Para captar mejor la teoría, se harán muchos ejercicios con los alumnos, para probar la teoría y verificar la integración de la materia.

También, el alumno podrá copiar sus códigos en un disquete al fin del curso para llevarse, con fin de seguir la práctica en su hogar.

1.4 Requisitos para atender a este curso

Una iniciación al lenguaje Java y el JSP es requerida para seguir este curso. La creación y el manejo de objetos Java así como el JSP básico están considerada cómo asimilado antes de empezar este curso.

Si el alumno tiene dificultades en un u otro capitulo, el debe sentirse libre de pedir explicaciones adicionales al profesor.

Pero si aparece que el alumno no posee los requisitos mínimos para este curso, por respeto a los otros alumnos que ya poseen esta materia, el alumno podría ser traslado para otro curso en el futuro, cuando el cumplirá con los requisitos.

2 Acciones personalizadas

Las acciones personalizadas permiten encapsular la lógica y ponerla a disposición del diseñador de página en un formato familiar.

Vamos a ver ejemplos que permiten acceder a la base de datos, codificar URL, incluir contenido personalizado a las opciones locales del sistema del clientes, etc... utilizando etiquetas personalizadas.

Las acciones personalizadas parecen mucho a los JavaBeans que veremos más tarde, pero a la diferencia de ellos, la acción personalizada tiene conocimiento del ambiente en el cual esta utilizada.

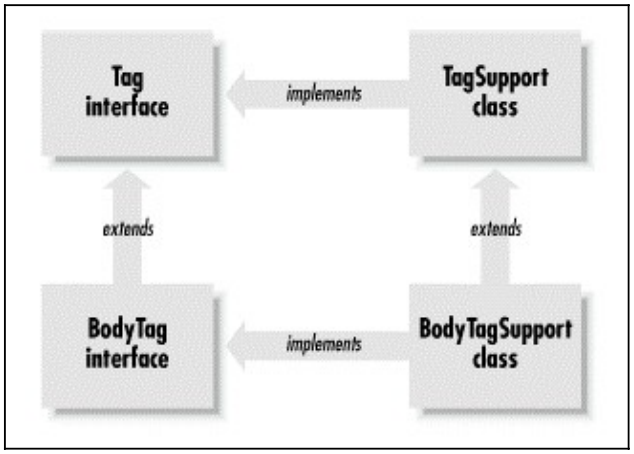
Como el JavaBean, la acción personalizada está contenida en una clase Java.

2.1.1 Introducción a la etiqueta de extensión

La acción personalizada, que es en realidad una clase de manejo de etiqueta para una acción personalizada, es básicamente una clase Java con método permitiendo de manejar sus propiedades, correspondiendo a los atributos de la acción personalizada. La clase de manejo de etiqueta tiene que implementar uno o dos interfaces definidos en las especificaciones del JSP.

Todas las clases e interfaces de acciones personalizadas deben de implementar un manejo de etiqueta definido en el paquete `javax.servlet.jsp.tagext`. Los dos interfaces primarios se llaman `Tag` y `BodyTag`. El interfaz `Tag` define los métodos que necesitamos para cualquier acción. El interfaz `BodyTag` extiende el interfaz `Tag` y adjunta métodos utilizadas para acceder al cuerpo de un elemento de acción.

Para facilitar el desarrollo de manejo de etiquetas, dos clases de soporte han sido definidas por el API: `TagSupport` y `BodyTagSupport`. Estas clases proveen una implementación por defecto para los métodos del interfaz correspondiente.



Una librería de etiquetas es una colección de acciones personalizadas. Al lado de las clases de manejo de etiquetas, la librería de etiqueta debe contener un descriptor de librería de etiquetas (`Tag Library Descriptor`, o `TLD`). Este `TLD` es un archivo `XML` que traduce los nombres de acciones personalizadas a las clase de manejo de etiquetas que le corresponde, y describe los atributos soportados por cada acción personalizada. Por facilidad, se pueden meter las clases y el `TLD` en un archivo `JAR`, para facilitar su instalación.

2.1.2 Primera acción personalizada

Vamos a crear nuestra primera acción personalizada, con el entorno `JBoss IDE`.

Primero, creamos una nueva clase de manejo de etiqueta. Por eso, hacemos un clic derecho sobre la carpeta curso, y elegimos New...Other, y debajo de JBoss-IDE...Webcontent, el JSP Tag Handler. Lo creamos en el paquete `tag`, con el nombre HolaTag y el contenido siguiente:

```
package tag;
import javax.servlet.jsp.tagext.TagSupport;
import java.io.*;
public class HolaTag extends TagSupport {
	private String nombre = "Mundo";
	public void setNombre(String miNombre){
		this.nombre=miNombre;
	}
public int doStartTag(){
			System.out.println("Hola1 "+this.nombre);
			return SKIP_BODY;
	}	public int doEndTag(){
		try{
			pageContext.getOut().println("Hola "+this.nombre);
			System.out.println("Hola2 "+this.nombre);
		}
		catch (IOException e) {} // Ignoralo
		return EVAL_PAGE;
	}
}
```

Luego, debajo de la carpeta WEB-INF creamos una nueva carpeta 'tlds'. Y a dentro, creamos un nuevo `Tag Library Descriptor` (debajo JBoss-IDE...Descriptor), que llamamos miTagLib.tld.

Editamos el contenido de miTagLib y lo cambiamos por:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib
	 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
	"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
	<tlib-version>1.2</tlib-version>
	<jsp-version>2.0</jsp-version>
	<short-name>miTest</short-name>
	<description>Etiqueta de prueba</description>
	<tag>
		<name>hola</name>
		<tag-class>tag.HolaTag</tag-class>
		<body-content>empty</body-content>
		<attribute>
			<name>nombre</name>
		</attribute>
	</tag>
</taglib>
```

Por fin, creamos tagSimple.jsp para que utiliza nuestra acción personalizada:

```
<%@ taglib uri="/WEB-INF/tlds/miTagLib.tld" prefix="miTest" %>
<html>
<head>
<title>Prueba de Taglig</title>
</head>
<body bgcolor="#FFFFFF">
<h1><miTest:hola nombre="Cedric" /></h1>
</body>
</html>
```

Y miramos el resultado en http://localhost:8080/curso/tagSimple.jsp

2.2 Acciones sin cuerpo

Para llamar a una acción sin cuerpo, se utiliza una clase que extiende la clase `TagSupport`, y la etiqueta siguiente: `<prefijo:nombre_accion atributo1="valor1" atributo2="valor2" ... />`

El prefijo es el prefijo declarado en la directiva de página `Taglib`.

El nombre de la acción corresponde al nombre de la acción declarado en la librería, y que refiere a una clase Java.

Los atributos son valores que se pueden atribuir a variables de la clase, utilizando los métodos `setAtributo1 ("valor1")`, `setAtributo2 ("valor2")`, etc... definidas en la clase.

La secuencia de ejecución es la siguiente:

```
<prefijo:nombre_accion
atributo1="valor1" ----------→ setAtributo1("valor1")
atributo2="valor2" ----------→ setAtributo2("valor2")
/>                 ----------→ doStartTag() + doEndTag()
```

Se puede también definir así:

```
<prefijo:nombre_accion
atributo1="valor1"            ----------→ setAtributo1("valor1")
atributo2="valor2"            ----------→ setAtributo2("valor2")
>                             ----------→ doStartTag()
Mi cuerpo de etiqueta
</prefijo:nombre_accion>      ----------→ doEndTag()
```

En este ejemplo, como estamos utilizando una clase extendiendo `TagSupport`, por defecto el método `doStarTag()` regresa `SKIP_BODY`, que significa que no utiliza el cuerpo de la etiqueta (aquí 'Mi cuerpo de etiqueta'). Las otras valores para el `return` son `EVAL_BODY_INCLUDE` y `EVAL_BODY_BUFFERED`.

Afuera de los métodos de inicialización de atributos (`setAtributo1`,...), de `doStarttag()` y `doEndTag()`, existen por lo menos dos otros métodos importante: `public void setPageContext(PageContext pageContext)` y `public void release()`. El método `setPageContext` permite de acceder a la consulta y respuesta de la página, así como a las variables JSP de la página.

Ejemplo:

```
package tag;
import javax.servlet.jsp.tagext.TagSupport;
import javax.servlet.jsp.*;
import java.io.*;
public class TagSinCuerpo extends TagSupport {
        private String context = "Mundo";
        protected PageContext pageContext;
        public void setPageContext(PageContext pageContext){
                this.pageContext=pageContext;
                context=(String) pageContext.getRequest().getParameter("test");
        }
        public int doEndTag(){
                try{
                        pageContext.getOut().println("Parámetro = "+this.context);
                        System.out.println("Parámetro = "+this.context);
                }
                catch (IOException e) {} // Ignoralo
                return EVAL_PAGE;}
}
```

Adjuntar en miTagLib.tld antes </taglib>:

```
<tag>
        <name>sinCuerpo</name>
        <tag-class> tag.TagSinCuerpo</tag-class>
</tag>
```

tagSinCuerpo.jsp:

```
<%@ taglib uri="/WEB-INF/tlds/miTagLib.tld" prefix="miTest" %>
<html>
<head>
<title>Prueba de Taglig</title>
</head>
<body bgcolor="#FFFFFF">
<h1><miTest:sinCuerpo /></h1>
</body>
</html>
```

Llamando a http://localhost:8080/curso/tagSinCuerpo.jsp?test=prueba produce en la consola y la página JSP: `Parámetro = prueba`

Para ahorrar la memoria, se debería de implementar el método `public void release()`, para quitar todas las referencias a objetos que fueron puestas: Por eso, se les asigna el valor nulo y se llama al método `release`() del objeto mayor (`super`). Este método será llamada cuando el manejo de etiquetas no será más necesario (por ejemplo, al recargo del contexto).

Ejemplo:

```
public void release(){
                nombre=null;
                context=null;
                pageContext=null;
                super.release();
                System.out.println("Liberado");
        }
```

2.3 Atributos de tipo diferente a String y otros parámetros del atributo

Además del nombre del parámetro, se puede también definir si el parámetro es requerido o no (por defecto no lo es), si permite un valor a evaluar en tiempo real (= scriplet) o no (por defecto no lo es), y el tipo de datos que accepta (por defecto es java.lang.String.

Ejemplo de un parámetro llamado “precio”, que es requerido (no se puede usar la acción sin definirlo), permite un valor “dinámica”, y es de tipo entero.

```
<attribute>
          <name>precio</name>
          <required>false</required>
          <rtexprvalue>true</rtexprvalue>
          <type>java.lang.Integer</type>
</attribute>
```

Ejemplo:

```
package tag;
import javax.servlet.jsp.tagext.TagSupport;
import java.io.*;

public class TagSuma extends TagSupport {

        private static final long serialVersionUID = 1L;
        private Integer numero1 = 1;
        private Integer numero2 = 2;
        public void setNumero1(Integer numero1){
                this.numero1=numero1;
        }
        public void setNumero2(Integer numero2){
                this.numero2=numero2;
        }
        @Override
        public int doEndTag(){
                try{
                        int resultado=0;
```



```
                        resultado=numero1+numero2;
                        System.out.println(resultado);
                        pageContext.getOut().println("Resultado "+resultado);
                }
                catch (IOException e) {} // Ignoralo
                return EVAL_PAGE;
        }
        public Integer getNumero1() {
                return numero1;
        }
        public Integer getNumero2() {
                return numero2;
        }
}
```

Modificación en el TLD:

```
<tag>
        <name>suma</name>
        <tag-class> tag.TagSuma</tag-class>
         <attribute>
          <name>numero1</name>
          <required>true</required>
          <rtexprvalue>true</rtexprvalue>
            <type>java.lang.Integer</type>
       </attribute>
       <attribute>
          <name>numero2</name>
          <required>true</required>
          <rtexprvalue>true</rtexprvalue>
            <type>java.lang.Integer</type>
       </attribute>
</tag>
```

tagSuma.jsp:

```
<%@ taglib uri="/WEB-INF/tlds/miTagLib.tld" prefix="miTest" %>
<html>
<head>
<title>Prueba de Taglig</title>
</head>
<body bgcolor="#FFFFFF">
<h1><miTest:suma numero1="1" numero2="5" /></h1>
</body>
</html>
```

2.4 Acciones con cuerpo

Para llamar a una acción con cuerpo, se utiliza una clase que extiende la clase `BodyTagSupport`, y la etiqueta siguiente: `<prefijo:nombre_accion atributo1=”valor1” atributo2=”valor2” ... >`

El uso y el funcionamiento es parecido a el de las acciones sin cuerpo, con tres métodos más: `setBodyContent(), doInitBody(), e doAfterBody().`

Con el cuerpo, se debe utilizar una etiqueta de inicio y una de fin.

La secuencia de ejecución es la siguiente:

```
<prefijo:nombre_accion
atributo1=”valor1”          ----------→ setAtributo1(“valor1”)
atributo2=”valor2”          ----------→ setAtributo2(“valor2”)
>                           ----------→ doStartTag()
                            ----------→ setBodyContent() + doInitBody()
Mi cuerpo de etiqueta
                            ----------→ doAfterBody()
</prefijo:nombre_accion>    ----------→ doEndTag()
```

En este ejemplo, como estamos utilizando una clase extendiendo `BodyTagSupport`, por defecto el método `doStarTag()` sobre escribe el método de `TagSupport` para que regresa `EVAL_BODY_INCLUDE`, lo que significa que

se utiliza el cuerpo de la etiqueta (aquí 'Mi cuerpo de etiqueta'). Las otras valores para el `return` son `SKIP_BODY y EVAL_BODY_BUFFERED`.

El método `setBodyContent()` guarda solamente el contenido del cuerpo en una variable.

Ejemplo:
```
protected BodyContent bodyContent;
...
public void setBodyContent(BodyContent body) {
this.bodyContent = body;}
```

El método `doInitBody()` se deja normalmente vacía. `public void doInitBody( ) throws JspException {}`

El método `doAfterBody()` es el que utilizamos para manejar el contenido del cuerpo, si es necesario.

Si no se necesita una bucle, se utiliza `return SKIP_BODY`, si no `return EVAL_BODY_AGAIN`. Cuidado que el código debe siempre tener una posibilidad de llegar al `SKIP_BODY`, si no, entrara en una bucle infinita.

Ejemplo:
Clase CifraHTML:
```
package tag;
import javax.servlet.jsp.tagext.BodyTagSupport;
import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class CifraHTML extends BodyTagSupport {
        static final long serialVersionUID =2;
        int count =0;
        String output="";
        public int doStartTag(){
                count =0;
                output="";
                return EVAL_BODY_BUFFERED;
        }
        public int doAfterBody( ) throws JspException {
                BodyContent bc = getBodyContent();
                JspWriter out = getPreviousOut();
                count++;
                if (count>3) {
                        try {
                        output+="<p>Ronda "+count+"</p>";
                        out.write(output);
                        return SKIP_BODY;
                        }
                        catch (IOException e) {return SKIP_BODY;} // Ignore
                } else {
                        if (count==1){
                        String miCuerpo=bc.getString();
                        miCuerpo=miCuerpo.replaceAll("&","&amp;");
                        miCuerpo=miCuerpo.replaceAll("<","&lt;");
                        miCuerpo=miCuerpo.replaceAll(">","&gt;");
                        miCuerpo=miCuerpo.replaceAll("\"","&#34;");
                        miCuerpo=miCuerpo.replaceAll("'","&#39;");
                        output+=miCuerpo+"<p>Ronda "+count+"</p>";
                        } else output+="<p>Ronda "+count+"</p>";
                return EVAL_BODY_AGAIN ;
                }
        }
}

miTagLib.tld:
<tag>
        <name>CifraHTML</name>
        <tag-class> tag.CifraHTML</tag-class>
</tag>

tagCifraHTML.jsp
<%@ taglib uri="/WEB-INF/tlds/miTagLib.tld" prefix="miTest" %>
<html>
```

```
<head>
<title>Prueba de Taglig</title>
</head>
<body bgcolor="#FFFFFF">
<h1><miTest:CifraHTML>
Hola "mis queridos" <alumnos> & 'amigos'.
</miTest:CifraHTML></h1>
<hr>
</body>
</html>
```


3 Utilización de COOKIES

3.1 ¿Qué son los COOKIES?

Los COOKIES son variable que se guardan en pequeños archivos de texto en la computadora del cliente y que permiten guardar ciertas informaciones el cliente. Eso permite por ejemplo guardar el nombre del cliente para recuperarlo la próxima vez que el cliente se conecta.

Al contrario de la variable de sesión que se borran al terminar la sesión, se puede definir el tiempo que el COOKIE esta válido. Por defecto, está valido solamente por la sesión corriente (como las variables de sesión), pero se puede cambiar la valor de su tiempo de vencimiento para poder recuperarlo más tarde, en la próxima conexión.

3.2 Creación de un COOKIE

El código de creación de un COOKIE tiene que ir de primero, antes la etiqueta `<HTML>` y de cualquier otro código JSP (directivas,...).

Después de haber creado el COOKIE, hay que declarar por las páginas de cual carpeta el COOKIE está disponible. También se puede declarar por cual dominio está disponible.

Por fin hay que enviar el COOKIE a la computadora del cliente.

Una vez creado, el COOKIE puede ser leído por las páginas JSP. Todos los COOKIES de un mismo sitio web están guardados juntos en un archivo texto en el cliente.

La sintaxis de creación de un COOKIE es la siguiente:

```
<%
Cookie nombreDeMiCookie = new Cookie("nombreDeMiVariable","valorDeMiVariable");
nombreDeMiCookie.setPath("/cursoAvanzado");
nombreDeMiCookie.setMaxAge(3600); // una hora, o 3600 segundos
response.addCookie(nombreDeMiCookie);
%>
```

3.3 Recuperación de información de un COOKIE

Para poder leer un COOKIE, este debe existir. Para leerlo, se utiliza el método `getCookies()` que regresa un arreglo de objeto `Cookie`. Una vez recuperada, se puede sacar el nombre de cada variable del COOKIE y su valor con los métodos `getName()` y `getValue()`. Hay que recorrer todo los COOKIES del arreglo para buscar el que nos interesa.

Por defecto el arreglo de COOKIES siempre tiene el COOKIE llamado JSESSIONID con el valor de la sesión. Así que el arreglo siempre tiene por lo menos un valor.

Ejemplo:

```
<%
Cookie [] miCookie = request.getCookies();
String nombreUsuario="",
for (int x=0;x<miCookie.length;x++)
{
out.print(miCookie[x].getName()+" tiene el valor siguiente: "+ miCookie[x].getValue()+"<br>");
// Buscar Cookie nombreUsuario
if (miCookie[x].getName().equals("nombreUsuario") nombreUsuario= miCookie[x].getValue();
}
%>
```

3.4 Borrado de un COOKIE

Para borrar un COOKIE, hay que seguir la misma sintaxis que para crearlo, solo que el valor del `setMaxAge` igual a 0, así que el COOKIE expiará directamente.

La sintaxis de borrado de un COOKIE es la siguiente:

```
<%
Cookie nombreDeMiCookie = new Cookie("nombreDeMiVariable","");
nombreDeMiCookie.setPath("/cursoAvanzado");
nombreDeMiCookie.setMaxAge(0); // se vence ahora
response.addCookie(nombreDeMiCookie);
%>
```


4 RequestDispacher

4.1 ¿Qué son los RequestDispacher?

El objecto RequestDispacher permite incluir dentro de un servlet el contenido que viene de otra página o código, o reenviar la consulta a otra página.

Esto es parecido al tag <jsp:include /> o un <jsp:forward />.

Ejemplo:

```
...
protected RequestDispatcher rd = null;
...
ServletContext context = this.getServletContext();
rd = context.getRequestDispatcher("/login.jsp");
rd.include(request, response);
rd = context.getRequestDispatcher("/test.txt");
rd.include(request, response);
...
```


5 Autenticación del usuario

5.1 Autenticación proveída por el contenedor

En contenedor en el cual se ejecuta la página JSP provee un mecanismo de autenticación integrado. Le implementación puede variar de un tipo de servidor a otro. Este mecanismo de autenticación esta bien probado y fiable, así que es un mecanismo de primera elección.

5.1.1 Métodos de autenticación

Existen cuarto tipos de autenticación: autenticación HTTP básica, 'digest', autenticación cliente HTTPS, y autenticación basado en un formulario.

5.1.1.1 Autenticación HTTP básica

Este tipo de autenticación es muy simple, pero no muy seguro, porque utiliza el cifrado Base64, la cual es conocida y se puede descifrar sin muchos esfuerzos por el que intercepta los paquetes de comunicación.

5.1.1.2 Autenticación HTTP 'digest'

Este medio de autenticación es un poco más complicado, pero más seguro. El problema es que no todos los navegadores soportan este tipo de autenticación.

5.1.1.3 Autenticación de cliente HTTPS

Este método de autenticación es el más seguro, porque utiliza certificado de clave publica, y una clave especifica por cada cliente, válida mientras el cliente queda conectado.

5.1.1.4 Autenticación por formulario

Este método le permite personalizar la página de entrada. Pero como los datos están enviados en texto claro, se debería de mezclar con un cifrado SSL.

Por utilizar este método, se requiere dos parámetros: `j_username` y `j_password`. La acción debe ser `j_security_check`.

Ejemplo:
```
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="password" name="j_password">
</form>
```

5.1.2 Control de acceso a los recursos web

Para controlar el acceso a los recursos, se necesita por lo menos dos cosas: tener usuarios definidos, y información registrada sobre el control del acceso a los recursos.

5.1.2.1 Definición de usuarios

La definición de usuario depende del servidor utilizado. Ciertos como IIS pueden utilizar los usuarios del sistema, otros acceden a un servidor externo LDAP. Para Tomcat, los usuarios están definidos en el archivo `tomcat-users.xml` que se encuentra debajo de la carpeta `$TOMCAT_BASE/conf`. La lista de usuarios es para todos los sitios del servidor.

Se pueden definir usuarios, y asignarlos a diferentes grupos.

Ejemplo:
```
<tomcat-users>
<user name="paula" password="boss" roles="admin" />
<user name="hans" password="secret" roles="user" />
</tomcat-users>
```


5.1.2.2 Definición de derechos de accesos

La definición de usuario depende del servidor utilizado. Para Tomcat, los usuarios están definidos en el archivo web.xml que se encuentra debajo de la carpeta WEB-INF del sitio web.

Ejemplo:

```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>admin</web-resource-name>
                <url-pattern>/seguridad/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
                <role-name>admin</role-name>
        </auth-constraint>
  </security-constraint>

 <login-config>
        <auth-method>FORM</auth-method>
        <realm-name>Prueba curso JSP</realm-name>
        <form-login-config>
                <form-login-page>/securityCheck.jsp</form-login-page>
                <form-error-page>/index.jsp</form-error-page>
        </form-login-config>
 </login-config>

 <security-role>
        <description>Administrador</description>
        <role-name>admin</role-name>
 </security-role>
```

El elemento `<security-constraint>` contiene un elemento `<web-resource-collection>` que define los recursos a proteger, y un elemento `<auth-constraint>` que define quien tiene derecho de acceder al recurso protegido.

A dentro del elemento `<web-resource-collection>` está definido el patrón de URL para los recursos protegidos, especificados en el elemento `<url-pattern>`. Así la carpeta con todas las páginas de registro /seguridad/*.

El elemento `<role-name>` a dentro del elemento `<auth-constraint>` menciona que solamente los usuarios del grupo admin. tienen derecho de acceder a este recurso protegido.

Usted puede también definir el tipo de autenticación a utilizar y un nombre asociado a las partes protegidas de la aplicación, conocidas como realm, con el elemento `<login-config>`. El elemento `<auth-method>` acepta los valores siguientes: BASIC, DIGEST, FORM, y CLIENT-CERT, que corresponden a los métodos vistas anteriormente. Cualquier texto puede ser utilizado para el valor del elemento `<realm-name>`. E texto está enseñado como parte del mensaje en la ventana de dialogo que se abre para pedir el nombre y clave del usuario.

Si utilizas la autenticación por formulario, hay que mencionar los nombres del formulario de autenticación y de la página de error en el elemento `<login-config>`.

Ejemplo:

```
<login-config>
        <auth-method>FORM</auth-method>
        <realm-name>Prueba curso JSP</realm-name>
        <form-login-config>
                <form-login-page>/securityCheck.jsp</form-login-page>
                <form-error-page>/securityCheck.jsp?estado=NOK</form-error-page>
        </form-login-config>
</login-config>
```


5.1.3 Recuperación de la información del usuario

A dentro de los scriptlets, se pueden recuperar el nombre de usuario y el grupo del usuario corriente (autenticado) con el objeto `request` y los métodos `request.getRemoteUser( )` y `request.isUserInRole("admin")`.

5.2 Autenticación manejada por la aplicación

Si la autenticación por el mecanismo del contenedor debe ser la primera elección, este mecanismo tiene sus limitaciones al nivel de dinamismo. El manejo de usuario es a veces muy estático porque guardado en un archivo texto (web.xml) en el servidor, y no en una base de datos.

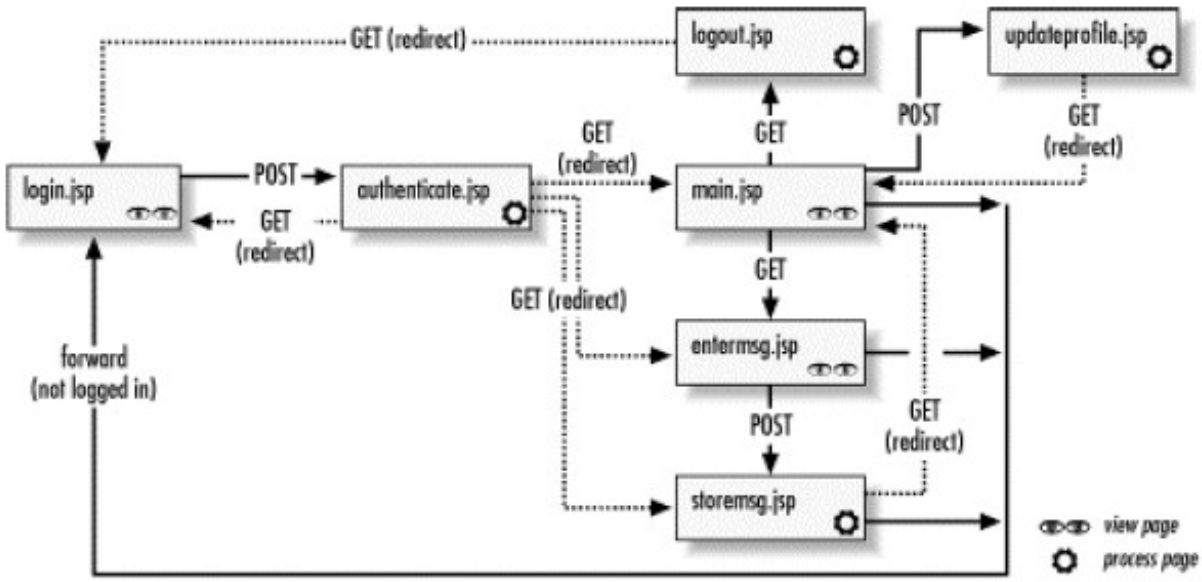
Por esta razón, se puede utilizar un mecanismo de seguridad manejado por la aplicación, que será, por ejemplo, ligado a una base de datos de usuarios.

La implementación de un mecanismo de autenticación de usuario y control de recursos necesita lo siguiente:

1. Registro de usuario
2. Página de autenticación
3. Mecanismo de autenticación, llamado por la página de autenticación
4. Información del usuario guardada al nivel de la sesión, como prueba de que el usuario está autenticado
5. Verificación de la validez de la información de sesión en cada página con acceso restringido.

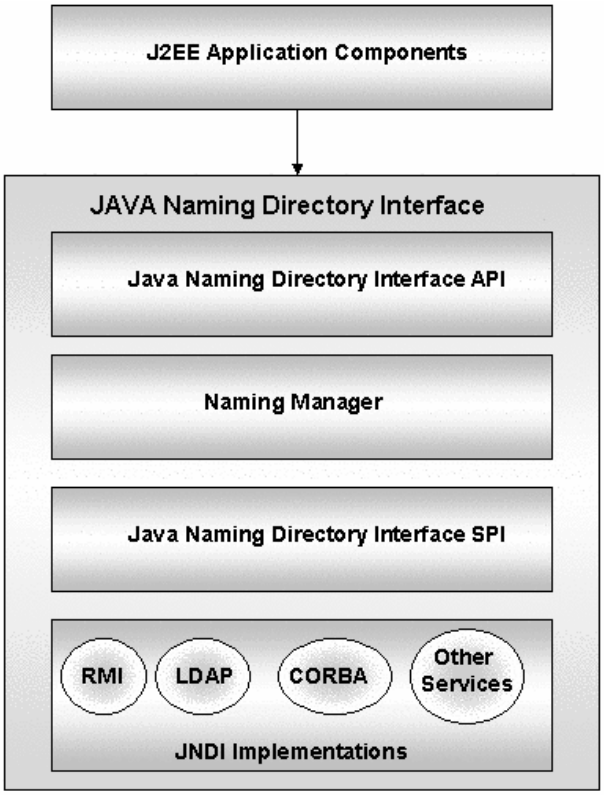
Al momento que se verifica su nombre de usuario y su nombre, se le asigna una o varias variables de sesión. En la páginas protegidas, se verifica si la variable de sesión existe (no es nulo) para este usuario. Si es nulo, es que el usuario probo de llagar a la página con un URL directo, sin autenticarse.

También, si el nombre del usuario es una de las variables de sesión, se puede recuperar de desde cualquiera página JSP después de la autenticación.

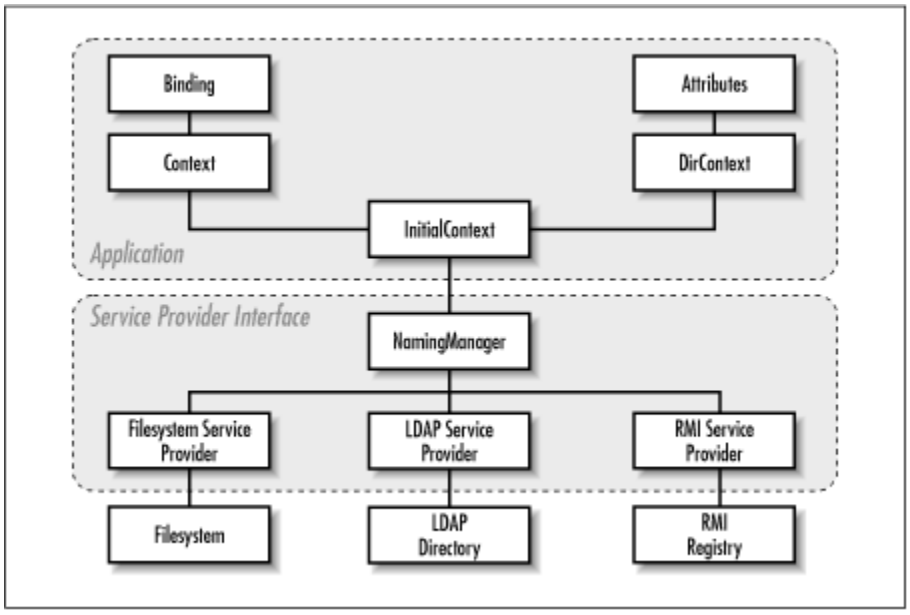


6 Arquitectura Java Naming Directory Interface (JNDI)

J2EE utiliza el API JNDI para acceder genéricamente a servicios de nombrado y directorio utilizando la tecnología Java. El API JNDI reside entre la aplicación y el servicio de nombres y hace que el servicio de nombres subyacente sea transparente para los componentes de la aplicación:



Un cliente puede buscar referencias a componentes EJB u otros recursos en un servicio de nombres como el mencionado arriba. El código del cliente no se modifica, sin importar el servicio de nombres que se esté utilizando o en qué tecnología esté basado, y esto no crea ninguna diferencia en el modo en que los clientes localizan los objetos remotos mediante el API JNDI.



Para que una aplicación pueda interactuar con un servicio de nombre, este debe conocer las propiedades del servicio JNDI al cual el quiere conectarse. Estas propiedades son entre otras el tipo de servicio JNDI (facory), el domicilio IP y el Puerto del servicio.

6.1.1 Ejemplo de JNDI

```
...
       Properties env = new Properties();
       // Definir las propiededes y ubicacion de busqueda de Nombres JNDI.
       env.setProperty("java.naming.factory.initial",  "org.jnp.interfaces.NamingContextFactory");
       env.setProperty("java.naming.provider.url",  "localhost:1099");
       env.setProperty("java.naming.factory.url.pkgs",  "org.jboss.naming");
       try {
              Context initial = new InitialContext(env);
              Object objref = initial.lookup("ejb/EmployeeCMP");
              EmployeeCMPHome home =
                     (EmployeeCMPHome)PortableRemoteObject.narrow(objref,
                                          EmployeeCMPHome.class);
...
```


7 Pool de conexiones

7.1 ¿Qué es un pool de conexiones?

Un pool de conexiones es un conjunto de conexiones que quedqn abiertas y que son compartidas según las necesidades.

El uso de pool de conexiones permite de mejorar las performancias de la aplicación ahorando el tiempo de conexión a la base de datos de cada conexión.

Con los pools de conexiones, se puede definir cuantas conexiones deben quedar abiertas de manera permanente (minimo de conexiones), así como el máximo de conexiones permitidas.

El pool de conexiones reparte las encuestas a la base de datos según las conexiones disponibles. Si el máximo de conexiones está llegado, se creara una fila de encuestas, esperando que se libera una conexión.

7.2 Creación de un pool de conexiones

Los pools de conexiones se crean en el archivo de configuración del servidor. Hay que reanudar el servidor (servicio) para que las modificciones sean efectivas.

7.2.1 JBoss ConnectionPooling

Ejemplo con CursoDS.XML:

```
<?xml version="1.0" encoding="UTF-8"?>

  <!-- ===================================================================== -->
  <!-- New ConnectionManager setup for firebird dbs using jca-jdbc xa driver-->
  <!-- Build jmx-api (build/build.sh all) and view for config documentation -->
  <!-- ===================================================================== -->

<connection-factories>
  <!--FBManager can be used to create and drop databases.
    Drop is especially useful during testing, since it
    assures a clean start next time. -->
  <mbean code="org.firebirdsql.management.FBManager" name="jboss.jca:service=FirebirdManager">
    <attribute name="FileName">/Firebird/data/CURSO.FDB</attribute>
    <attribute name="UserName">sysdba</attribute>
    <attribute name="Password">masterkey</attribute>
    <attribute name="CreateOnStart">false</attribute>
    <attribute name="DropOnStop">false</attribute>
  </mbean>

    <tx-connection-factory>
    <jndi-name>CursoDS</jndi-name>
    <xa-transaction/>

    <rar-name>firebirdsql.rar</rar-name>
    <connection-definition>javax.sql.DataSource</connection-definition>

    <config-property name="Database"
type="java.lang.String">localhost/3050:/Firebird/data/CURSO.FDB</config-property>
    <user-name>sysdba</user-name>
    <password>masterkey</password>

    <!--additional properties. only use one way of setting tx isolation, please
    <config-property name="TransactionIsolation"></config-property>
    <config-property name="TransactionIsolationName">TRANSACTION_READ_COMMITTED</config-property>
    <config-property name="BlobBufferLength"></config-property>
    <config-property name="Encoding">UNICODE_FSS</config-property>
```

```
    -->

    <min-pool-size>0</min-pool-size>
        <!-- sql to call when connection is created
        <new-connection-sql>some arbitrary sql</new-connection-sql>
        -->

        <!-- sql to call on an existing pooled connection when it is obtained from pool
        <check-valid-connection-sql>some arbitrary sql</check-valid-connection-sql>
        -->

      <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml (optional) -->
      <metadata>
         <type-mapping>Firebird</type-mapping>
      </metadata>
  </tx-connection-factory>

</connection-factories>
```

7.2.2 TomcatConnectionPooling

Modificaciones en el archivo `server.xml`:

```
  <Context path="/dbTest" docBase="dbTest"
         debug="5" reloadable="true" crossContext="true">

    <Logger className="org.apache.catalina.logger.FileLogger"
           prefix="dbTest." suffix=".txt" timestamp="true"/>

    <Resource name="jdbc/dbTest" auth="Container" type="org.firebirdsql.pool.FBWrappingDataSource"/>

    <ResourceParams name="jdbc/dbTest">
      <parameter>
        <name>factory</name>
        <value>org.firebirdsql.pool.FBWrappingDataSource</value>
      </parameter>
      <parameter>
        <name>userName</name>
        <value>SYSDBA</value>
      </parameter>
      <parameter>
        <name>password</name>
        <value>masterkey</value>
      </parameter>
      <parameter>
        <name>database</name>
        <value>localhost/3050:c:/database/EMPLOYEE.GDB</value>
      </parameter>
      <parameter>
        <name>maxConnections</name>
        <value>10</value>
      </parameter>
      <parameter>
        <name>minConnections</name>
        <value>3</value>
      </parameter>
    </ResourceParams>

  </Context>
```


8 Tomcat con Apache

8.1 El servidor Apache HTTP

El servidor Apache HTTP es el servidor web el mas utilizado en Internet. Funciona bajo varias plataformas, entre otros Windows y Linux.

El servidor Apache tiene varios módulos que se le pueden agregar, lo que extiende considerablemente sus capacidades.

Por defecto solo trata paginas estáticas en formato HTML. Pero gracias al uso de módulos, permite tratar también paginas PHP, JSP, usar repartidor de carga (load balancing), servir como proxy, etc...

Las dos últimas versiones mayores del servidor Apache HTTP son las versiones 2.0 y 2.2.

8.2 Porque usar el servidor Apache

El servidor Tomcat trabaja por defecto con el puerto 8080, lo que obliga a especificar el puerto en el URL, ya que por defecto HTTP usa el puerto 80.

Tomcat se puede configurar para utilizar el puerto 80 en vez, pero bajo Linux eso nos obligaría a correr el servidor Tomcat como ROOT ya que los puertos menos de 1024 requieren en derecho de root.

Como Tomcat ejecuta código en el servidor, es mejor no exponerlo directamente a Internet, ya que seria un riesgo adicional a nivel de seguridad.

También, el uso de Apache, ademas de ser mas seguro, permite hospedar varios sitios, con varias tecnologías (PHP,...) bajo un mismo servidor. Nos permite también filtrar las solicitudes de paginas para enviar a Tomcat solo las consultas JSP, y no las imágenes o otros tipos de archivos (paginas HTML, PDF,...), ya que eso afectaría el rendimiento de Tomcat que tendría que procesar paginas para nada.

8.3 Conectar Tomcat con Apache

La conexión en Tomcat y Apache se hace de diferente manera, dependiendo de la versión de Tomcat y de la versión de Apache.

En practica vamos a ver la conexión de Tomcat 7 con Apache 2.2, pero daré también la información de como conectar Apache 7 con Apache 2.0, ya que Apache 2.0 puede existir todavía en muchos servidores de producción.

Apache trabaja con sitios virtuales (virtual host), lo que permite configurar y hospedar varios sitios en un mismo servidor.

8.4 Conectar Tomcat 7 con Apache 2.0

Para conectar Tomcat 7 con Apache 2.0, se usa el modula mod_jk.

Para instalar el conector, hay que ponerlo disponible en la carpeta de módulos de Apache, activarlo en el archivo httpd.conf de Apache, crear los archivos mod_jk.conf y workers.properties, y activar el conector AJP en server.xml de Tomcat.

Para la documentación completa, ver el sitio de Apache HTTPD (http://httpd.apache.org) y Tomcat (http://tomcat.apache.org).

Aquí abajo un ejemplo de los archivos nuevos y las lineas a modificar en los archivos existentes.

8.4.1 Archivo httpd.conf:

Adjuntar las lineas siguientes:

```
LoadModule jk_module                  /usr/lib/apache2-prefork/mod_jk # Linux, ver abajo
Include /tomcat/conf/jk/mod_jk.conf # ruta hacia nuevo archivo mod_jk.conf

# en openSuse, se modifica mejor el archivo /etc/sysconfig/apache2:
APACHE_MODULES="... jk"
```

8.4.2 Archivo server.xml:

Adjuntar las lineas siguientes o quitar el comentario si están en comentario:

```
    <!-- Define an AJP 1.3 Connector on port 8009 -->
    <Connector port="8009"
               enableLookups="false" redirectPort="8443" protocol="AJP/1.3" />
```

8.4.3 Nuevo archivo mod_jk.conf

```
<IfModule !mod_jk.c>
  LoadModule jk_module "/usr/lib/apache2/mod_jk.so"
</IfModule>

JkWorkersFile "/tomcat/conf/jk/workers.properties" # ruta hacia nuevo archivo workers.properties
JkLogFile "/tomcat/logs/mod_jk.log"

JkLogLevel    info

<VirtualHost *:80>
    ServerName localhost
    ServerAdmin cedric@solucionjava.com
    DocumentRoot /srv/www/htdocs  # camino por defecto de los archivos del sitio web (no Tomcat)
    JkMount /* miConnector # manda solicitud de archivos de en carpeta rais (/) y abajo a miConnector
(Tomcat)
    JkUnMount /images/*.* miConnector # no manda solicitud de archvos de en carpeta /images a Tomcat
        #  JkUnMount solo esta disponible en ultimas versiones del conector.
</VirtualHost>
```

8.4.4 Nuevo archivo workers.properties

```
worker.list= miConnector

 worker.miConnector.port=8009
 worker.miConnector.host=localhost
 worker.miConnector.type=ajp13
```

8.5 Conectar Tomcat 5.5 o 6.0 con Apache 2.2

En Apache 2.2, la configuración es mas fácil que en Apache 2.0, y menos dependiendo de la versión de Tomcat.

Se utilizan los módulos de mod_proxy, mod_proxy_ajp, y opcionalmente el modulo de reparticion de carga mod_proxy_balancer.

Solo hay que modificar dos archivos existentes. Los ejemplos abajo pueden necesitar adaptaciones, dependiendo de las rutas a donde están instalados los programas/archivos.

8.5.1 Archivo httpd.conf:

Adjuntar las lineas siguientes:

```
LoadModule proxy_module                 /usr/lib/apache2-prefork/mod_proxy.so
LoadModule proxy_ajp_module             /usr/lib/apache2-prefork/mod_proxy_ajp.so
LoadModule proxy_balancer_module        /usr/lib/apache2-prefork/mod_proxy_balancer.so

<VirtualHost *:80>
    ServerName localhost # nombre del servidor. Debería coincidir con el nombre de servidor Tomcat
    ServerAdmin cedric@solucionjava.com
    DocumentRoot /srv/www/htdocs/relih_html # camino de archivos que no son se envian a Tomcat

    ProxyPass /images ! # no manda solicitud hacia /images a Tomcat pero las trata en Apache
    ProxyPass /scripts ! # no manda solicitud hacia /images a Tomcat pero las trata en Apache
    ProxyPass /styles ! # no manda solicitud hacia /images a Tomcat pero las trata en Apache
    ProxyPass / balancer://ajp-cluster/ stickysession=JSESSIONID nofailover=On # rais ==> Tomcat
    ProxyPassReverse / balancer://ajp-cluster/
    <Proxy balancer://ajp-cluster>
        BalancerMember ajp://localhost:8009/relih/ route=tomcat1a # definition de un servidor Tomcat
       # en caso de load balancing, se pueden definir aquí varios servidores
    </Proxy>
</VirtualHost>
```

8.5.2 Archivo server.xml:

Adjuntar las lineas siguientes o quitar el comentario si están en comentario:

```
    <!-- Define an AJP 1.3 Connector on port 8009 -->
    <Connector port="8009"
               enableLookups="false" redirectPort="8443" protocol="AJP/1.3" />
```

9 Cifrar con SSL y uso de HTTPS

9.1 Porque cifrar

Cuando su sitio contiene datos sensibles, o usa restricción de acceso, cifrar la transmisión disminuye el riesgo de que alguien intercepta y pueda entender el contenido de la transmisión (usuario y clave, contenido,...)

El cifrado usa un poco mas de recursos al nivel del servidor y al nivel del cliente, y también aumenta un poco el tamaño de las transmisiones, pero en un nivel razonable.

Para poder cifrar, se necesita emitir un certificado. Los certificados SSL pueden ser auto firmado (uso interno o sin necesidad de grande garantía de seguridad) o firmado por una autoridad certificadora (de 20 a 1,000 US$, dependiendo del tipo de certificado y la autoridad certificadora).

La creación de certificados auto firmados depende del sistema operativo.

Por ejemplo en Suse 10.2, se usa una consola como usuario root, y de la carpeta `/usr/share/doc/packages/apache2` se ejecuta `./mkcert.sh make --no-print-directory /usr/bin/openssl /usr/sbin/ custom`

Mas detalles sobre los certificados SSL para Suse 10.2 en
http://www.novell.com/documentation/opensuse102/index.html?page=/documentation/opensuse102/opensuse102_reference/data/sec_apache2_ssl.html

Es importante que el campo COMMON NAME corresponde con el URL del sitio web.

Los certificados deben estar disponible en el servidor.

Tomcat se puede configurar para trabajar con SSL directamente, pero aquí vamos a ver la configuracion de Apache con SSL, ya que Apache es el principal servidor web y puede comunicar con Tomcat.

Al nivel de Apache, hay que instalar y configurar el modulo SSL, y iniciar Apache con la opción `-D SSL`.

9.1.1 Archivo httpd.conf:

Adjuntar las lineas siguientes:

```
LoadModule ssl_module  /usr/lib/apache2-prefork/mod_ssl.so
<IfModule mod_ssl.c> Include ssl.conf # Depende de a donde guarda su archivo ssl.conf
</IfModule>
```

9.1.2 Archivo ssl.conf:

Revisarlo y modificar al menos las lineas siguientes:

```
<VirtualHost *:443>
#Adjuntar modificaciones de virtual host normal (ver arriba, depende de la versión)
...
SSLEngine on
...
SSLCertificateFile /etc/apache2/ssl/server.crt # Depende de a donde guarda sus certificados
SSLCertificateKeyFile /etc/apache2/ssl/server.key # Depende de a donde guarda sus certificados
...
</VirtualHost>
```


10Marcos de trabajo y EJB

10.1Marcos de trabajo

Los marcos de trabajo fueron diseñados con la idea de estandarizar el desarrollo de aplicaciones, para aumentar la rapidez del desarrollo (para los que ya conocen el marco) y disminuir el mantenimiento.

Los marcos de trabajo son muy populares, pero tienen sus pro y contra.

Pro:

- Estándar de desarrollo
- Trae objetos y códigos 'preparados', listos para usar, bien probados
- Usan el modelo MVC con EJB
- Escoden parte del código, lo que permite programar sin necesidad de entender lo que va detrás
- Se integran con las herramientas de desarrollo
- Facilita el reclutamiento de programadores (si el marco es lo suficiente popular en la área)

Contra:

- Necesidad de aprender como funciona el marco de trabajo, ademas de conocer el Java.
- Genera mucho archivos de códigos adicionales, que hay que mantener.
- Agregan una o varias capas más al código: configuración, acciones,... Puede dificultar la depuración del código
- Esconden parte del código. Si uno quiere hacer algo no previsto en el marco de trabajo, le puede salir difícil o a veces imposible
- Trabajan con EJB para el modelo, lo que obliga duplicar la base de datos en EJB
- Usan archivos XML para configuración --> XML no es orientado objeto, y es un archivo 'fijo'
- Obligan respetar el modelo MVC --> genera más código
- No hay un estándar, existen muchos: Struts2, Spring, Hibernate, ADF, JSF,...
- Para Nicaragua, ya es difícil hallar un programador Java. Será más difícil todavía hallar un programador que conoce el marco de trabajo que usted utiliza, ya que hay varios 'muy populares'

10.2EJB

Los Enterprise Java Beans son clases Java que se encuentran en un contenedor específico del servidor de aplicaciones, y que pueden ser llamados desde otros máquinas virtuales (de manera remota).

La ventaja es que un mismo objeto se puede compartir entre varias aplicaciones, web o de escritorio.

Se usan entre otros para guardar los datos en memoria (Java), con el doble objetivo de disminuir la carga de la base de datos, y de independizar el código Java de la base de datos (el driver se encarga que generar los SQL necesarios).

Existen marcos de trabajo, como Hibernate, que también permiten un cache 'de segundo nivel', para limitar al máximo las llamadas a la base de datos.

Pero el uso de EJB también tiene su lado malo:

- Obliga a recrear todos los objetos de la base de datos como clase Java
- Agrega otra capa de programación --> de depuración
- Impide el uso de ciertas capacidades de la base de datos, como los triggers.
- En ciertos casos, por razón de rendimiento, hay que escribir y hacer la llamada SQL 'manualmente', ya que posiblemente el SQL generado no es óptimo.

10.3Marco de trabajo de SoluciónJava.com

SolucionJava ha liberado parte del código que utilizó para crear la aplicación Relih (www.relih.com).

Vamos a ver las clases de este marco que están relacionadas con este curso:

- ConnectDB (JavaBean)
- Tools (JavaBean)
- Calculo (JavaBean)
- Report (TagLib)
- Pager (JavaBean)

11XDoclet

Las notacion XDoclet son anotaciones que permiten generar los archivos de configuración (web.xml, taglib.xml,...) de manera automática.

La versión de Eclipse Ganymede agrega automáticamente los nuevos Servlet al archivo web.xml, y trae una implementación de XDoclet pero que no es satisfactoria a mis ojos.

¿Porque?

1. No genera los comentarios XDoclet al crear el servlet. Solo modifica el web.xml a la creación (no en caso de cambio del servlet a otro paquete, otro nombre,...)
2. Si alguna clase tiene un comentario XDoclet, borra el archivo web.xml y lo remplaza con una generado a partir de lso tags XDoclet encontrados --> pierde todos los servlet si tag XDoclet.
3. No permite personalizar XDoclet (filtro de clases, versiónes, archivos adicionales a incluir,...)
4. Solo sirve para Servlet. No soporta los Taglibs.
5. Corre (por defecto) cada vez que se modifica un servlet --> si tienes centenas de servlet, se vuelve muy pesado

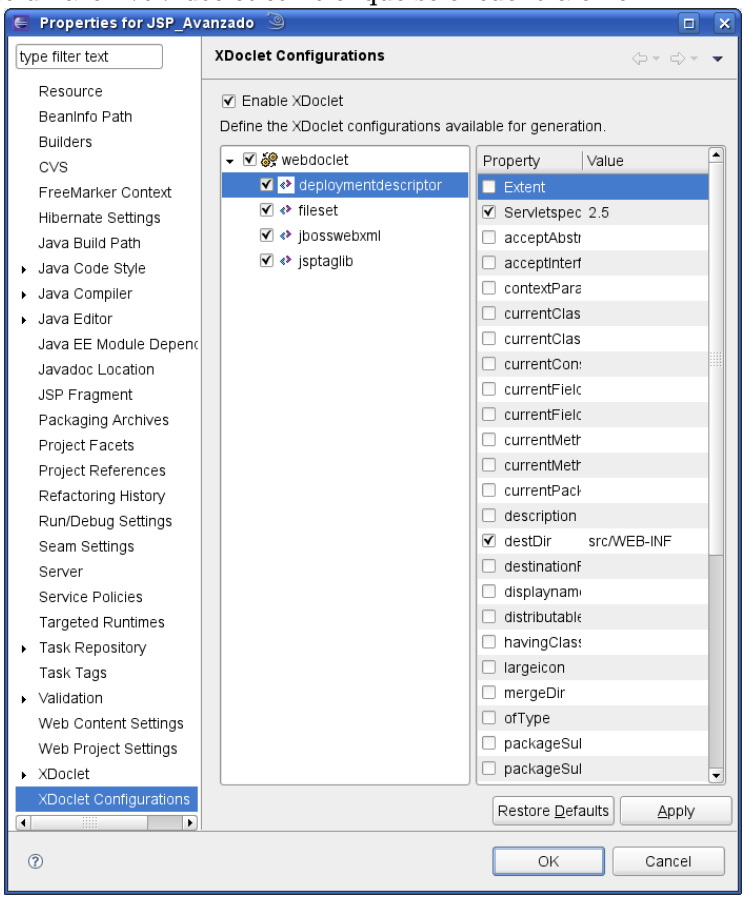
Entonces, ¿que usar?

Personalmente, uso el XDoclet del JBoss IDE para Eclipse. La funcion de XDoclet de JBoss IDE, permite definir de manera muy precisa lo que debe incluir, y funciona también para Taglibs, EJB, Hibernate, JMX,... Este plugin funciona con las versiones de JBoss Tools hasta la versión 2.1 (no en la versión 3 !).

Para activar el XDoclet de JBoss, hay que editar el archivo .project, cambiando en `<buildCommand>`:

```
<name>org.eclipse.jst.j2ee.ejb.annotations.xdoclet.xdocletbuilder</name>
<name>org.jboss.ide.eclipse.xdoclet.run.XDocletRunBuilder</name>
```

Ademas hay que copiar en la raiz del proyecto un archivo .xdoclet como el que se encuentra en el proyecto JSP_Avanzado.



Hay que incluir tambien las librerias Xdoclet en la configuracion de Eclipse. Copiar las librerias en la carpeta de Eclipse, y luego mencionar esta ubicación en Eclipse...Window...Preferences...XDoclet.

Para editar/correr la XDoclet, hay que estar en la perspectiva Java y en la vista Package Explorer.

La mala noticia:

XDoclet de JBoss está en abandono ya que no está más presente en la versión 3.0 de JBoss Tools.

El mismo sitio XDoclet no tiene nuevas versiones.

A medio plazo, hará que ir por las soluciones integradas en las herramientas existentes (como Eclipse) :-(.

12Ejercicios

1) Acciones personalizadas:
   a) Crear una acción personalizada que regresa la fecha y hora local.
   b) Crear una acción personalizada que escribe el texto en rojo, en mayúscula, y encuadrado.
   c) Modificar el tag CifraHTML para que solo cifra el código HTML, sin iteración ni adjunto de 'Ronda...'
2) Cookies:
   a) Utilizar un cookie para recordar el nombre de usuario, y proponer el ultimo nombre de usuario por defecto.
   b) Crear la opción de borrar el cookie y invalidar la sesión al mismo tiempo

3) Autenticación:
   a) Crear sitios utilizando los métodos de autenticación BASIC, DIGEST, y FORM.
   b) Crear una autenticación manejada por la aplicación

Ejercicio final (si queda tiempo):

Crear una aplicación web que:

1) Trae todo su contenido de una base de datos: Textos, etiquetas, imágenes,...
2) Tiene tres partes:
   i. una publica: bienvenida, informaciones generales, pagina de registro, lista de producto sin precio.
   ii. una con autenticación de la aplicación: lista de productos con precio.
   iii. una con autenticación del contenedor: manejo de usuarios de a aplicación
3) Tiene página de error personalizada utilizada por cada página.
4) Una vez autenticado, el nombre del usuario debe aparecer en las pantallas.
5) Manejo de un sistema de compra en línea: el usuario puede llenar una lista de compras, ver las compras anteriores, etc...