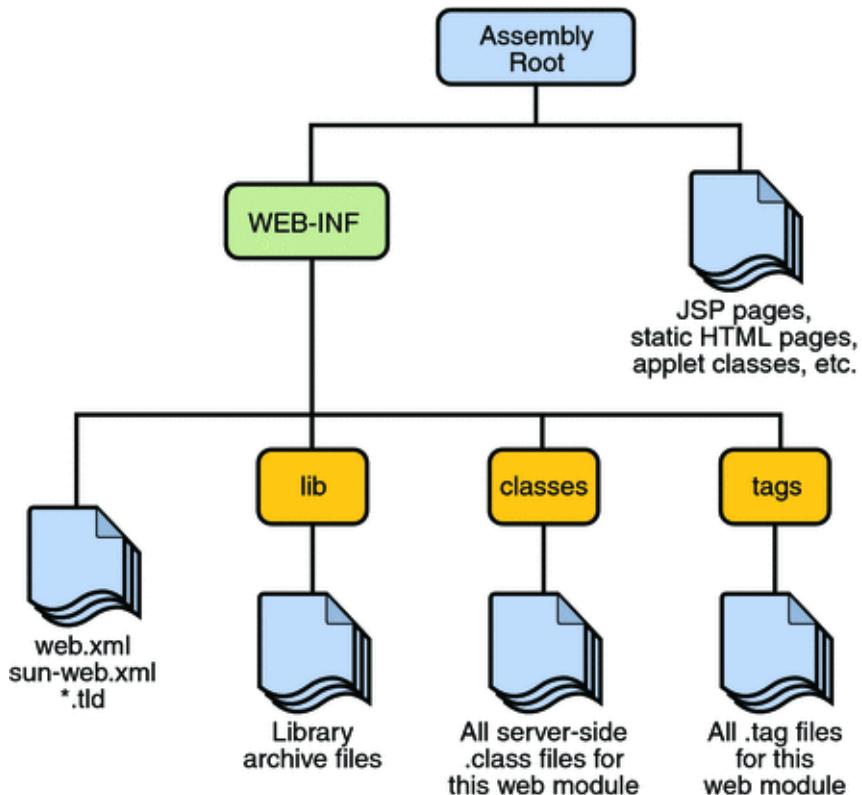




DESARROLLO DE APLICACIONES JAVA WEB



Programa: JAVA PROFESSIONAL DEVELOPER

Curso: Desarrollo de Aplicaciones Java Web

Edición: Mayo – 2016

Eric Gustavo Coronel Castillo

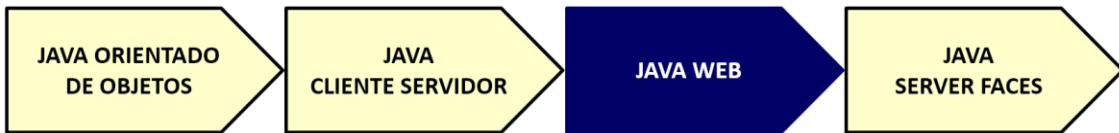
Instructor Java EE

www.desarrollasoftware.com

gcoronelc@gmail.com



PRESENTACIÓN



El uso de la tecnología Java EE para desarrollar soluciones empresariales sigue siendo la preferida por las empresas.

Por ejemplo, se desarrollan soluciones comerciales, como: Sistemas de gestión de ventas, Sistemas de gestión de compras, Sistemas de gestión de almacenes, Sistemas financieros, Etc.

También se aplica en el área del conocimiento, por ejemplo en comunicación, minería, medicina, educación, etc.

Sin duda alguna que la persona que conoce el desarrollo de software utilizando tecnología Java EE tiene muchas posibilidades en el campo laboral, y sobre bien remunerado.

El curso DESARROLLO DE APLICACIONES JAVA WEB corresponde al área de tecnología y es de naturaleza teórico-aplicativo. Tiene como propósito el desarrollo de aplicaciones web para la plataforma Java EE utilizando Servlets, Java Server Pages (JSP), librerías de etiquetas JSTL y JasperReport para los reportes. Se aplicará AJAX con JQuery y JSON con GSON.

Al finalizar el curso los estudiantes crean aplicaciones Web de mediana complejidad, bajo un enfoque Orientado a Objetos, utilizando como lenguaje de programación Java, plataforma Java EE y utilizando los patrones de software estándares que el mercado actual exige.

Esta separata es un material de referencia y complementario para el curso DESARROLLO DE APLICACIONES CON JAVA WEB.

Eirc Gustavo Coronel Castillo
Instructor Java EE
CEPS - UNI



ÍNDICE

CAPÍTULO 1 FUENDAMENTOS WEB	8
INTRODUCCIÓN AL DESARROLLO WEB	8
<i>Un poco de historia</i>	8
<i>Terminología Básica</i>	8
<i>La Web por dentro</i>	9
HTML.....	10
<i>Introducción</i>	10
<i>Estructura de una página Web</i>	11
<i>Ejemplo de una pagina Web</i>	11
JAVA SCRIPT.....	12
<i>Introducción</i>	12
<i>¿Qué es JavaScript?</i>	13
<i>¿Qué podemos hacer con Javascript?</i>	13
EXTENSIBLE MARKUP LANGUAGE – XML	13
<i>¿Qué es XML?</i>	13
<i>HTML Vs. XML</i>	14
<i>Sintaxis XML</i>	15
<i>Contenido de un Documento XML</i>	15
AJAX	15
<i>Definición</i>	15
<i>Conversación tradicional Cliente-Servidor</i>	16
<i>Interacción AJAX Cliente-Servidor</i>	17
<i>El objeto XMLHttpRequest</i>	17
<i>Propiedades del objeto XMLHttpRequest</i>	18
<i>Métodos del objeto XMLHttpRequest</i>	18
JSON	19
<i>Definición</i>	19
<i>Formato: Object</i>	19
<i>Formato: Array</i>	20
<i>Formato: Value</i>	20
JQUERY	21
<i>Definición</i>	21
<i>Sintaxis Básica</i>	22
<i>AJAX con jQuery</i>	23
<i>jQuery-UI</i>	23
CAPÍTULO 2 ARQUITECTURA JAVA EE	24
INTRODUCCIÓN	24



<i>Servidor de aplicaciones JEE</i>	26
<i>Un ejemplo de servlet</i>	28
LAS CAPAS DE LA ARQUITECTURA.....	30
<i>Escenario desde un navegador</i>	31
<i>Escenario desde una aplicación</i>	32
<i>Escenario basado en la web (web-centric application)</i>	32
CAPÍTULO 3 INTRODUCCIÓN A JAVA EE	34
INTRODUCCIÓN	34
<i>¿Qué es java EE?</i>	34
<i>¿Cómo se relaciona con Java SE?</i>	34
<i>¿Para qué Java EE?</i>	35
<i>Modelo de aplicación Java EE</i>	35
<i>Independencia del proveedor.....</i>	36
CONCEPTOS DE JAVA EE	37
<i>Clientes Java EE</i>	37
<i>Servidores Java EE</i>	38
<i>Contenedores Java EE</i>	39
<i>Componentes Java EE</i>	39
<i>APIs de Java EE</i>	40
CAPÍTULO 4 CONTENEDORES JAVA EE	43
ENPAQUETADO EN ARCHIVOS WAR	43
<i>Archivos WAR</i>	43
<i>Ventajas</i>	43
<i>Estructura.....</i>	44
<i>Descriptor de Despliegue: web.xml</i>	44
TOMCAT	45
<i>Definición</i>	45
<i>Estado de su desarrollo.....</i>	45
<i>Entorno.....</i>	45
<i>Estructura de directorios.....</i>	45
<i>Características del producto</i>	46
JBOSS.....	47
<i>Introducción</i>	47
<i>Servidor de aplicaciones JBoss</i>	48
CAPÍTULO 5 SERVLETS	49
¿QUÉ ES UN SERVLET?.....	49
<i>Abstracción.....</i>	49
<i>Aspectos Técnicos.....</i>	49
INTERACCIÓN CON LOS CLIENTES	50



<i>Un servlet sencillo</i>	51
<i>Interacción con los clientes</i>	52
Peticiones y respuestas.....	52
Manejar peticiones GET y POST	52
Problemas con los threads.....	53
Descripciones de servlets	53
<i>Objeto HttpServletRequest</i>	53
<i>Objeto HttpServletResponse</i>	54
<i>Cabecera de Datos HTTP</i>	54
<i>El método service()</i>	54
<i>Peticiones GET y POST</i>	55
Manejar peticiones GET	55
Manejar peticiones POST.....	56
PROGRAMACIÓN DE SERVLETS	57
<i>void init(ServletConfig config)</i>	58
<i>void destroy()</i>	58
<i>void service(ServletRequest request, ServletResponse reponse)</i>	58
<i>Esquema de Funcionamiento</i>	58
INTERACCIÓN CON UN SERVLET	59
<i>Consideraciones previas</i>	59
<i>Escribiendo la URL del servlet en un Navegador Web</i>	60
<i>Llamar a un Servlet desde dentro de una página HTML</i>	60
<i>Llamada a un servlet desde otro servlet</i>	61
SERVLETS Y JAVABEANS	62
SESIONES	63
<i>Introducción</i>	63
<i>La API de Seguimiento de Sesión</i>	63
<i>Objeto HttpSession de la sesión actual</i>	63
<i>Asociar Información con una Sesión</i>	63
<i>Finalizar una Sesión</i>	63
FILTROS	64
<i>Introducción</i>	64
<i>Un sencillo ejemplo</i>	65
<i>Despliegue</i>	67
SERVLET PARA CARGA DE ARCHIVO	68
<i>Introducción</i>	68
<i>El formulario</i>	68
<i>El servlet: init</i>	69
<i>El servlet: manejo de FileItem</i>	69
<i>El servlet: tratar el contenido del campo 'file'</i>	73
<i>Consideraciones de seguridad</i>	73
CAPÍTULO 6 JAVA SERVER PAGE	74



¿QUÉ ES JAVA SERVER PAGES?	74
<i>Definición</i>	74
<i>Características</i>	75
<i>Ciclo de vida a nivel de documentos</i>	75
<i>Ciclo de vida a nivel de eventos</i>	75
ELEMENTOS BÁSICOS	76
<i>Declaraciones</i>	76
<i>Expresiones</i>	77
<i>Scriptlets</i>	77
DIRECTIVAS	78
<i>Introducción</i>	78
<i>Directiva: include</i>	78
<i>Directiva: page</i>	78
<i>Directiva: taglib</i>	79
ACCIONES	80
<i>Acción: <jsp:forward></i>	80
<i>Acción: <jsp:param></i>	80
<i>Acción: <jsp:include></i>	81
<i>Acción: <jsp:useBean></i>	81
<i>Acción: <jsp:getProperty></i>	83
<i>Acción: <jsp:setProperty></i>	83
OBJETOS IMPLÍCITOS	84
<i>Objeto: request</i>	84
<i>Objeto: response</i>	84
<i>Objeto: out</i>	85
<i>Objeto: session</i>	85
<i>Objeto: application</i>	85
<i>Objeto: config</i>	86
<i>Objeto: pageContext</i>	86
<i>Objeto: page</i>	86
MANEJO DE ESTADOS	86
<i>Cookies</i>	87
<i>URL Rewriting</i>	87
<i>Crear un cookie</i>	87
<i>Recuperar un cookie</i>	89
<i>Utilizar los cookies</i>	90
<i>Las sesiones en el protocolo HTTP</i>	93
<i>javax.servlet.http.HttpSession</i>	94
<i>Cerrar sesiones</i>	95
<i>Session timeout</i>	95
MANEJANDO LOS EVENTOS DE SESIÓN	96
<i>HttpSessionAttributeListener</i>	97



<i>HttpSessionBindingListener</i>	97
<i>HttpSessionListener</i>	97
<i>HttpSessionActivationListener</i>	98
CAPÍTULO 7 JAVASERVER PAGES STANDARD TAG LIBRARY	99
¿QUÉ SON TAG LIBRARIES?	99
CARACTERÍSTICAS	99
¿QUÉ ES “EL”?	100
<i>Variables implícitas definidas en EL:</i>	100
JSTL TAG LIBRARIES	101
<i>Declarar Tag Libraries</i>	101
¿QUÉ ES JSTL?	101
USO DE LAS LIBRERÍAS ESTÁNDARES DE JSTL	102
<i>Core</i> :.....	102
<i>Internacionalizacion / Formateo</i>	105
Acciones de formateo:.....	105
Acciones de internacionalización:	106
PROCESAMIENTO XML.....	106
SQL/DB	107
DESARROLLAR ETIQUETAS JSP PERSONALIZADAS	108
<i>Introducción</i>	108
<i>Etiquetas JSP Personalizadas</i>	109
<i>Beneficios de las Etiquetas Personalizadas</i>	109
<i>Definir una etiqueta</i>	110
<i>Primera etiqueta</i>	111
Desarrollar el controlador de etiqueta	111
Crear el descriptor de librería de etiquetas	113
Probar la etiqueta	114
<i>Etiquetas parametrizadas</i>	117
<i>Librerías de etiquetas</i>	119
<i>Etiquetas con cuerpo</i>	122
Una evaluación.....	123
Multiples evaluaciones	125
<i>Guías de programación</i>	127
CAPÍTULO 8 PATRÓN: MODEL VIEW CONTROLLER	129
INTRODUCCIÓN	129
<i>¿QUÉ ES UN PATRÓN DE DISEÑO?</i>	129
<i>¿POR QUÉ DEBEMOS UTILIZAR PATRONES DE DISEÑO?</i>	129
<i>CLASIFICACIÓN DE LOS PATRONES</i>	130
PATRÓN: MODEL VIEW CONTROLLER	130
<i>Introducción</i>	131
<i>Orígenes del patrón MVC</i>	131



<i>El controlador</i>	132
<i>El modelo</i>	133
<i>La vista</i>	134
<i>Resumen</i>	134
VARIANTES DEL PATRÓN MVC.....	135
<i>Variante inicial</i>	135
<i>Variante intermedia</i>	135
CAPÍTULO 9 JQUERY.....	136
INTRODUCCIÓN	136
SELECTORES JQUERY	137
<i>Selectores básicos</i>	137
<i>Selectores de atributos</i>	137
<i>Selectores de widgets</i>	138
<i>Otros selectores</i>	139
GESTIONANDO UNA COLECCIÓN JQUERY	140
MODIFICAR LA PÁGINA CON JQUERY.....	141
EVENTOS EN JQUERY.....	142
ANIMACIONES CON JQUERY.....	144
JQUERY Y AJAX	145
CAPÍTULO 10 FUENTES DE DATOS	147
FUENTES DE DATOS BÁSICAS.....	147
REPASO RÁPIDO DE JNDI.....	148
POOL DE CONEXIONES	148
<i>Usando Tomcat 5.x, 6.x</i>	148
<i>Usando GlassFish v2, v3</i>	150
<i>Usando JBoss 5.x, 6.x</i>	153



Capítulo 1

Fundamentos Web

INTRODUCCIÓN AL DESARROLLO WEB

Un poco de historia

En 1989, Tim Berners-Lee inventó la World Wide Web. Él acuñó el término "World Wide Web", escribió el primer servidor World Wide Web, "httpd", y el primer programa cliente (un navegador y editor), "WorldWideWeb", en octubre de 1990. Él escribió la primera versión del "HyperText Markup Language" (HTML), el lenguaje de formato de documentos con capacidad para los enlaces de hipertexto que se convirtió en el principal formato de publicación para la Web. Las especificaciones iniciales para URIs, HTTP y HTML fueron mejoradas y discutidas en círculos más amplios como la propagación de tecnología web.

En octubre de 1994, Tim Berners-Lee fundó el World Wide Web Consortium (W3C) en el Instituto de Tecnología de Massachusetts, Laboratorio de Ciencias de la Computación [MIT/LCS], en colaboración con el CERN, donde se originó la Web, con el apoyo de DARPA y la European Commission. En abril de 1995, el INRIA (Institut National de Recherche en Informatique et Automatique) se convirtió en la primera sede europea del W3C, seguida por la Universidad de Keio de Japón (Shonan Fujisawa Campus) en Asia en 1996. En 2003, el ERCIM (European Research Consortium in Informatics and Mathematics) asumió en papel de sede de European W3C para INRIA.

Referencias:

- <http://www.w3.org/Consortium/history.html>
- <http://www.w3.org/History/1989/proposal.html>

Terminología Básica

La WWW es una combinación de 5 ideas básicas

- Hipertexto
 - Habilidad de navegar desde un documento a otro a través de conexiones (hiperenlaces)
- Identificadores de Recursos
 - Permite encontrar un recurso particular (un documento, imagen) en la red a través de dicho identificador



- Uniform Resource Identifier(URI), Uniform Resource Locator(URL)
- Modelo cliente / servidor
 - Un cliente software demanda servicios o recursos a un servidor software
- Un lenguaje de marcado
 - Además de texto incluyen conjuntos de caracteres especiales que indican al navegador como presentar dicho texto (HTML)
- La WWW no es Internet, es un servicio que está montado sobre Internet
 - Internet (la red) está formado por el conjunto de computadores que están interconectados entre sí.

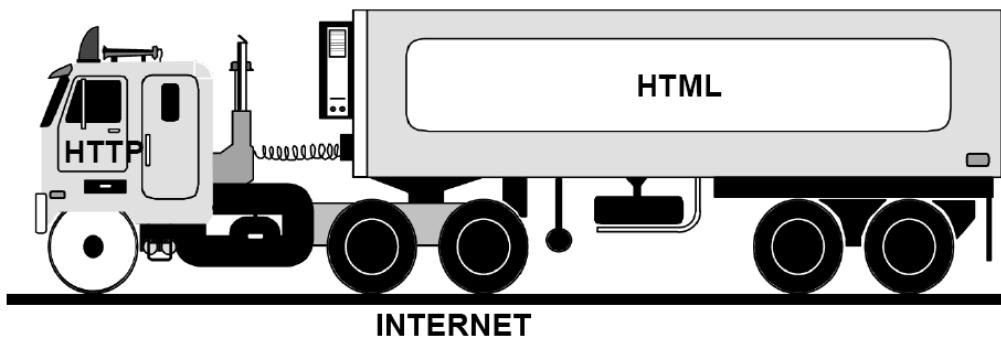


Figura 1: Funcionamiento de la WWW

La Web por dentro

Una página WEB no es un único objeto, está compuesto por múltiples archivos (imágenes, video, películas flash, JavaScript...)

Existe un archivo principal cuyo contenido es HTML y que describe el contenido de la página, tanto texto como otros elementos



Figura 2: Página Web

HTML

Introducción

- HTML es el lenguaje con el que se definen páginas WEB
 - Es texto plano
 - Se trata de un conjunto de etiquetas que sirven para indicar que el texto dentro de la etiqueta hay que tratarlo de manera especial
 - ✓ Ej: Texto en negrita
- Ha evolucionado a lo largo del tiempo desde que Tim Berners-Lee propusiera la primera versión
 - Se ha llevado a cabo un proceso de estandarización por la World Wide Web Consortium: <http://www.w3.org>
 - Las versiones actuales son HTML 4.0.1 y XHTML 1.1

Estructura de una página Web

- Las etiquetas están estructuradas (en un árbol)
- Hay dos secciones claramente diferenciadas (cabecera y cuerpo)
- Las etiquetas van pareadas

```
<html>
  <head>
    <title>Título de la página</title>
    <script type="text/javascript">...</script>
  </head>
  <body>
    <h1>Encabezado</h1>
    <p>Podemos crear un párrafo <b>resaltando</b>
      parte de sus contenidos.</p>
  </body>
</html>
```

Ejemplo de una pagina Web

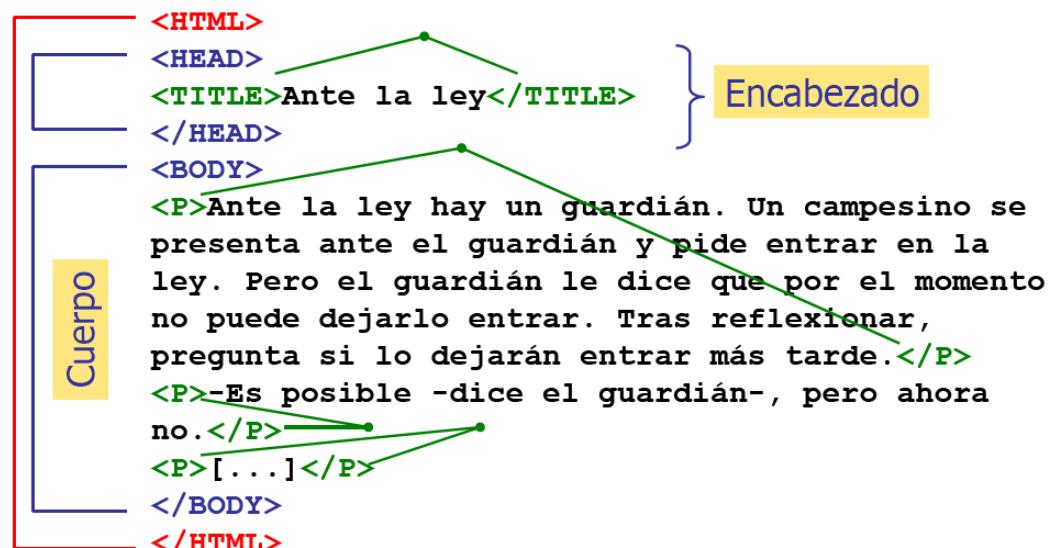


Figura 3: Ejemplo de una página Web



El resultado en el navegador es el siguiente:

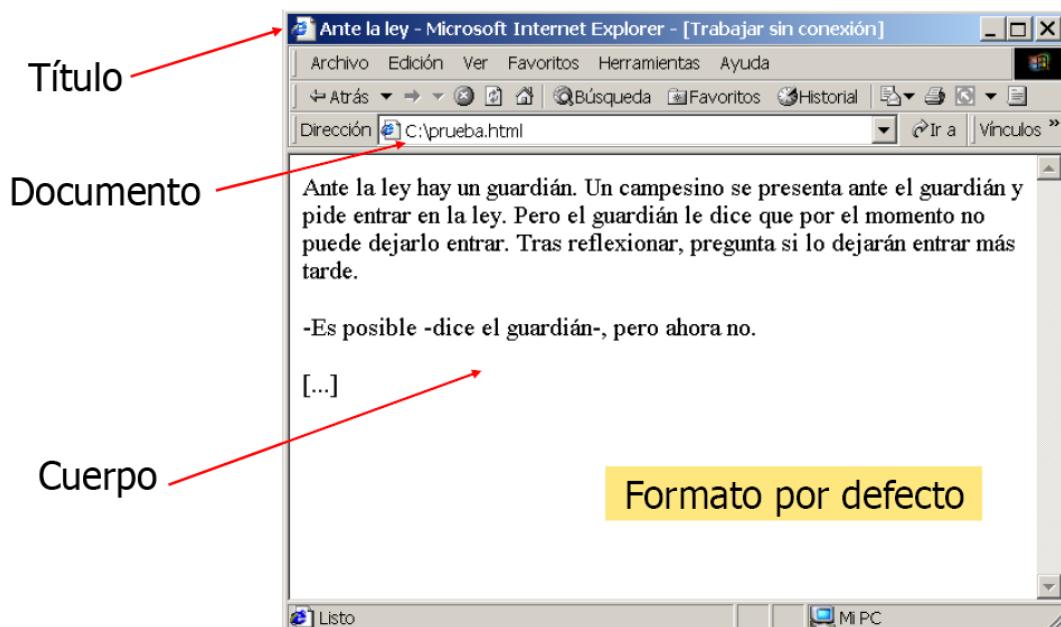


Figura 4: Visualización de una página Web en el browser

JAVA SCRIPT

Introducción

- ¿Por qué usar JavaScript?
 - Permite crear efectos visualmente atractivos
 - Permite crear sitios WEB que se visualicen de la misma manera en distintos navegadores
 - Permite suplir las carencias de implementación de especificaciones entre los distintos navegadores
 - Permite interactuar con el usuario sin los problemas de la latencia de la red.
- ¿Por qué aprender JavaScript?
 - Habitualmente se copian scripts genéricos disponibles en la WEB.
 - En la mayor parte de casos es necesario particularizarlos para adaptarlo a nuestro sitio WEB.



¿Qué es JavaScript?

- Características
 - Es un lenguaje de script (secuencia de instrucciones)
 - Tiene algunas características de orientación a objetos
 - Es un lenguaje interpretado (No se compila)
- JavaScript NO es Java
- Se ejecuta en el cliente
 - El navegador del cliente es el que se encarga de interpretar y ejecutar los comandos de JavaScript

¿Qué podemos hacer con Javascript?

- Crear páginas WEB más atractivas
 - Crear etiquetas dinámicamente
 - Intercambiar imágenes
- Controlar el navegador
 - Mostrar mensajes al usuario
 - Crear ventanas emergentes
- Interactuar con formularios
 - Validar formularios
- Interactuar con el usuario
 - Responder a sucesos generados al interactuar el usuario con el navegador

EXTENSIBLE MARKUP LANGUAGE – XML

¿Qué es XML?

XML (Extensible Markup Language) es un meta-lenguaje de marcas, es decir, permite que el usuario diseñe sus propias marcas (tags) y les dé el significado que se le antoje, con tal de que siga un modelo coherente.

```
<Catalogo>
  <Articulo>
    <Nombre>Monitor</Nombre>
```



```
<Precio>120.00</Precio>
</Articulo>
<Articulo>
    <Nombre>Teclado</Nombre>
    <Precio>24.00</Precio>
</Articulo>
<Articulo>
    <Nombre>Mouse</Nombre>
    <Precio>18.00</Precio>
</Articulo>
</Catalogo>
```

HTML Vs. XML

HTML describe como mostrar la data.

```
<BODY>
    <TABLE BORDER=1>
        <TR>
            <TD>Nombre</TD>
            <TD>Juan</TD>
        </TR>
        <TR>
            <TD>Sueldo</TD>
            <TD>3500</TD>
        </TR>
        <TR>
            <TD>Region</TD>
            <TD>Lima</TD>
        </TR>
    </TABLE>
</BODY>
```

XML define el significado de la data.

```
<empleado>
    <nombre>Juan</nombre>
    <sueldo>3500</sueldo>
    <region>Lima</region>
</empleado>
```



Sintaxis XML

XML usa etiquetas para definir el contexto de los datos.

La unidad básica de información es el elemento.

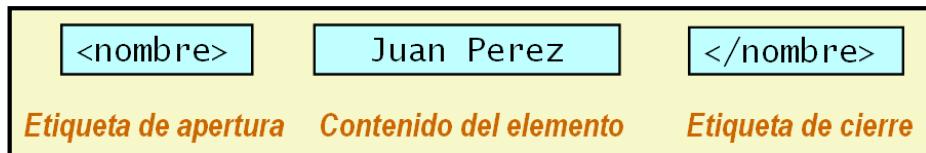


Figura 5: Estructura de un elemento XML

Los elementos pueden ser anidados

```
<empleado>
    <nombre>Juan Perez</nombre>
    <suelo>3500</suelo>
</empleado>
```

Figura 6: Elementos anidados en un documento XML

Contenido de un Documento XML

Instrucción de Procesamiento	<?xml version="1.0"?>
Comentario	<!-- Esto es un comentario -->
Elemento raíz	<empleado>
Elemento hijo	<nombre>Juan Perez</nombre>
Elemento vacío	<cargo />
Elemento con atributo	<suelo moneda="Soles"> 3500 </suelo>
Fin de elemento raíz	</empleado>

AJAX

Definición

AJAX, acrónimo de **A**synchronous **J**ava**S**cript **A**nd **X**ML (JavaScript y XML asíncronos, donde XML es un acrónimo de eXtensible Markup Language).



Es una técnica de desarrollo web para crear aplicaciones interactivas. Éstas se ejecutan en el cliente, es decir, en el navegador del usuario, y mantiene comunicación asíncrona con el servidor en segundo plano.

De esta forma es posible realizar cambios sobre la misma página sin necesidad de recargarla. Esto significa aumentar la interactividad, velocidad y usabilidad en la misma.

AJAX es la combinación de tres tecnologías ya existentes:

- XHTML y hojas de estilos en cascada (CSS) para el diseño que acompaña a la información.
- Document Object Model (DOM) accedido con un lenguaje de scripting por parte del usuario, especialmente implementaciones de ECMAScript como JavaScript, para mostrar e interactuar dinámicamente con la información presentada.
- El objeto XMLHttpRequest para intercambiar datos asíncronicamente con el servidor web.
- XML es el formato usado comúnmente para la transferencia devuelta por el servidor, aunque cualquier formato puede funcionar, incluyendo HTML preformateado, texto plano, JSON y hasta EBML.

Conversación tradicional Cliente-Servidor

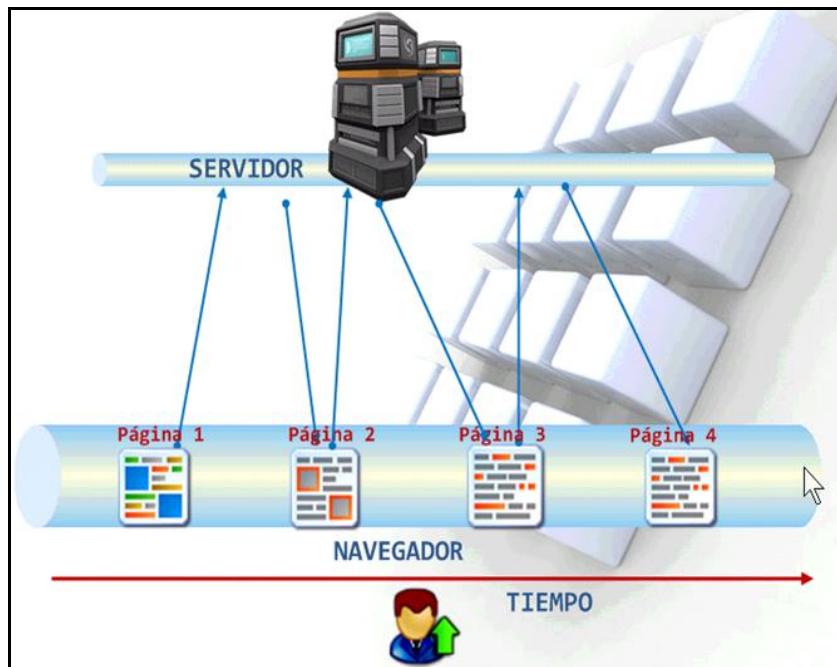


Figura 7: Esquema tradicional de una aplicación Web

Interacción AJAX Cliente-Servidor

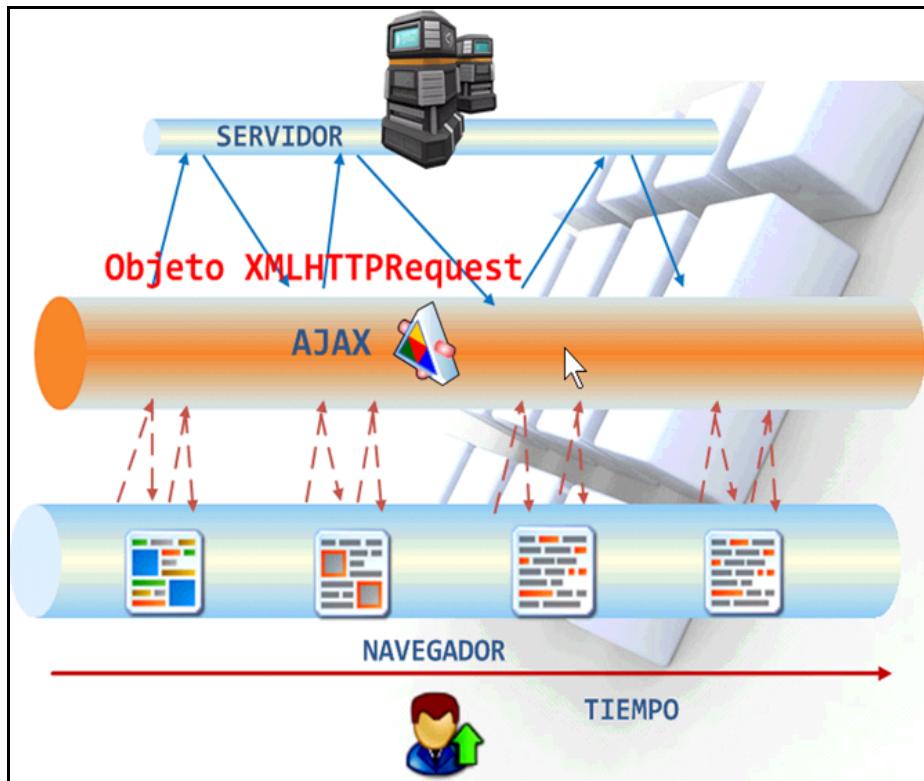


Figura 8: Esquema de una aplicación Web con AJAX

El objeto XMLHttpRequest

Diferentes reglas para diferentes navegadores.

```
function getXMLHttpRequest() {
    var req;
    try {
        req = new XMLHttpRequest();
    } catch(err1) {
        try {
            req = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (err2) {
            try {
                req = new ActiveXObject("Microsoft.XMLHTTP");
            } catch (err3) {
                req = false;
            }
        }
    }
}
```



```
    return req;  
}
```

Propiedades del objeto XMLHttpRequest

Propiedad	Descripción
onreadystatechange	Determina el gestor de evento que será llamado cuando la propiedad readyState del objeto cambie.
readyState	Número entero que indica el estado de la petición: 0: No iniciada 1: Cargando 2: Cargado 3: Interactivo 4: Completado
responseText	Datos devueltos por el servidor como una cadena de texto.
responseXML	Datos devueltos por el servidor como un objeto XMLDocument que se puede recorrer usando las funciones de JavaScript DOM.
status	Código de estado HTTP devuelto por el servidor.
statusText	Mensaje de texto enviado por el servidor junto al código (status), para el caso de código 200 contendrá "OK".

Métodos del objeto XMLHttpRequest

Propiedad	Descripción
abort()	Detiene la petición actual.
getAllResponseHeaders()	Devuelve todas las cabeceras como una cadena.
getResponseHeader(etiqueta)	Devuelve el valor de la etiqueta en las cabeceras de la respuesta.
open(método,URL,asíncrona)	Especifica un método HTTP (GET o POST), la URL objetivo, y si la petición debe ser manejada asíncronamente (true o false).
send(contenido)	Envía la petición, opcionalmente con datos POST.
setRequestHeader(etiqueta,valor)	Establece el valor de una etiqueta de las cabeceras de petición.

JSON

Definición

JSON (JavaScript Object Notation - Notación de Objetos de JavaScript) es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generarlo. Está basado en un subconjunto del Lenguaje de Programación JavaScript, Standard ECMA-262 3rd Edition - Diciembre 1999. JSON es un formato de texto que es completamente independiente del lenguaje pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos.

JSON está constituido por dos estructuras:

- Una colección de pares de nombre/valor. En varios lenguajes esto se conoce como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un arreglo asociativo.
- Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como arreglos, vectores, listas o secuencias.

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

JSON se puede escribir en los siguientes formatos:

- Object
- Array
- Value

Formato: Object

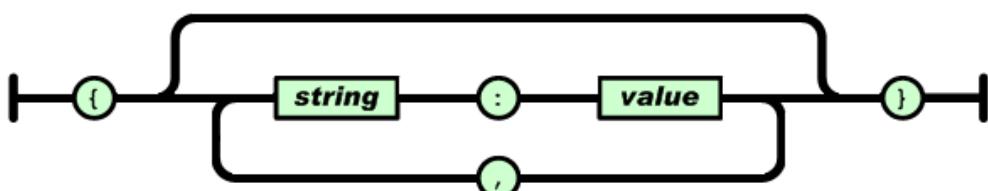


Figura 9: Formato JSON - Object



Un objeto es un conjunto desordenado de pares nombre/valor. Un objeto comienza con { (llave de apertura) y termine con } (llave de cierre). Cada nombre esta seguido por : (dos puntos) y los pares nombre/valor están separados por , (coma).

Formato: Array

Un arreglo es una colección de valores. Un arreglo comienza con [(corchete izquierdo) y termina con] (corchete derecho). Los valores se separan por , (coma).

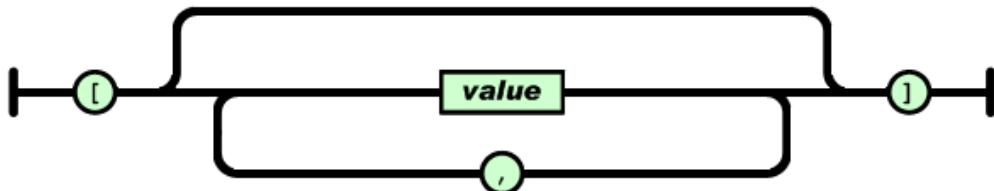


Figura 10: Formato JSON – Array

Formato: Value

Un valor puede ser una cadena de caracteres con comillas dobles, o un número, o true o false o null, o un objeto o un arreglo. Estas estructuras pueden anidarse.

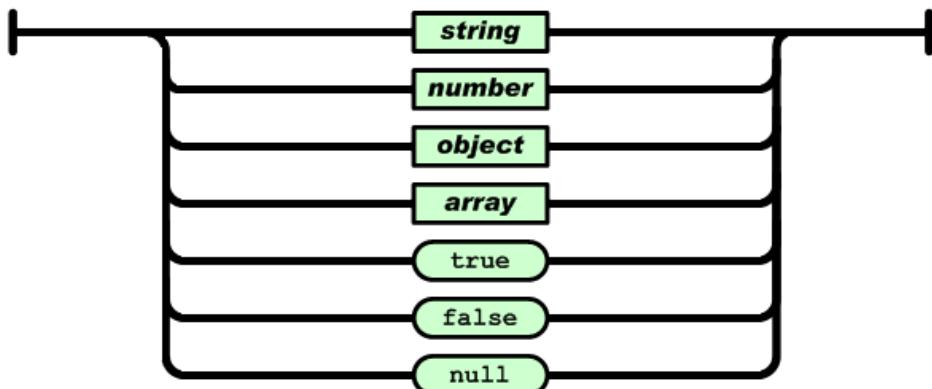


Figura 11: Formato JSON – Value

Una cadena de caracteres es una colección de cero o más caracteres Unicode, encerrados entre comillas dobles, usando barras divisorias invertidas como escape. Un carácter está representado por una cadena de caracteres de un único carácter. Una cadena de carateres es parecida a una cadena de caracteres C o Java.

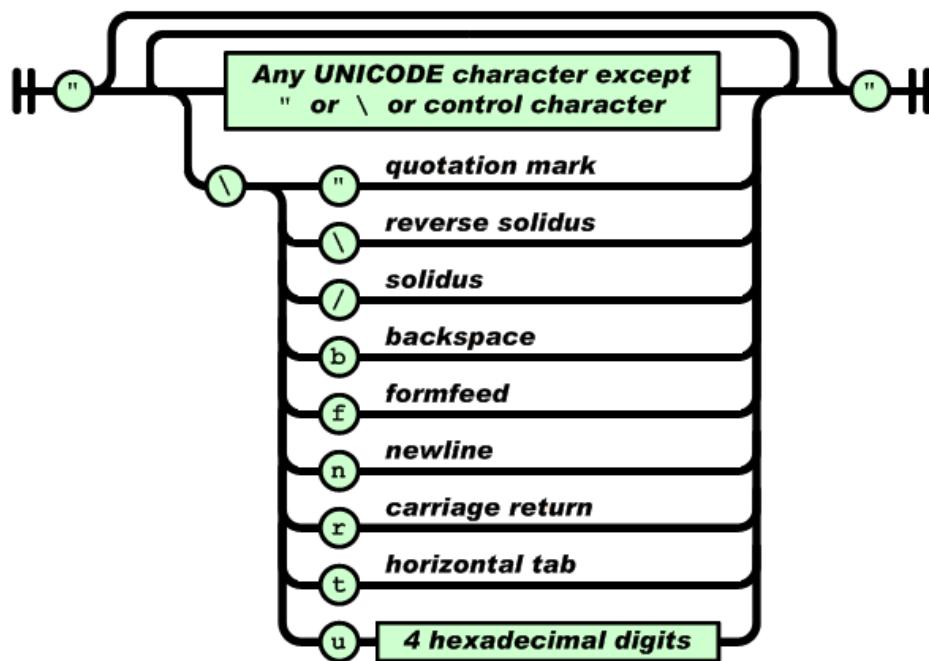


Figura 12: Formato JSON – String

Un número es similar a un número C o Java, excepto que no se usan los formatos octales y hexadecimales.

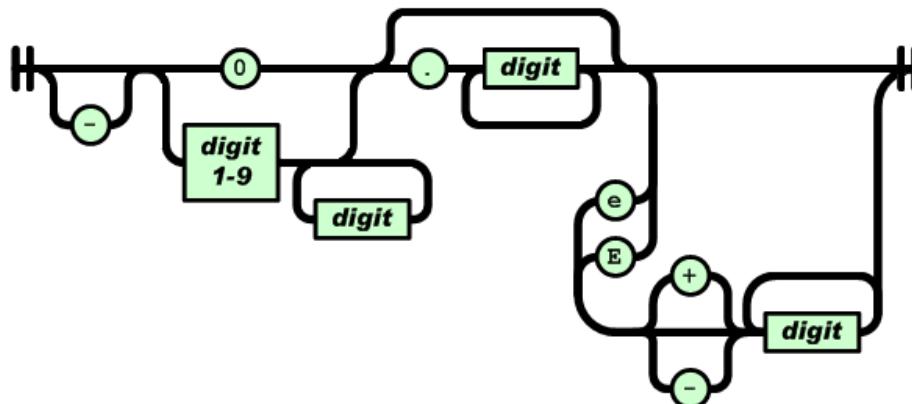


Figura 13: Formato JSON – Number

JQUERY

Definición

jQuery es framework de JavaScript, que permite simplificar la manera de interactuar con los documentos HTML, manipular el árbol DOM, manejar eventos, desarrollar animaciones y agregar interacción con tecnología AJAX.

Para obtener la librería debe acceder al siguiente sitio web: <http://jquery.com/>

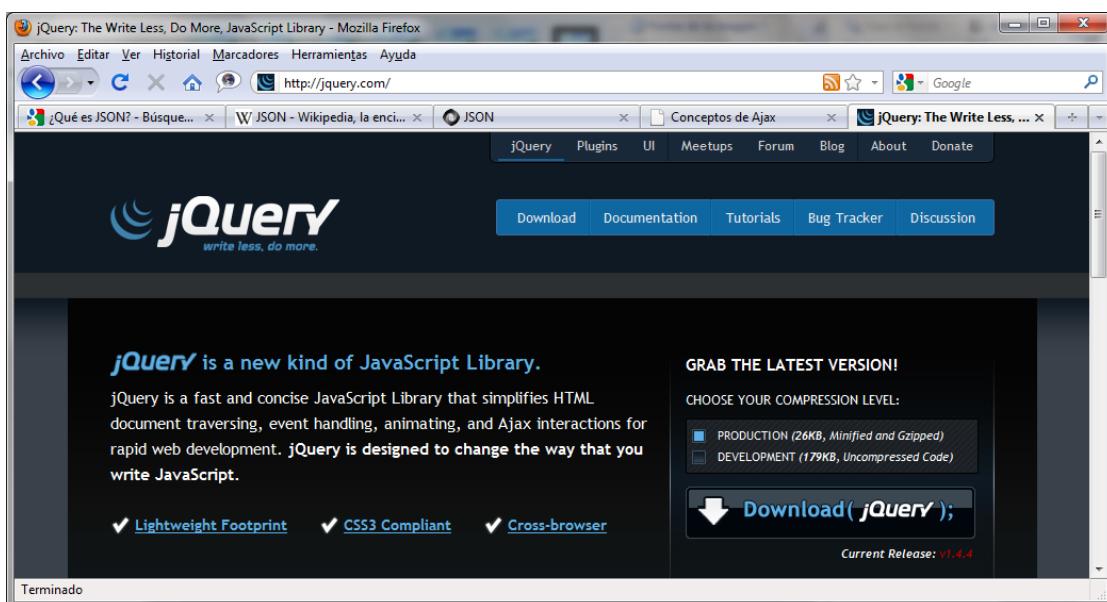


Figura 14: JQuery

Sintaxis Básica

- Enlazar la librería a nuestras páginas web:

```
<script type="text/javascript" src="jquery.js"></script>
```

- Estructura de una sentencia jQuery:

```
$(elemento).evento(funcion-o-parametro);
```

- Sentencia de inicio de jQuery:

```
<script type="text/javascript">
$(document).ready( function() {
    ...
});
```



AJAX con jQuery

```
$.ajax({  
    url: 'procesar.jsp',  
    type: 'POST',  
    data: 'n1=' + n1 + '&n2=' + n2,  
    success: function(result){  
        $('#suma').val(result);  
    }  
});
```

jQuery-UI

Es un plugin para jQuery que provee librerías para interacción y animación, efectos avanzados y de alto nivel, que se pueden utilizar para construir aplicaciones web altamente interactivas.



Figura 15: JQuery-UI



Capítulo 2

Arquitectura Java EE

INTRODUCCIÓN

Las razones que empujan a la creación de la plataforma JEE:

- **Programación eficiente.** Para conseguir productividad es importante que los equipos de desarrollo tengan una forma estándar de construir múltiples aplicaciones en diversas capas (cliente, servidor web, etc.). En cada capa necesitaremos diversas herramientas, por ejemplo en la capa cliente tenemos applets, aplicaciones Java, etc. En la capa web tenemos servlets, páginas JSP, etc. Con JEE tenemos una tecnología estándar, un único modelo de aplicaciones, que incluye diversas herramientas; en contraposición al desarrollo tradicional con HTML, Javascript, CGI, servidor web, etc. que implicaba numerosos modelos para la creación de contenidos dinámicos, con los lógicos inconvenientes para la integración.
- **Extensibilidad frente a la demanda del negocio.** En un contexto de crecimiento de número de usuarios es precisa la gestión de recursos, como conexiones a bases de datos, transacciones o balanceo de carga. Además los equipos de desarrollo deben aplicar **un estándar que les permita abstraerse de la implementación del servidor**, con aplicaciones que puedan ejecutarse en múltiples servidores, desde un simple servidor hasta una arquitectura de alta disponibilidad y balanceo de carga entre diversas máquinas.
- **Integración.** Los equipos de ingeniería precisan estándares que favorezcan la integración entre diversas capas de software.

La plataforma JEE implica una forma de implementar y desplegar aplicaciones empresariales. La plataforma se ha abierto a numerosos fabricantes de software para conseguir satisfacer una amplia variedad de requisitos empresariales. La arquitectura JEE implica un **modelo de aplicaciones distribuidas en diversas capas o niveles (tier)**. La capa cliente admite diversos tipos de clientes (HTML, Applet, aplicaciones Java, etc.). La capa intermedia (middle tier) contiene subcapas (el contenedor web y el contenedor EJB). La tercera capa dentro de esta visión sintética es la de aplicaciones 'backend' como ERP, EIS, bases de datos, etc. Como se puede ver un concepto clave de la arquitectura es el de **contenedor**, que dicho de forma genérica no es más que un entorno de ejecución estandarizado que ofrece unos servicios por

medio de componentes. Los componentes externos al contenedor tienen una forma estándar de acceder a los servicios de dicho contenedor, con **independencia del fabricante**.

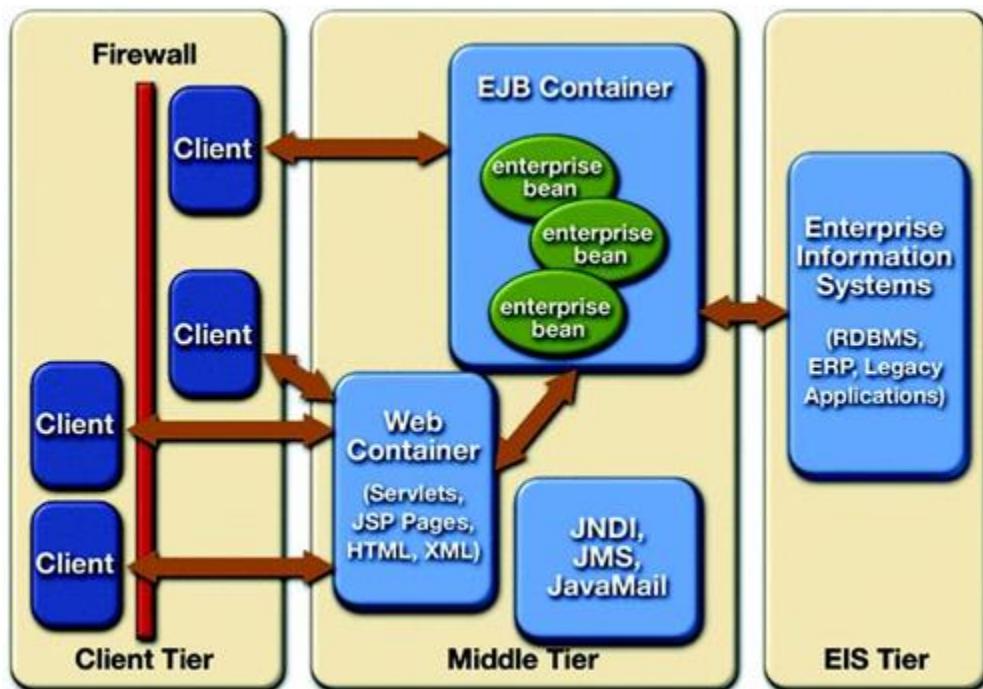


Figura 16: Arquitectura Java EE

Algunos tipos de contenedores:

- Contenedor Web, también denominado contenedor Servlet/JSP, maneja la ejecución de los servlets y páginas JSP. Estos componentes se ejecutan sobre un servidor Enterprise Edition.
- Contenedor Enterprise JavaBeans, que gestiona la ejecución de los EJB. Esta ejecución requiere de un server EE.

Los contenedores incluyen **descriptores de despliegue** (deployment descriptors), que son archivos XML que nos sirven para configurar el entorno de ejecución: rutas de acceso a aplicaciones, control de transacciones, parámetros de inicialización, etc.

La plataforma JEE incluye APIs para el acceso a sistemas empresariales:

- JDBC es el API para acceso a GBDR desde Java.
- Java Transaction API (JTA) es el API para manejo de transacciones a través de sistemas heterogéneos.
- Java Naming and Directory Interface (JNDI) es el API para acceso a servicios de nombres y directorios.



- Java Message Service (JMS) es el API para el envío y recepción de mensajes por medio de sistemas de mensajería empresarial como IBM MQ Series.
- JavaMail es el API para envío y recepción de email.
- Java IDL es el API para llamar a servicios CORBA.

Servidor de aplicaciones JEE

A continuación vamos a entrar en más detalle. Para ello, hemos **subrayado en el siguiente gráfico los elementos más importantes** y usuales. La **arquitectura de un servidor de aplicaciones** incluye una serie de subsistemas:

- **Servidor HTTP** (también denominado servidor Web o servidor de páginas). Un ejemplo, el servidor Apache.
- **Contenedor de aplicaciones** o contenedor Servlet/JSP. Un ejemplo, Tomcat (que incluye el servicio anterior sobre páginas).
- **Contenedor Enterprise Java Beans**, que contiene aplicativos Java de interacción con bases de datos o sistemas empresariales. Un ejemplo es JBoss que contiene a los anteriores (servidor de páginas web y contenedor de aplicación web).

Pero conviene empezar por el principio, es decir, el lenguaje básico de interconexión: el protocolo **HTTP**. Es un protocolo de aplicación, generalmente implementado sobre TCP/IP. Es un protocolo sin estado basado en solicitudes (request) y respuestas (response), que usa por defecto el puerto 8080:

- "**Basado en peticiones y respuestas**": significa que el cliente (por ejemplo un navegador) inicia siempre la conexión (por ejemplo, para pedir una página). No hay posibilidad de que el servidor realice una llamada de respuesta al cliente (retrollamada). El servidor ofrece la respuesta (la página) y cierra la conexión. En la siguiente petición del cliente se abre una conexión y el ciclo vuelve a empezar: el servidor devuelve el recurso y cierra conexión.
- "**Sin estado**": el servidor cierra la conexión una vez realizada la respuesta. No se mantienen los datos asociados a la conexión. Más adelante veremos que hay una forma de persistencia de datos asociada a la "sesión".

¿Qué ocurre cuando un navegador invoca una aplicación? El cliente (el navegador) no invoca directamente el contenedor de aplicaciones, sino que llama al servidor web por medio de HTTP. **El servidor web se interpone en la solicitud** o invocación; siendo el servidor web el responsable de trasladar la solicitud al contenedor de aplicaciones.

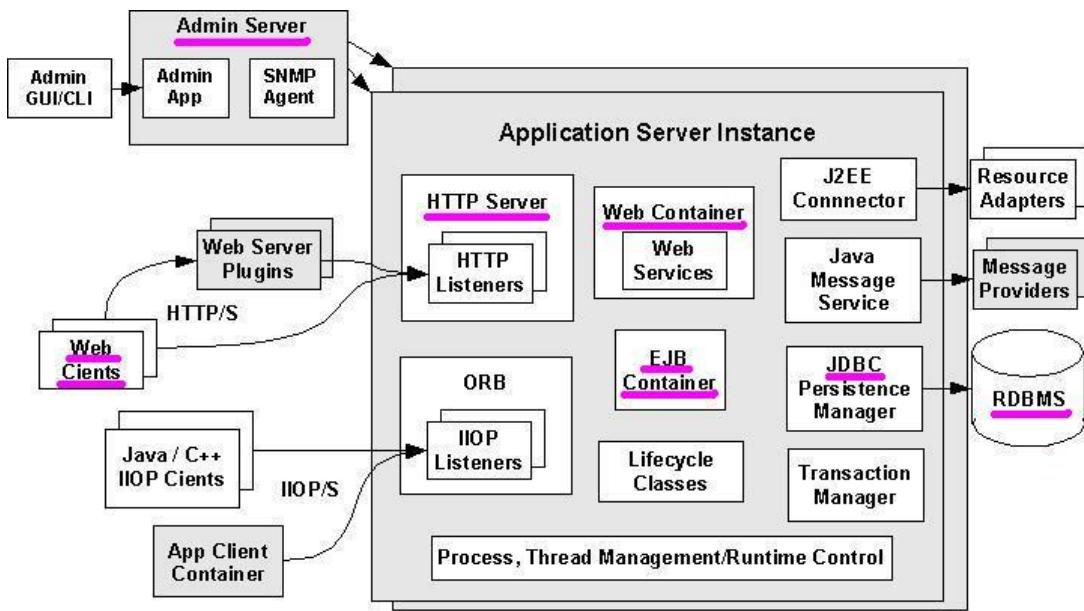


Figura 17: Servidor de aplicaciones Java EE

Aspectos a considerar de forma síncrona:

- El **cliente** (normalmente por medio de un navegador, aunque podría ser una aplicación Swing) solicita un recurso por medio de HTTP. Para localizar el recurso al cliente especifica una URL (Uniform Resource Locator), como por ejemplo <http://www.host.es/aplicacion/recurso.html>. El URI (Uniform Resource Identifier) es el URL excluyendo protocolo y host. Existen diversos métodos de invocación, aunque los más comunes son POST y GET. Los veremos más adelante.
- Sobre una misma máquina podemos tener **diversas instancias** de un AS (Application Server), procurando que trabajen sobre puertos diferentes, para que no se produzcan colisiones (por defecto HTTP trabaja con 8080).
- Un servicio crucial es la capacidad de recibir peticiones HTTP, para lo cual tenemos un **HTTP Listener** (aunque puede tener listeners para otros protocolos como IIOP).
- La solicitud llega al servidor de páginas web, que tiene que descifrar si el recurso solicitado es un recurso estático o una aplicación. Si es una aplicación delega la solicitud en el **contenedor web** (contenedor Servlet/JSP). El contenedor web gestiona la localización y ejecución de Servlets y JSP, que no son más que pequeños programas. El contenedor web o contenedor Servlet/JSP recibe la solicitud. Su máquina Java (JVM) invoca al objeto Servlet/JSP, por tanto nos encontramos ante **un tipo de aplicaciones que se ejecutan en el servidor**, no en el cliente. No



conviene olvidar que **un Servlet o un JSP no es más que una clase Java**.

Lo más interesante en este sentido es que:

- La JVM (generalmente) no crea una instancia de la clase por cada solicitud, sino que **con una única instancia de un Servlet/JSP se da servicio a múltiples solicitudes HTTP**. Esto hace que el consumo de recursos sea pequeño en comparación con otras opciones, como el uso de CGIs, en donde cada solicitud se resuelve en un proceso.
- **Para cada solicitud se genera un hilo** (thread) para resolverla (pero con una única instancia de la clase, como hemos dicho).
- Un Application Server tendrá un **servidor de administración** (y normalmente un manager de las aplicaciones).

Otros aspectos del contenedor web:

- El contenedor necesita conectores que sirven de intermediarios para comunicarse con elementos externos. Los **conectores** capacitan al AS para acceder a sistemas empresariales (backends). Por ejemplo:
- El **Java Message Service** ofrece conectividad con sistemas de mensajería como MQSeries.
- El API **JDBC** da la capacidad de gestionar bases de datos internas al AS, pero además permite ofrecer servicios como un pool de conexiones.
- Es necesario una **gestión de hilos**, ya que será necesario controlar la situación en la que tenemos **una** instancia de un componente (por ejemplo, un servlet) que da respuesta a **varias** peticiones, donde cada petición se resuelve en un **hilo**.

Un ejemplo de servlet

Para empezar a familiarizarnos con lo que es un servlet vamos a ver un pequeño ejemplo en el que se crea una página que contiene el típico "Hola mundo":

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class Saludo extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
```



```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head><title>Bienvenido al primer servlet</title></head>");
out.println("<body bgcolor=\"#FFFF9D\"><FONT color=\"#000080\" " +
    "FACE=\"Arial,Helvetica,Times\" SIZE=2>" +
    "<CENTER><H3>Servlets</H3></CENTER>");
Date d = new Date();
out.println("<HR><p>iHola Mundo!. Fecha y hora: " + d.toString() + ". Esto es
Java.</p>");
    out.println("</body></font></html>");
}

}
```

La mayoría de servlets heredan de HttpServlet. La JVM del servidor de aplicaciones recibe una solicitud, para dar la respuesta realiza una serie de tareas:

- **Un hilo para cada solicitud.** Normalmente se cumple que una solicitud corresponde con un hilo y todos los hilos comparten una **única instancia** del servlet (a menos que nuestra clase servlet herede de **SingleThreadModel**, que no es lo habitual). La JVM traduce la solicitud en la creación de un hilo, sobre el que se realiza una llamada a:
 - Método **service()**, heredado de HttpServlet.
 - Si no se sobreescribe el método service(), invocará al método correspondiente (si el método de invocación al servlet es GET, entonces la JVM llamará al método **doGet()** de nuestro servlet, si el método de invocación es POST la JVM llamará a **doPost()**, etc.). Más adelante veremos la diferencia de uno u otro.
- Ya hemos visto que la JVM invoca al método de entrada (doGet(), doPost(), etc). Pero además **debe transferir al método de entrada dos objetos**: uno es de la clase **HttpServletRequest**, que encapsula información diversa de la petición Http (cabecera, parámetros, etc.) y otro encapsula el stream o flujo Http de respuesta, de la clase **HttpServletResponse**.

Esto sólo es un pequeño antílope de los que es un Servlet o un JSP. Los detalles del ciclo de vida de estas clase/páginas vendrán más adelante.

En el siguiente ejemplo de formulario en HTML podemos ver como se invoca al servlet anterior:

```
<form action="../../servlet/hola_mundo" method="get" target=_blank >  
<p>  
    <input type="submit" name="Submit"  
    value="Pulse aqui para llamar al servlet que saluda">  
</p>  
</form>
```

En la parte '**action**' del formulario indicamos la URL o URI del recurso (en nuestro ejemplo es el servlet de saludo). Con el atributo '**method**' indicamos el método de invocación. Con este método los parámetros de la solicitud aparecen explícitos, es decir, se incluyen en la URL de solicitud. Sin embargo con el método POST los parámetros quedan ocultos al cliente, ya que se transmiten en el cuerpo de la solicitud. El botón es el tipo 'submit', lo que indica que su pulsación disparará la acción (solicitud de recurso).

LAS CAPAS DE LA ARQUITECTURA

En la arquitectura JEE se contemplan cuatro capas, en función del tipo de servicio y contenedores:

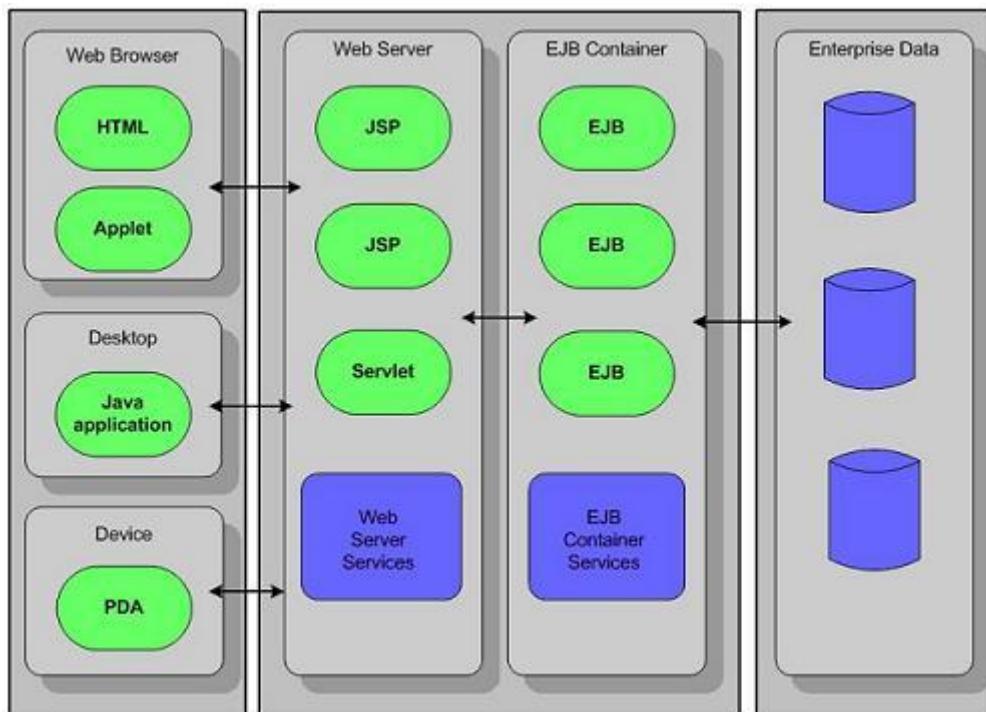


Figura 18: Capas de la arquitectura Java EE

- Capa de **cliente**, también conocida como capa de presentación o de aplicación. Nos encontramos con componentes Java (applets o aplicaciones) y no-Java (HTML, JavaScript, etc.).
- Capa **Web**. Intermediario entre el cliente y otras capas. Sus componentes principales son los servlets y las JSP. Aunque componentes de capa cliente (applets o aplicaciones) pueden acceder directamente a la capa EJB, lo normal es que Los servlets/JSPs puedan llamar a los EJB.
- Capa **Enterprise JavaBeans**. Permite a múltiples aplicaciones tener acceso de forma concurrente a datos y lógica de negocio. Los EJB se encuentran en un servidor EJB, que no es más que un **servidor de objetos distribuidos**. Un EJB puede conectarse a cualquier capa, aunque su misión esencial es conectarse con los sistemas de información empresarial (un gestor de base de datos, ERP, etc.)
- Capa de sistemas de información empresarial.

La visión de la arquitectura es un **esquema lógico**, no físico. Cuando hablamos de capas nos referimos sobre todo a **servicios** diferentes (que pueden estar físicamente dentro de la misma máquina e incluso compartir servidor de aplicaciones y JVM)

A continuación veremos algunos de los **diversos escenarios de aplicación** de esta arquitectura.

Escenario desde un navegador

Es el escenario canónico, donde aparecen todas las capas, empezando en un navegador HTML/XML. La generación de contenidos dinámicos se realiza normalmente en páginas JSP. La capa EJB nos permite desacoplar el acceso a datos EIS de la interacción final con el usuario que se produce en las páginas HTML y JSP:

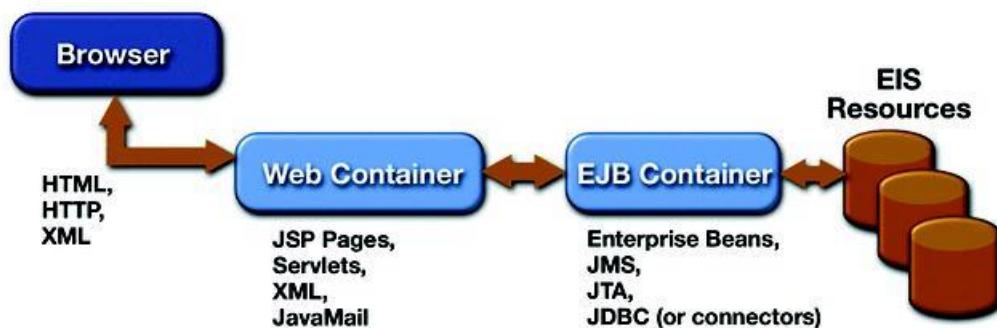


Figura 19: Escenario de un navegador

XML utiliza HTTP como su soporte para el trasnporte.

Una pregunta muy común es **cuando usar servlets y cuando usar páginas JSP**. La pregunta es lógica, al fin y al cabo ambos mecanismos permiten generar contenidos dinámicos y además las JSP son servlets generados por el servidor de aplicaciones. La norma es que la mayor parte de las interacciones con el usuario se realizarán en las JSP debido a su flexibilidad, ya que integran de forma natural etiquetas HTML, XML, JSF, etc. Los servlets serán la excepción (un ejemplo típico es usar un servlet como controlador (un controlador recibe peticiones o eventos desde el interfaz de cliente y "sabe" el componente que debe invocar)).

Escenario desde una aplicación

Podemos considerar que tenemos como cliente una aplicación stand-alone, que puede ser una aplicación Java o incluso un programa en Visual Basic. La aplicación puede acceder directamente a la capa EJB o a la base de datos del EIS (esto último por medio de JDBC):

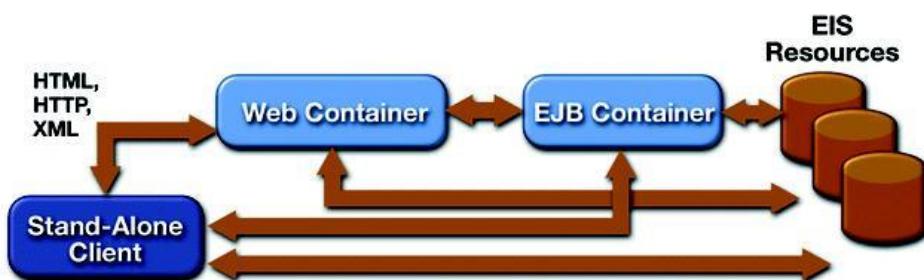


Figura 20: Escenario de una aplicación

Escenario basado en la web (web-centric application)

La plataforma JEE no obliga a usar en un sistema todas las capas. Lo esencial es escoger el mecanismo adecuado para el problema. En este sentido, en ocasiones no hay (ni prevemos que haya) la complejidad como para requerir una capa EJB. Se denomina escenario web-centric porque el contenedor web es el que realiza gran parte del trabajo del sistema:

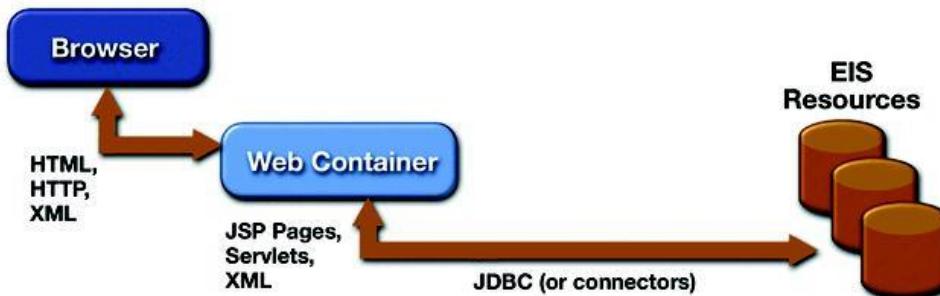


Figura 21: Escenario basado en web

En este tipo de escenario la capa web implica tanto lógica de presentación como lógica de negocio. Pero lo deseable es no mezclar todas las cosas, planteando un diseño limpio y modular. Para ello las JSP y servlets no suelen acceder de forma directa a la base de datos, sino que lo hacen por medio de un servicio de acceso a datos.

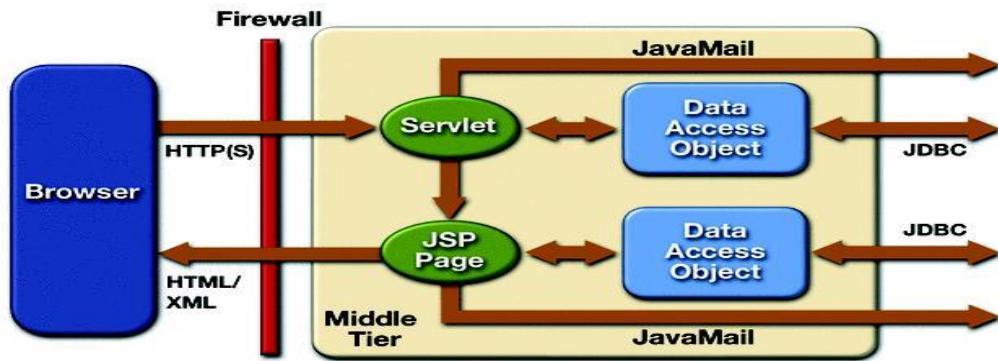


Figura 22: Separando la capa web

El escenario web-centric es el más ampliamente utilizado actualmente.



Capítulo 3

Introducción a Java EE

INTRODUCCIÓN

¿Qué es java EE?

- Java Enterprise Edition (JEE) es una plataforma empresarial que define el estándar para el desarrollo de aplicaciones empresariales distribuidas, basadas en componentes, utilizando un modelo de múltiples capas.
- Orientado a la construcción de sistemas empresariales.
- Proporciona
 - Un modelo de componentes
 - Un modelo estandarizado
 - Un conjunto de servicios estandarizados
- Facilitan la tarea de construir una aplicación empresarial
- Entre sus ventajas se encuentran la gran robustez, fiabilidad, estabilidad y seguridad.

¿Cómo se relaciona con Java SE?

- Basado en Java SE
 - Java Platform, Standard Edition
- No es un reemplazo de Java SE
 - Java SE proporciona el soporte de lenguaje básico sobre el que Java EE ejecuta
 - Es la base de cada componente y servicio construido



¿Para qué Java EE?

- Se requiere una cierta infraestructura para construir este tipo de aplicaciones
- Algunas tareas que deben hacerse:
 - Procesar paquetes HTTP
 - Generar una pagina HTML
 - Conectarse a una base de datos para recuperar y almacenar datos
 - Manejar transacciones
 - Seguridad
 - Procesar archivos XML
 - Enviar mail
 - etc

Modelo de aplicación Java EE

Si pensamos en una aplicación empresarial típica, encontramos tres capas fundamentales:

- Capa de presentación
 - Deplegar informacion al usuario
 - Recoger informacion del usuario
- Capa de negocios
 - Realiza el procesamiento de negocio de la aplicación
 - Procesos y reglas de negocio
- Capa de acceso a datos
 - Toda aplicación no-trivial, requiere alguna forma de persistencia (leer/almacenar datos)

Las aplicaciones Java EE se suelen distribuir en tres lugares diferentes:

- La máquina cliente
- La máquina servidor

- Java EE Server
- Base de datos o sistemas legados
 - En realidad forma la parte persistente de la aplicación
 - No contiene componentes Java

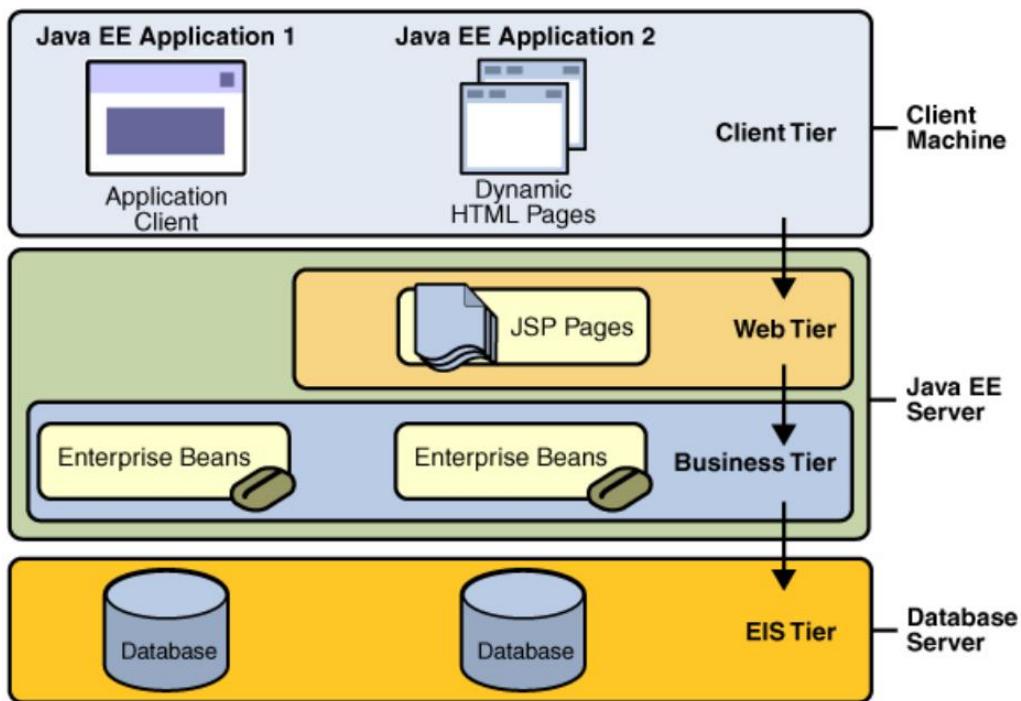


Figura 23: Módelo de una Aplicación Java EE

Independencia del proveedor

- La plataforma promueve la construcción de sistemas independientes de la plataforma
 - Heredado de Java
- La especificación es abierta, puede ser implementada por cualquier proveedor.
- Este deberá cumplir al pie de la letra dicho estándar
 - Hay procesos de certificación
 - No implica que sólo debe soportar lo que el estándar establece
- Un servidor de aplicaciones que sea Java EE certificado, deberá proveer los servicios definidos por la especificación.

- Esta característica, permite que una aplicación ejecutando en un servidor de aplicaciones X, puede ejecutarse también en un servidor de aplicación Y.

CONCEPTOS DE JAVA EE

Clients Java EE

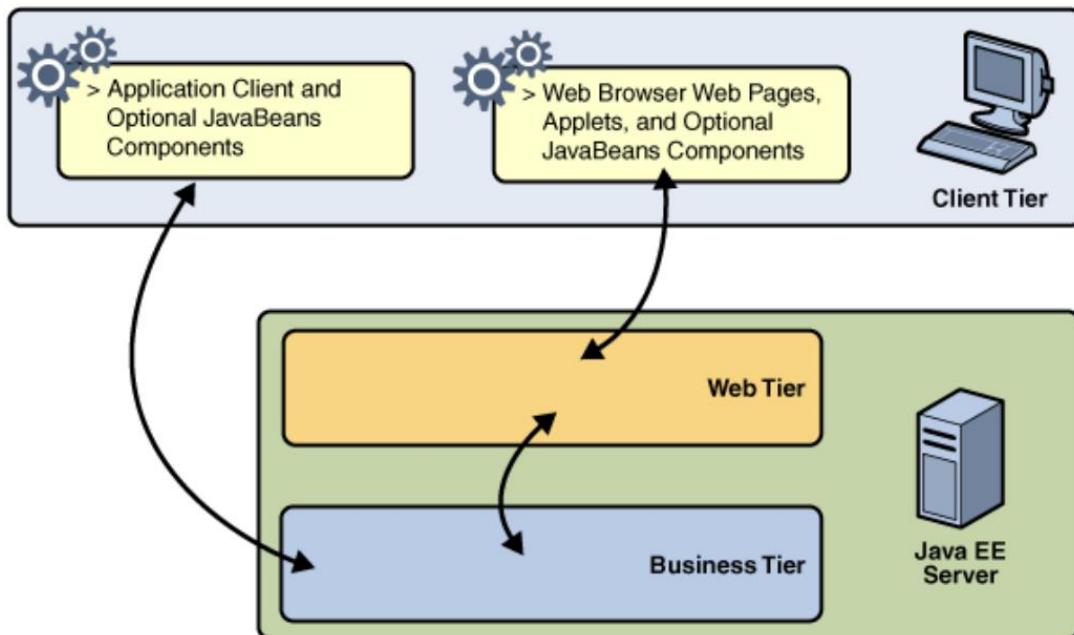


Figura 24: Clientes Java EE

- Web clients
 - Código HTML generado dinámicamente por un servidor de aplicaciones
 - Un navegador web renderiza el HTML devuelto por el componente web
 - Se denominan "clientes livianos"
 - Delegan toda la lógica de negocios, al servidor Java EE
- Applets
 - Aplicación Java (pequeña) que ejecuta en el contexto del cliente, dentro de una JVM embebida en el navegador web.
 - Estos componentes presentan un mecanismo más rico para la construcción de interfaces gráficas.
 - Poco usado en soluciones reales.



➤ Aplicaciones de escritorio

- Ejecutan en el contexto del cliente en forma similar al applet, pero fuera del navegador web.
- Son aplicaciones Java tradicionales
- En general se comunican con los componentes de negocio para ejecutar funcionalidades expuestas por esta capa.

Servidores Java EE

- Representa el ambiente en el que ejecutan los componentes Java EE
- Estos componentes se denominan componentes server-side o componentes de aplicación JEE
- Pueden tomar la forma de:
 - Componentes web (JSP / Servlets / JSF)
 - Componentes de negocio (EJB)
- Estos componentes ejecutan en un runtime denominado contenedor

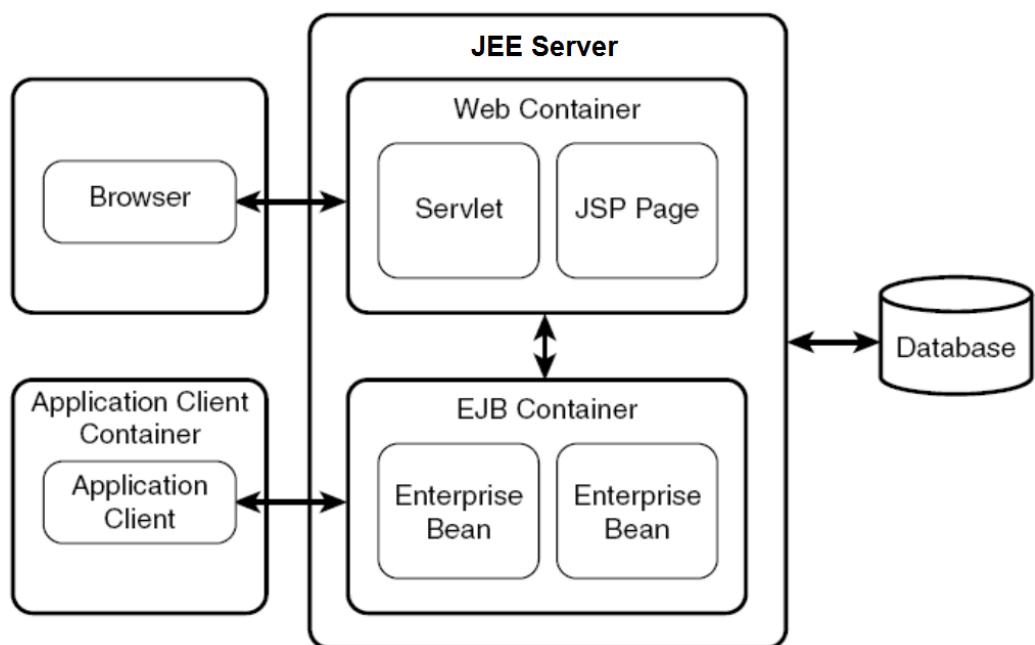


Figura 25: Servidor Java EE

Contenedores Java EE

- Los componentes web y de negocio, existen y se ejecutan dentro de contenedores
- Los componentes de aplicación JEE nunca interactúan directamente entre sí:
 - Utilizan protocolos y métodos del contenedor para interactuar entre ellos y con servicios de la plataforma
 - Este rol de intermediario le permite al contenedor injectar servicios requeridos por los componentes
- Un contenedor permite a los componentes interactuar con los servicios brindados por el servidor de aplicaciones
 - Seguridad
 - Acceso a datos
 - Transacciones
 - Acceso a recursos
 - Comunicaciones

Componentes Java EE

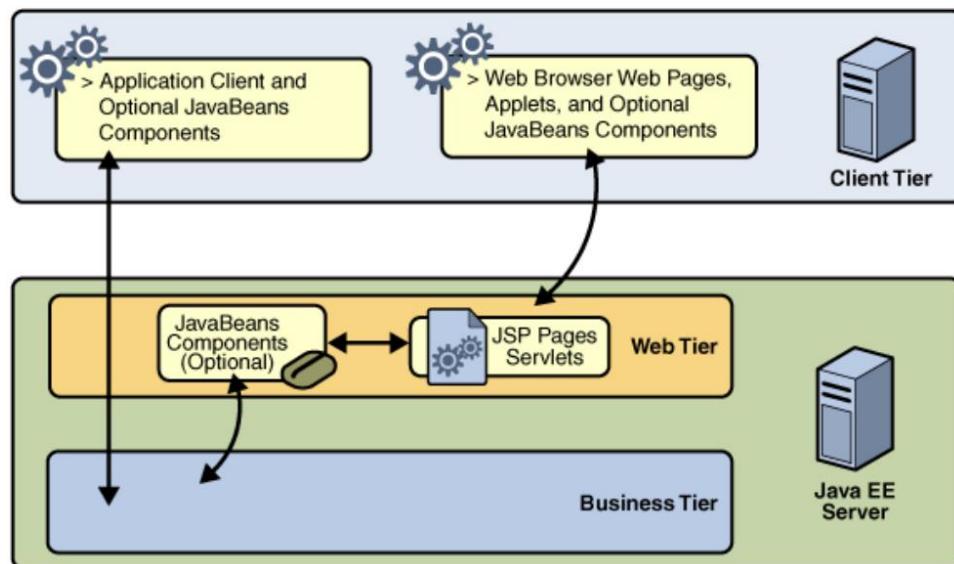


Figura 26: Componentes Web

La plataforma Java brinda dos tipos de componentes.

➤ Componentes web

- Especializados en la generación de contenido HTML dinámico
- Pensados para implementar la capa de presentación de una aplicación empresarial
- Servlets / JSP / JSF

➤ Componentes de negocio

- Especializados en la implementación de lógica de negocio y acceso a datos persistentes en forma relacional
- Modelo de ejecución sincrónico y asincrónico
- Session beans / Message driven beans
- Java Persistence API

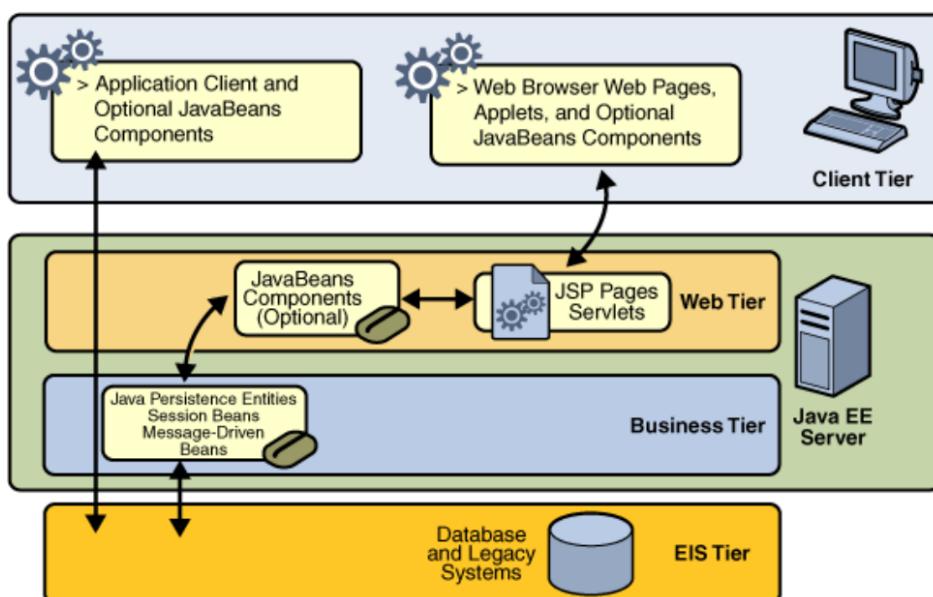


Figura 27: Componentes de Negocio

APIs de Java EE

➤ Java Persistence API

- Object/relational mapper
- Puede ser usado en ambientes Java SE

➤ JDBC (Java DataBase Connectivity) – SE

- API para el acceso a base de datos relacionales

- Pool de conexiones, transacciones distribuidas, etc.
- JTA (Java Transactional API)
- Permite demarcar transacciones
 - Define interfaces entre manejador de transacciones y los recursos manejados

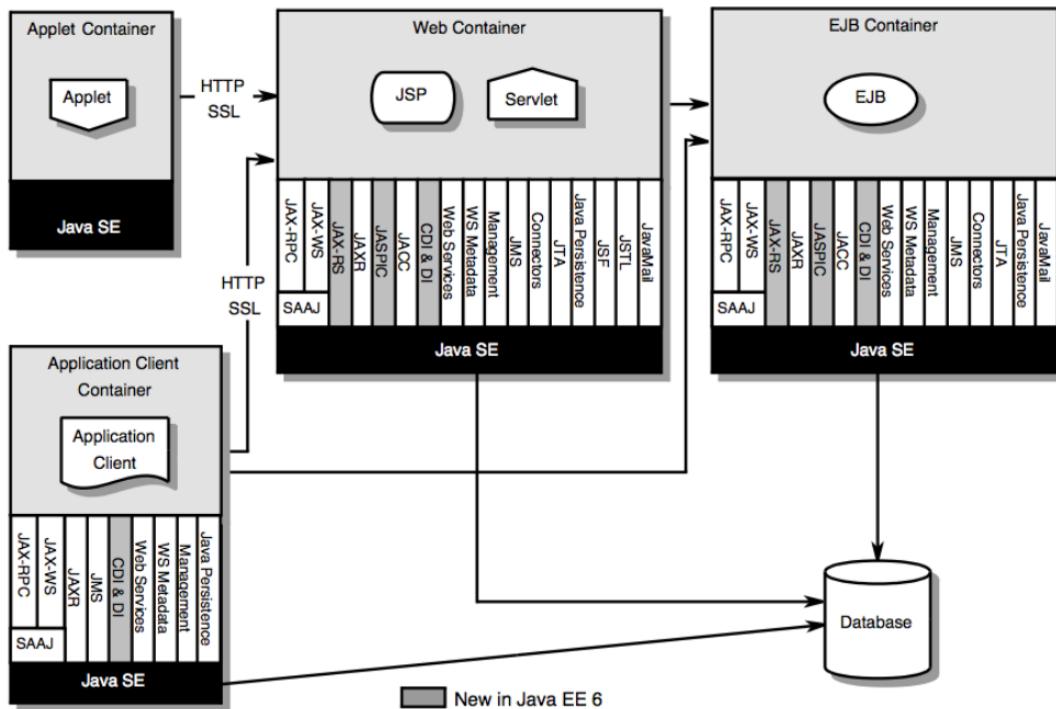


Figura 28: APIs de Java EE

- JNDI (Java Naming and Directory Interface) – SE
- Permite el acceso a servicios de nombres o servicios de directorios
 - Define una interfaz de acceso estándar a los mismos
 - Soporte nativo para LDAP (Lightweight Directory Access Protocol)
- JMS (Java Messaging Service)
- Provee un API estándar para el acceso a MOM (Message Oriented Middleware)
 - Provee dos modelos de mensajería Point-to-Point (Queue) y Publish-Subscriber (Topics)
- JavaMail



- API estándar independiente del protocolo que permite el envío y la recepción de mails
 - Soporta IMAP4, POP3, SMTP
- JCA (Java Connector Architecture)
- Estándar para la integración de servidores de aplicaciones Java EE con Enterprise Information Systems (EIS)
 - Ejemplo de EIS
 - ✓ Mainframes
 - ✓ Sistemas Legados (desarrolladas en lenguajes distintos a Java)
 - ✓ Enterprise Resource Planning (ERP)
 - ✓ Bases de datos
 - El EIS debe proveer un Resource Adapter (JCA compatible)
- Web Services
- Publicar / consumir
 - Java API for XML Web Services (JAX-WS)
 - ✓ Soporta múltiples bindings/protocolos
 - ✓ Soporta REST
 - Java API for XML-based RPC (JAX-RPC)
 - Java Architecture for XML Binding (JAXB)
 - ✓ Define mapping entre XML y clases Java

Capítulo 4

Contenedores Java EE

ENPAQUETADO EN ARCHIVOS WAR

Archivos WAR

- Web Application Archive
- Permiten empaquetar en una sola unidad aplicaciones web java completas.
 - Servlets y JSPs
 - Contenido estático
 - ✓ Html
 - ✓ Imágenes
 - ✓ etc.
 - Otros recursos web
- Son una extensión del archivo JAR
- Se introdujeron en la especificación 2.2 de los servlets.
- Multiplataforma
- MultiVendor

Ventajas

- Simplifican el despliegue de aplicaciones web.
 - Facilidad de instalación
 - Un solo fichero para cada servidor en un cluster.
- Seguridad
 - No permite el acceso entre aplicaciones web distintas

Estructura

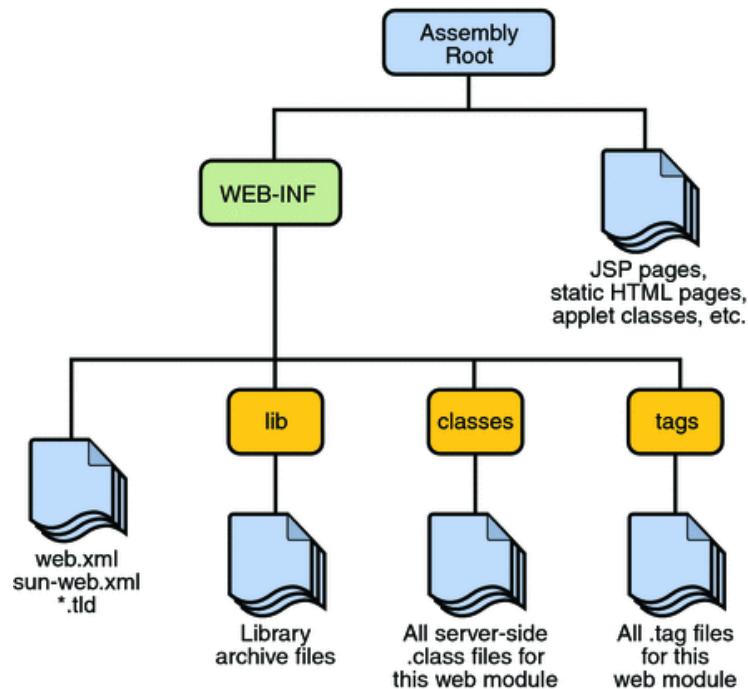


Figura 29: Estructura de un Módulo Web

Descriptor de Despliegue: web.xml

- WEB-INF/web.xml
- Documento XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server
9.0 Servlet 2.5//EN" "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-
0.dtd">
```

- En él se dan de alta
 - Servlets
 - Parámetros del contexto
 - TLDs
 - Filtros
 - Etc.



TOMCAT

Definición

Tomcat (también llamado Jakarta Tomcat o Apache Tomcat) funciona como un contenedor de servlets desarrollado bajo el proyecto Jakarta en la Apache Software Foundation. Tomcat implementa las especificaciones de los servlets y de JavaServer Pages (JSP) de Sun Microsystems.

Estado de su desarrollo

Tomcat es mantenido y desarrollado por miembros de la Apache Software Foundation y voluntarios independientes. Los usuarios disponen de libre acceso a su código fuente y a su forma binaria en los términos establecidos en la Apache Software Licence. Las primeras distribuciones de Tomcat fueron las versiones 3.0.x. Las versiones más recientes son las 6.x, que implementan las especificaciones de Servlet 2.5 y JSP 2.1. A partir de la versión 4.0, Jakarta Tomcat utiliza el contenedor de servlets Catalina.

Entorno

Tomcat es un servidor web con soporte de servlets y JSPs. Tomcat no es un servidor de aplicaciones, como JBoss o JOnAS. Incluye el compilador Jasper, que compila JSPs convirtiéndolas en servlets. El motor de servlets de Tomcat a menudo se presenta en combinación con el servidor web Apache.

Tomcat puede funcionar como servidor web por sí mismo. En sus inicios existió la percepción de que el uso de Tomcat de forma autónoma era sólo recomendable para entornos de desarrollo y entornos con requisitos mínimos de velocidad y gestión de transacciones. Hoy en día ya no existe esa percepción y Tomcat es usado como servidor web autónomo en entornos con alto nivel de tráfico y alta disponibilidad.

Dado que Tomcat fue escrito en Java, funciona en cualquier sistema operativo que disponga de la máquina virtual Java.

Estructura de directorios

La jerarquía de directorios de instalación de Tomcat incluye:

- **bin** - arranque, cierre, y otros scripts y ejecutables
- **common** - clases comunes que pueden utilizar Catalina y las aplicaciones web



- **conf** - ficheros XML y los correspondientes DTD para la configuración de Tomcat
- **logs** - logs de Catalina y de las aplicaciones
- **server** - clases utilizadas solamente por Catalina
- **shared** - clases compartidas por todas las aplicaciones web
- **webapps** - directorio que contiene las aplicaciones web
- **work** - almacenamiento temporal de ficheros y directorios

Características del producto

- Tomcat 3.x (distribución inicial)
- ✓ Implementado a partir de las especificaciones Servlet 2.2 y JSP 1.1
- ✓ Recarga de servlets
- ✓ Funciones básicas HTTP
- Tomcat 4.x
 - Implementado a partir de las especificaciones Servlet 2.3 y JSP 1.2
 - Contenedor de servlets rediseñado como Catalina
 - Motor JSP rediseñado con Jasper
 - Conecotor Coyote
 - Java Management Extensions (JMX), JSP Y administración basada en Struts
- Tomcat 5.x
 - Implementado a partir de las especificaciones Servlet 2.4 y JSP 2.0
 - Recolección de basura reducida
 - Capa envolvente nativa para Windows y Unix para la integración de las plataformas
 - Análisis rápido JSP



- Tomcat 6.x
 - Implementado de Servlet 2.5 y JSP 2.1
 - Soporte para Unified Expression Language 2.1
 - Diseñado para funcionar en Java SE 5.0 y posteriores
 - Soporte para Comet a través de la interfaz CometProcessor
- Tomcat 7.x
 - Implementado de Servlet 3.0, JSP 2.2 y EL 2.2
 - Mejoras para detectar y prevenir "fugas de memoria" en las aplicaciones web
 - Limpieza interna de código
 - Soporte para la inclusión de contenidos externos directamente en una aplicación web
- Tomcat 8.x
 - Tomcat 8 está alineado con Java EE 7
 - Implementado de Servlet 3.1, JSP 2.3 y EL 3.0
 - WebSocket 1.0.
 - Funcionar con la JDK 1.7 o superior

JBOSS

Introducción

JBoss es un servidor de aplicaciones J2EE de código abierto implementado en Java puro. Al estar basado en Java, JBoss puede ser utilizado en cualquier sistema operativo para el que esté disponible Java. Los principales desarrolladores trabajan para una empresa de servicios, JBoss Inc., adquirida por Red Hat en abril del 2006, fundada por Marc Fleury, el creador de la primera versión de JBoss. El proyecto está apoyado por una red mundial de colaboradores. Los ingresos de la empresa están basados en un modelo de negocio de servicios.

JBoss implementa todo el paquete de servicios de J2EE.

Por ejemplo, Los Sims online utilizan JBoss para sus juegos multiusuario.



Servidor de aplicaciones JBoss

JBoss AS es el primer servidor de aplicaciones de código abierto, preparado para la producción y certificado JEE, disponible en el mercado, ofreciendo una plataforma de alto rendimiento para aplicaciones de e-business. Combinando una arquitectura orientada a servicios revolucionaria con una licencia de código abierto, JBoss AS puede ser descargado, utilizado, incrustado y distribuido sin restricciones por la licencia. Por este motivo es la plataforma más popular de middleware para desarrolladores, vendedores independientes de software y, también, para grandes empresas.

Las características destacadas de JBoss incluyen:

- Producto de licencia de código abierto sin coste adicional.
- Cumple los estándares.
- Confiable a nivel de empresa
- Incrustable, orientado a arquitectura de servicios.
- Flexibilidad consistente
- Servicios del middleware para cualquier objeto de Java
- Ayuda profesional 24x7 de la fuente
- Soporte completo para JMX

Capítulo 5

Servlets

¿QUÉ ES UN SERVLET?

Abstracción

Los Servlets son módulos que extienden los servidores orientados a requerimiento/respuesta, como los servidores web compatibles con Java.

Por ejemplo, un servlet podría ser responsable de tomar los datos de un formulario de entrada de pedidos en HTML y aplicarle la lógica de negocios utilizada para actualizar la base de datos de pedidos de una compañía.

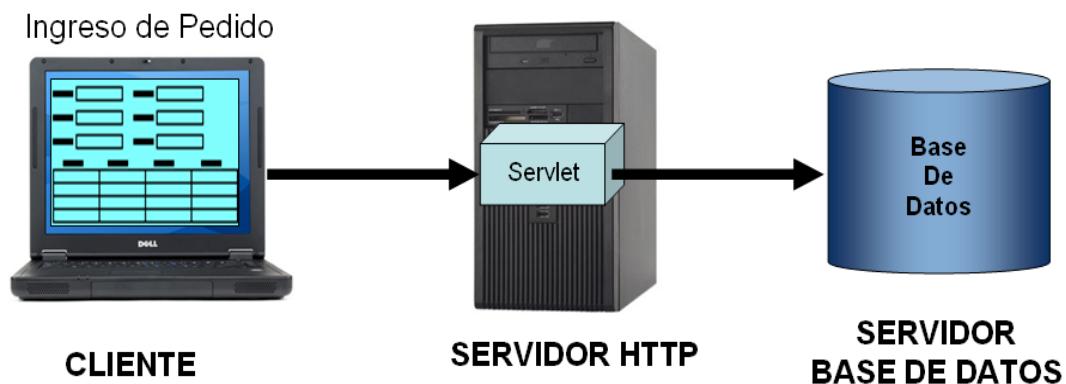


Figura 30: Esquema general del funcionamiento de un servlet

Aspectos Técnicos

Un servlet es un objeto que se ejecuta en un servidor o contenedor JEE, especialmente diseñado para ofrecer contenido dinámico desde un servidor web, generalmente HTML. Otras opciones que permiten generar contenido dinámico son los lenguajes ASP, PHP, JSP (un caso especial de servlet), Ruby y Python. Forman parte de JEE (Java Enterprise Edition), que es una ampliación de JSE (Java Standard Edition).

Un servlet implementa la interfaz `javax.servlet.Servlet` o hereda alguna de las clases más convenientes para un protocolo específico (ej: `javax.servlet.HttpServlet`). Al implementar esta interfaz el servlet es capaz de interpretar los objetos de tipo `HttpServletRequest` y `HttpServletResponse` quienes contienen la información de la página que invocó al servlet.

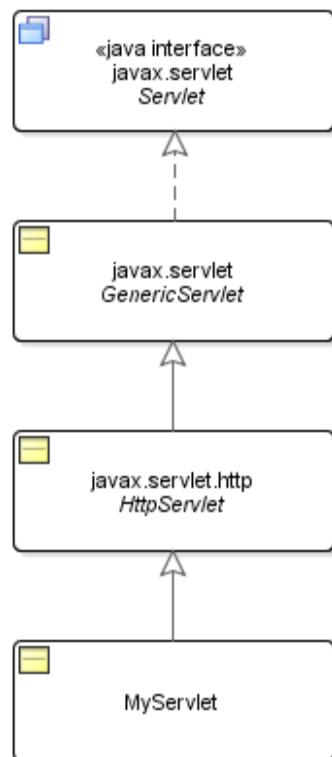


Figura 31: Arquitectura del paquete servlet

INTERACCIÓN CON LOS CLIENTES

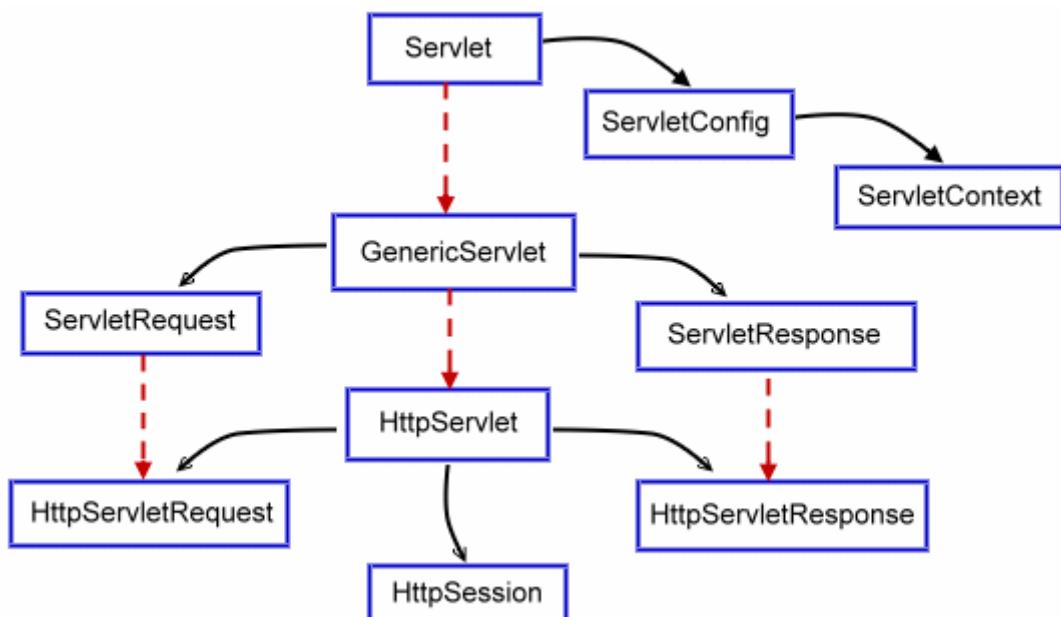


Figura 32: Estructura de funcionamiento de los servlets



Un servlet sencillo

La siguiente clase define completamente un servlet.

```
public class SimpleServlet extends HttpServlet {  
  
    /**  
     * Maneja el método GET de HTPP para construir una sencilla página Web.  
     */  
    @Override  
    public void doGet (HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException{  
        PrintWriter      out;  
        String      title = "Simple Servlet Output";  
  
        // primero selecciona el tipo de contenidos y otros campos de cabecera de la respuesta  
        response.setContentType("text/html");  
  
        // Luego escribe los datos de la respuesta  
        out = response.getWriter();  
        out.println("<HTML><HEAD><TITLE>");  
        out.println(title);  
        out.println("</TITLE></HEAD><BODY>");  
        out.println("<H1>" + title + "</H1>");  
        out.println("<P>This is output from SimpleServlet.");  
        out.println("</BODY></HTML>");  
        out.close();  
    }  
}
```

- *SimpleServlet* extiende la clase *HttpServlet*, que implementa el interface *Servlet*.
- *SimpleServlet* sobreescribe el método *doGet* de la clase *HttpServlet*. Este método es llamado cuando un cliente hace un petición GET (el método de petición por defecto de HTTP), y resulta en una sencilla página HTML devuelta al cliente.
- Dentro del método *doGet*



- La petición del usuario está representada por un objeto *HttpServletRequest*.
- La respuesta al usuario está representada por un objeto *HttpServletResponse*.
- Como el texto es devuelto al cliente, la respuesta se envía utilizando el objeto *Writer* obtenido desde el objeto *HttpServletResponse*.

Interacción con los clientes

Un Servlet HTTP maneja peticiones del cliente a través de su método *service*. Este método soporta peticiones estándar de cliente HTTP despachando cada petición a un método designado para manejar esa petición. Por ejemplo, el método *service* llama al método *doGet* mostrado anteriormente en el ejemplo del servlet sencillo.

Peticiones y respuestas

El objeto que representan peticiones de clientes es de tipo *HttpServletRequest* y las respuestas del servlet es un objeto de tipo *HttpServletResponse*. Estos objetos se proporcionan al método *service* y a los métodos que *service* llama para manejar peticiones HTTP.

Manejar peticiones GET y POST

Los métodos en los que delega el método *service* las peticiones HTTP, incluyen

- *doGet*, para manejar GET, GET condicional, y peticiones de HEAD
- *doPost*, para manejar peticiones POST
- *doPut*, para manejar peticiones PUT
- *doDelete*, para manejar peticiones DELETE

Por defecto, estos métodos devuelven un error BAD_REQUEST (400). Nuestro servlet debería sobrescribir el método o métodos diseñados para manejar las interacciones HTTP que soporta. Esta sección muestra cómo implementar método para manejar las peticiones HTTP más comunes: GET y POST.

El método *service* de *HttpServletRequest* también llama al método *doOptions* cuando el servlet recibe una petición OPTIONS, y a *doTrace* cuando recibe una petición TRACE. La implementación por defecto de *doOptions* determina automáticamente que opciones HTTP son soportadas y devuelve esa información. La implementación por defecto de

doTrace realiza una respuesta con un mensaje que contiene todas las cabeceras enviadas en la petición trace. Estos métodos no se sobreescriben normalmente.

Problemas con los threads

Los Servlets HTTP normalmente pueden servir a múltiples clientes concurrentes. Si los métodos de nuestro Servlet no funcionan con clientes que acceden a recursos compartidos, deberemos:

- Sincronizar el acceso a estos recursos, o
- Crear un servlet que maneje sólo una petición de cliente a la vez.

Descripciones de servlets

Además de manejar peticiones de cliente HTTP, los servlets también son llamados para suministrar descripción de ellos mismos. Para este caso debemos sobreescribiendo el método *getServletInfo*, que suministra una descripción del servlet.

Objeto *HttpServletRequest*

Un objeto *HttpServletRequest* proporciona acceso a los datos de cabecera HTTP, como cualquier cookie encontrada en la petición, y el método HTTP con el que se ha realizado la petición. El objeto *HttpServletRequest* también permite obtener los argumentos que el cliente envía como parte de la petición.

Para acceder a los datos del cliente tenemos:

- El método *getParameter* devuelve el valor de un parámetro nombrado. Si nuestro parámetro pudiera tener más de un valor, deberíamos utilizar *getParameterValues* en su lugar. El método *getParameterValues* devuelve un array de valores del parámetro nombrado. (El método *getParameterNames* proporciona los nombres de los parámetros).
- Para peticiones GET de HTTP, el método *getQueryString* devuelve en un String una línea de datos desde el cliente. Debemos analizar estos datos nosotros mismos para obtener los parámetros y los valores.
- Para peticiones POST, PUT, y DELETE de HTTP:
 - Si esperamos los datos en formato texto, el método *getReader* devuelve un *BufferedReader* utilizado para leer la línea de datos.
 - Si esperamos datos binarios, el método *getInputStream* devuelve un *ServletInputStream* utilizado para leer la línea de datos.



Objeto `HttpServletResponse`.

Un objeto `HttpServletResponse` proporciona dos formas de devolver datos al usuario.

- El método `getWriter` devuelve un `Writer`
- El método `getOutputStream` devuelve un `ServletOutputStream`

Se utiliza el método `getWriter` para devolver datos en formato texto al usuario y el método `getOutputStream` para devolver datos binarios.

Si cerramos el `Writer` o el `ServletOutputStream` después de haber enviado la respuesta, permitimos al servidor saber cuando la respuesta se ha completado.

Cabecera de Datos HTTP

Debemos seleccionar la cabecera de datos HTTP antes de acceder a `Writer` o a `OutputStream`. La clase `HttpServletResponse` proporciona métodos para acceder a los datos de la cabecera. Por ejemplo, el método `setContentType` selecciona el tipo del contenido. (Normalmente esta es la única cabecera que se selecciona manualmente).

El método `service()`

En la declaración se puede notar que el método `service` toma dos parámetros (objetos): uno llamado `request` del tipo `HttpServletRequest` y otro por nombre `response` del tipo `HttpServletResponse`, ambos representan el objeto de entrada y salida del Servlet respectivamente.

Lo anterior significa que dentro de estos objetos reside la información con que será atendida la solicitud así como la información que será enviada de respuesta, por decirlo de otra manera, son estos dos objetos con los que juega todo Servlet.

A continuación se ilustra el Servlet llamado `Saludos.java` con la respectiva lógica en el método `service`:

```
import javax.servlet.*;
import javax.servlet.http.*;

public class Saludos extends HttpServlet {

    @Override
    public void service(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        response.setContentType("text/html");
    }
}
```



```
PrintWriter out = response.getWriter();
out.println("<HTML>");
out.println("<BODY>");
out.println("Pagina Basica");
out.println("</BODY>");
out.println("</HTML>");
}

}
```

La primer linea del método `service` invoca el método `setContentType` sobre el objeto `response`, en otras palabras, se esta indicando que la respuesta de este Servlet será en formato HTML. Posteriormente se indica que el objeto llamado `out` se formará del método `getWriter` del objeto `response`, en otros términos, todo valor asignado a `out` será enviado a la impresión del Servlet (llevada acabo por `getWriter`).

Si no se sobreescribe el método `service`, este por defecto invocara el método `doGet` ó `doPost` según el tipo de petición.

Peticiones GET y POST

Para manejar peticiones HTTP en un servlet, extendemos la clase `HttpServlet` y sobreescrivimos los métodos del servlet que manejan las peticiones HTTP que queremos soportar. En este caso se ilustra el manejo de peticiones GET y POST. Los métodos que manejan estas peticiones son `doGet` y `doPost`.

Manejar peticiones GET

Manejar peticiones GET implica sobreescribir el método `doGet`. El siguiente ejemplo muestra a `BookDetailServlet` haciendo esto.

```
public class BookDetailServlet extends HttpServlet {

    @Override
    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        ...
        // selecciona el tipo de contenido en la cabecera antes de acceder a Writer
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // Luego escribe la respuesta
    }
}
```



```
out.println("<html>" +  
    "<head><title>Book Description</title></head>" +  
    ...);  
  
//Obtiene el identificador del libro a mostrar  
String bookId = request.getParameter("bookId");  
if (bookId != null) {  
    // Y la información sobre el libro y la imprime  
    ...  
}  
out.println("</body></html>");  
out.close();  
}  
...  
}
```

El servlet extiende la clase *HttpServlet* y sobreescribe el método *doGet*. Dentro del método *doGet*, el método *getParameter* obtiene los argumentos esperados por el servlet.

Para responder al cliente, el método *doGet* utiliza un *Writer* del objeto *HttpServletResponse* para devolver datos en formato texto al cliente. Antes de acceder al writer, el ejemplo selecciona la cabecera del tipo del contenido. Al final del método *doGet*, después de haber enviado la respuesta, el *Writer* se cierra.

Manejar peticiones POST

Manejar peticiones POST implica sobreescribir el método *doPost*. El siguiente ejemplo muestra a *ReceiptServlet* haciendo esto.

```
public class ReceiptServlet extends HttpServlet {  
  
    @Override  
    public void doPost(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
        ...  
        // selecciona la cabecera de tipo de contenido antes de acceder a Writer  
        response.setContentType("text/html");  
        PrintWriter out = response.getWriter();  
  
        // Luego escribe la respuesta
```



```
out.println("<html>" +  
    "<head><title> Receipt </title>" +  
    ...);  
  
out.println("<h3>Thank you for purchasing your books from us " +  
    request.getParameter("cardname") +  
    ...);  
out.close();  
}  
...  
}
```

El servlet extiende la clase *HttpServlet* y sobreescribe el método *doPost*. Dentro del método *doPost*, el método *getParameter* obtiene los argumentos esperados por el servlet.

Para responder al cliente, el método *doPost* utiliza un *Writer* del objeto *HttpServletResponse* para devolver datos en formato texto al cliente. Antes de acceder al writer, el ejemplo selecciona la cabecera del tipo de contenido. Al final del método *doPost*, después de haber enviado la respuesta, el *Writer* se cierra.

PROGRAMACIÓN DE SERVLETS

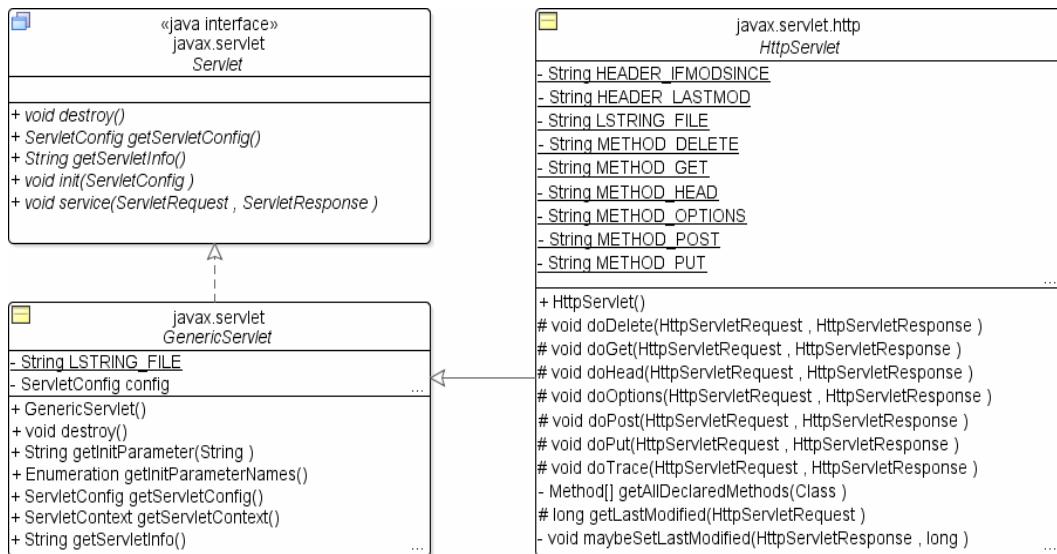


Figura 33: Programación de servlets



void init(ServletConfig config)

Es invocado una sola vez por el contenedor del servidor JEE compatible donde se hospeda el servlet y se emplea para inicializarlo. Se ejecuta cuando se realiza el primer requerimiento del servlet.

void destroy()

Es invocado por el contenedor antes de que el servlet se descargue de memoria y deje de prestar servicio.

void service(ServletRequest request, ServletResponse response)

Es invocado por el contenedor para procesar el requerimiento, una vez que el servlet se ha inicializado. Es el llamado método de servicio. Sus argumentos son instancias de las interfaces *javax.servlet.ServletRequest* y *javax.servlet.ServletResponse* que modelan, respectivamente, el requerimiento del cliente y la respuesta del servlet.

Esquema de Funcionamiento

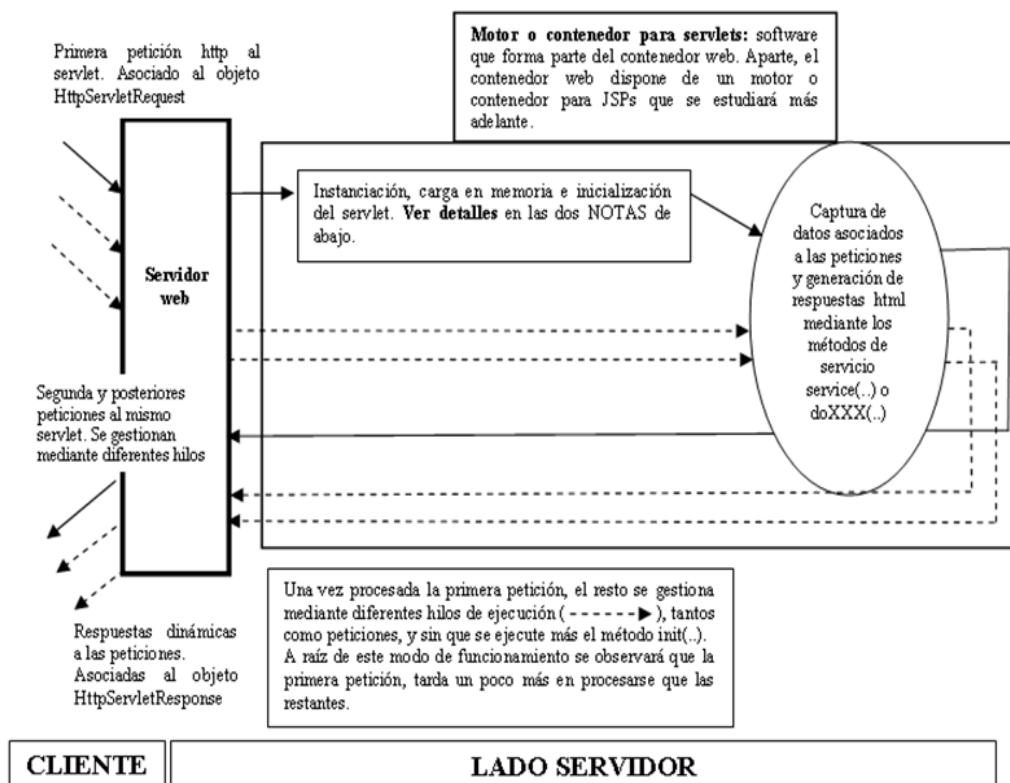


Figura 34: Esquema de funcionamiento de un servlet

- Finalizada la inicialización, el servlet ya está disponible para procesar los requerimientos y generar una respuesta a los mismos, con el método `service(HttpServletRequest request, HttpServletResponse response)`.
- Una vez procesado el primer requerimiento, el resto de requerimientos se gestiona mediante diferentes hilos de ejecución, tantos como requerimientos existan, tal como se puede apreciar en la Figura 34 y sin que se ejecute más el método `init(..)`.

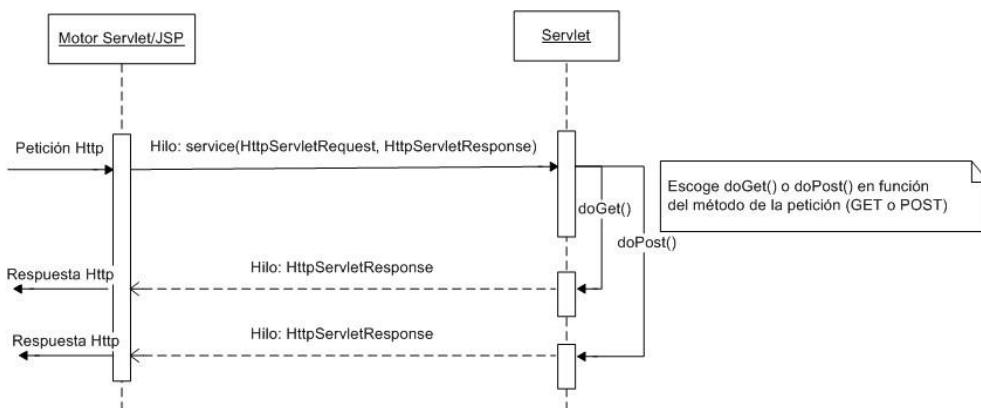


Figura 35: Diagrama de secuencia del funcionamiento de un servlet

INTERACCIÓN CON UN SERVLET

Consideraciones previas

Para hacer referencia a un servlet debemos tener en cuenta como esta mapeado en el descriptor de despliegue (archivo web.xml).

```
<servlet>
    <servlet-name>Empleado</servlet-name>
    <servlet-class>servlets.Empleado</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Empleado</servlet-name>
    <url-pattern>/empleado</url-pattern>
</servlet-mapping>
```

La etiqueta url-pattern representa el alias con que debemos hacer referencia al servlet, normalmente se utiliza el mismo nombre de la clase pero no tiene que ser así.

Si utilizamos anotaciones la definición del servlet sería de la siguiente manera:



```
@WebServlet(name="Empleado", urlPatterns={"/Empleado"})
public class Empleado extends HttpServlet {
    ...
    ...
}
```

Escribiendo la URL del servlet en un Navegador Web

Los servlets pueden ser llamados directamente escribiendo su URL en el campo dirección del navegador Web.

`http://localhost:8080/Proyecto04/empleado?sucursal=001`

Llamar a un Servlet desde dentro de una página HTML

- Si el servlet está en otro servidor, debemos utilizar la URL completa.

En un formulario:

```
<form method="post" action="http://localhost:8080/Proyecto04/empleado">
    ...
    ...
</form>
```

En un enlace:

```
<a href="http://localhost:8080/Proyecto04/empleado?sucursal=001">
    Consultar
</a>
```

- Si el servlet está en la misma aplicación sólo debemos hacer referencia al alias del servlet.

En un formulario:

```
<form method="post" action="empleado">
    ...
    ...
</form>
```

En un enlace:

```
<a href="empleado?sucursal=001">
    Consultar
</a>
```



Llamada a un servlet desde otro servlet

Tenemos dos posibilidades, ejecutar un *sendRedirect()* o un *forward()*, que tienen el mismo objetivo, pero que funcionan diferente.

A continuación tenemos sus diferencias:

- *forward()* se ejecuta completamente en el servidor. Mientras que *sendRedirect()* conlleva a responder con un mensaje HTTP y esperar a que el navegador cliente acuda a la URL especificada. Es por ello que *forward()* es más rápido. Y es por ello que *sendRedirect()* modifica la URL del navegador.
- *forward()* permite llamar a un servlet o página JSP. Por el contrario en *sendRedirect()* se indica una URL que puede ser incluso una URL externa como "http://www.google.com" o cualquier otra.
- En un *forward()* se pasan dos argumentos: *request* y *response*. Esto permite pasar objetos en el scope *request*, por ejemplo. Mientras que en *sendRedirect()* los únicos parámetros que se pueden pasar son los de una URL "...?parametro1=valor1....". Obviamente también se podría usar otro scope, pero no el scope *request*.

Supongamos que tenemos dos servlets de nombre *Datos* y *Respuesta*. A continuación tenemos dos ejemplos, uno utilizando *sendRedirect()* y otro utilizando *forward()*.

- Desde el servlet *Datos* se realiza un *sendRedirect()* al servlet *Respuesta*:

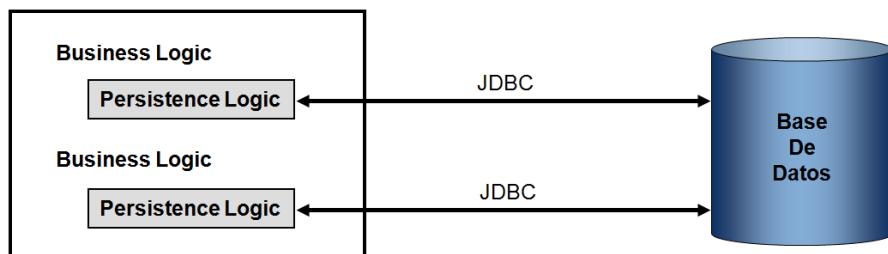
```
response.sendRedirect("Respuesta");
```

- Desde el servlet *Datos* se realiza un *forward()* al servlet *Respuesta*:

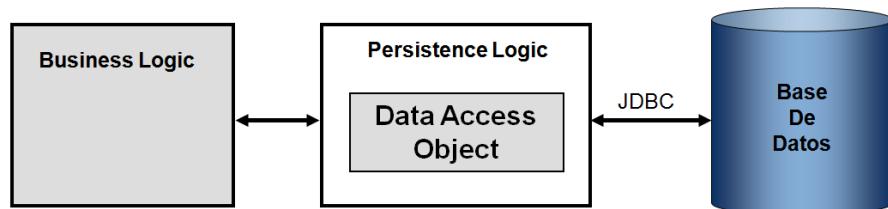
```
RequestDispatcher rd = request.getRequestDispatcher("Respuesta");
rd.forward(request, response);
```



SERVLETS Y JAVABEANS



Antes de DAO: código persistente esparcido dentro del la lógica de negocio



Después de DAO: nueva capa para encapsular la interacción con el almacén de persistencia

Figura 36: Separando la aplicación en capas

No es una buena práctica programar el acceso a la base de datos en los servlets, algunas de las razones son las siguientes:

- Cada servlet estaría consumiendo una conexión a la base de datos.
- Estaríamos juntando en una sola clase la lógica de control de la aplicación, la lógica del negocio y la capa de acceso a la base de datos, y de lo que se trata es de desacoplar estas capas para un mejor control, desarrollo y reutilización de componentes.
- Para la capa de acceso a la fuente de datos deberíamos utilizar el patrón de diseño DAO y si tenemos varias fuentes de datos deberíamos utilizar Factory Patterns, ó quizás un framework ORM.
- También podemos aplicar Factory Patterns a la lógica de negocio.



SESIONES

Introducción

Hay un número de problemas que vienen del hecho de que HTTP es un protocolo "sin estado". En particular, cuando estamos haciendo una compra on-line, es una molestia real que el servidor Web no puede recordar fácilmente transacciones anteriores.

Existen tres soluciones típicas a este problema:

- Cookies.
- Reescribir la URL
- Campos de formulario ocultos

La API de Seguimiento de Sesión

Usar sesiones en servlets es bastante sencillo, la búsqueda del objeto de sesión asociado con el requerimiento actual, crear un nuevo objeto de sesión cuando sea necesario, buscar la información asociada con una sesión, almacenar la información de una sesión, y descartar las sesiones completas o abandonadas.

Objeto HttpSession de la sesión actual

```
HttpSession session = request.getSession(true);
```

Asociar Información con una Sesión

```
HttpSession session = request.getSession(true);
int cant = (Integer) session.getAttribute("A001");
cant += n;
session.setAttribute("A001", cant);
```

Finalizar una Sesión

```
HttpSession session = request.getSession(true);
session.invalidate();
```



FILTROS

Introducción

Hasta ahora sabemos que el contenedor web intercepta las peticiones y respuestas HTTP (no las peticiones EJB). Los filtros son el único mecanismo por el que podemos participar en el mecanismo de intercepción. Podemos introducir filtros que serán llamados antes de que se invoque a otros contenidos, como servlets, páginas html o jsp, etc.

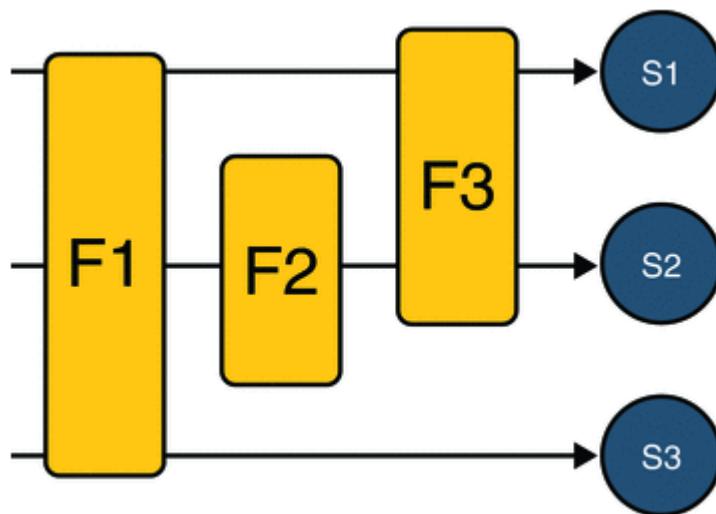


Figura 37: Esquema de funcionamiento de los filtros

El ciclo de vida es similar al de los servlets, también tienen un método **init()** por el que podemos conseguir un `ServletContext`, además tiene el método **doFilter()**, funcionalmente similar a `doGet()` o `doPost`, que recibe las invocaciones al filtro. Además se declaran en el descriptor `web.xml` de forma parecida. Los filtros pueden ser anidados, del mismo modo que un servlet puede invocar a otro.

Algunas utilidades de los filtros:

- Validar peticiones HTTP
- Gestionar el contenido dinámicamente, ya que el filtro puede crear el recurso necesario
- Modificar los objetos `request` y `response`, con lo que creamos un entorno de adaptación a nuestros servlets y JSPs

Hay una ventaja interesante: podemos introducir filtros declarativamente, en `web.xml`, sin tener que modificar los recursos que finalmente se invocan.



Un sencillo ejemplo

En el siguiente filtro vamos a contar las veces que se producen invocaciones a un determinado recurso. El recurso lo definiremos en web.xml. El contador lo almacenaremos como atributo del contexto. Lo que significa que el reinicio del contexto de aplicación, reiniciará el contador.

Al igual que un servlet el inicio del filtro supone una invocación a **init()**. **doFilter()** funciona de manera semejante a **service()** en un servlet. En este ejemplo puede observarse que el filtro debe **implementar el interface Filter**.

```
package docen_servlet01.filtro01;

import javax.servlet.Filter;
import javax.servlet.FilterConfig;
import javax.servlet.FilterChain;
import javax.servlet.ServletContext;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;

/************************************************************************************************
 * Filtro que cuenta las llamadas al recurso filtrado. La cuenta se almacena en
 * atributo de contexto.
 *****/
public class FiltroContador implements Filter {

    FilterConfig config;

    public void init(FilterConfig config) {
        this.config = config;
    }

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        ServletContext context = config.getServletContext();

        //// Obtengo el contador del atributo del contexto
        Integer contador = (Integer)
            context.getAttribute("contador.java.servlets.index");
    }
}
```



```
     //// El rearanque de contexto inicia contador
    if (contador == null) {
        contador = new Integer(0);
    }

    //// Incremento contador y guardo como atributo del contexto
    contador = new Integer(contador.intValue() + 1);
context.setAttribute("contador.java.servlets.index", contador);

    // Invoca al siguiente filtro. Si no lo hay continua con el recurso filtrado
    chain.doFilter(request, response);
}

public void destroy() {}

}
```

Vamos a hacer un servlet que simplemente muestra el contador. Si el filtro no ha sido invocado, entonces devuelve el mensaje "contador no disponible":

```
package docen_servlet01.filtro01;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import docen_servlet01.JDBC01.presentacion.UtilGeneral;

*****
 * Este servlet muestra el contador del filtro, que está almacenado
 * en un atributo del contexto. El contador se reinicia cuando se reinicia el
 * contexto de aplicación.
 * Para las salidas HTML usamos docen_servlet01.JDBC01.presentacion.UtilGeneral
****

public class MostrarContador extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
```



```
response.setContentType("text/html; charset=iso-8859-1"); // Definir tipo de salida
PrintWriter out= response.getWriter(); // Obtener flujo salida;

try {
    UtilGeneral.imprimirInicioPagina( "Ejemplo de servlet",
        "Muestra el contador usado por el filtro", out);
    ServletContext context = getServletContext();
    Integer contador = (Integer) context.getAttribute("contador.java.servlets.index");
    if ( contador != null) {
        UtilGeneral.imprimir( out, "Contador:" + contador.intValue());
    }
    else {
        UtilGeneral.imprimir( out, "Contador no disponible" );
    }
}
catch (Exception e) {
    UtilGeneral.imprimir( out, "Excepción: " + e.getMessage() );
}
UtilGeneral.imprimirFinPagina(out);
}

}
```

Despliegue

En web.xml vamos a indicar el recurso que interceptamos. La forma en la que indicamos el filtro en web.xml es semejante a un servlet. Póngalos antes de la declaración de servlets.

```
<filter>
<filter-name>contador</filter-name>
<display-name>contador</display-name>
<filter-class>docen_servlet01.filtro01.FiltroContador</filter-class>
</filter>

<filter-mapping>
<filter-name>contador</filter-name>
<url-pattern>/java/servlets/index.html</url-pattern>
</filter-mapping>
```



Como era de esperar, **filter-class** hace referencia a la ruta física de la clase. Hay una diferencia respecto a los servlets: **url-pattern** en un servlet es la ruta lógica para invocarlo; en un filtro es la ruta lógica del recurso filtrado o interceptado. **El filtro será invocado inmediatamente antes de que la petición llegue al recurso.** En el ejemplo hemos puesto un archivo html (</java/servlets/index.html>), pero podríamos poner cualquier archivo html (ejemplo: /java/servlets/*.html), también podríamos poner un servlet (por ejemplo, </servlet/FormClientes>). Observe que no hace falta anteponer el contexto de aplicación (no sería /public_html/java/servlets/index.html). Si anida filtros procure ponerlos en web.xml en el orden en que se anidan.

SERVLET PARA CARGA DE ARCHIVO

Introducción

Un problema típico es conseguir que nuestro servidor almacene en el sistema de ficheros o en base de datos un archivo del cliente. El proceso es sencillo, el cliente indica en el navegador (por medio de un parámetro de request) el archivo y a continuación invoca al servlet. El servlet recupera en un stream el contenido y lo vuelca en un archivo o base de datos.

Para el servlet vamos a utilizar una librería del proyecto Jakarta: Jakarta Commons FileUpload package: commons-fileupload-1.2.jar. Se consigue en la página de commons/fileupload del proyecto Jakarta. Además esta librería requiere Jakarta Commons IO librarie: commons-io-1.3.1.jar. Se consigue en la página de commons/io del proyecto Jakarta.

El formulario

Un formulario de ejemplo que invoca a nuestro servlet:

```
<form action="http://localhost:8060/ejerWeb_ServletUpload/servBlob"
enctype="multipart/form-data" method="post">
    NIF: <input type="text" name="NIF">
    <br />
    Curriculum: <input type="file" name="cv" size="70" accept="text/plain;image/jpeg">
    <br /><br />
    <input type="submit" name="enviar" value="Enviar">
</form>
```

El formulario debe incluir:



- enctype="multipart/form-data"
- method="post"
- Campo 'file'

Es necesario especificar en el elemento <input type="file"> el atributo accept. Esta recomendación se hace aunque haya navegadores que transfieren el archivo a pesar de no pertenecer a uno de los tipos especificados en accept, es decir, accept no filtra. Mantenemos la recomendación de usar este atributo porque evita errores en algunas versiones de navegador, en concreto, el no procesar el elemento de tipo "file".

El servlet: init

Empezamos usando el ServletContext para construir el path del directorio donde guardamos el archivo:

```
public class ServletBlob extends HttpServlet {  
  
    String directorioArchivos; // Donde guardaré archivos  
  
    /****** * Obtengo el directorio donde guardaré los archivos ******/  
    public void init(ServletConfig config) throws ServletException {  
        ServletContext sc = config.getServletContext(); // Conseguimos un contexto  
        // El directorio donde estarán los archivos  
        directorioArchivos = sc.getRealPath("/") + "archivos";  
    }  
  
    ...  
}
```

El servlet: manejo de FileItem

Con un formulario multipart tratado con la librería de Jakarta todos los parámetros recibidos son **FileItem** (sean 'simples' o del tipo 'file'). Con **servletFileUpload.parseRequest(request)** recibimos una lista de FileItem:

```
response.setContentType("text/html");  
PrintWriter out = response.getWriter();
```



```
try {  
    // Inicio de pagina  
    out.println("<html>");  
    out.println("<head><title></title></head>");  
    out.println("<body bgcolor=\"#FFFF9D\"><FONT color=\"#000080\" "+  
        "FACE=\"Arial,Helvetica,Times\" SIZE=2>"+  
        "<CENTER><H3>Carga de Blob</H3></CENTER><HR>");  
    out.println( "<p>Directorio de archivos: " + directorioArchivos + "</p>");  
  
    // Si la request es del tipo multipart ...  
    if (ServletFileUpload.isMultipartContent(request)){  
  
        // fileItemsList contendrá una lista de items de archivo  
        // que son instancias de FileItem  
        // Un item de archivo puede contener un archivo para  
        // upload o un campo del formulario  
        // con la estructura simple nombre-valor  
        // (ejemplo: <input name="text_field" type="text" />)  
        ServletFileUpload servletFileUpload =  
            new ServletFileUpload(new DiskFileItemFactory());  
        List fileItemsList = servletFileUpload.parseRequest(request);  
        ...  
    }  
}
```

Por defecto la instancia de `ServletFileUpload` tiene los siguientes valores:

- `Size threshold` = 10,240 bytes. Si el tamaño del archivo está por debajo del umbral, se almacenará en memoria. En otro caso se almacenara en un archivo temporal en disco.
- Tamaño Maximo del cuerpo de la request HTTP = -1. El servidor aceptará cuerpos de request de cualquier tamaño.
- `Repository` = Directorio que el sistema usa para archivos temporales. Se puede recuperar llamando a `System.getProperty("java.io.tmpdir")`.

Podemos cambiar las opciones mediante `setSizeThreshold()` y `setRespository()` de la clase `DiskFileItemFactory` y el método `setSizeMax()` de la clase `ServletFileUpload`, por ejemplo:

```
DiskFileItemFactory diskFileItemFactory = new DiskFileItemFactory();  
diskFileItemFactory.setSizeThreshold(40960); // bytes  
File repositoryPath = new File("/temp");  
  
diskFileItemFactory.setRepository(repositoryPath);
```



```
ServletFileUpload servletFileUpload = new ServletFileUpload(diskFileItemFactory);
servletFileUpload.setSizeMax(81920); // bytes
```

En nuestro ejemplo vamos a trabajar con generalidad: programaremos como si quisieramos leer todos los campos sean 'simples' o 'file'. Por ello iteramos sobre todos los FileItem que recibimos. Los parámetros simples los diferenciaremos de los parámetros 'file' por medio del método **isFormField()**.

El método `getPost()`:

```
// Itero para obtener todos los FileItem
Iterator it = fileItemsList.iterator();
while (it.hasNext()){
    FileItem fileItem = (FileItem)it.next();
    // El FileItem es un campo simple, del tipo nombre-valor
    if (fileItem.isFormField()){
        String nombre = fileItem.getFieldName();
        String valor = fileItem.getString();
        out.println( "<p>Parámetro:" + nombre + " Valor:" + valor + "</p>");
    }
    // El FileItem contiene un archivo para upload
    else{

        // Atributo "name" del elemento input type="file"
        String nombreCampo = fileItem.getFieldName();

        // Tamaño de archivo en bytes
        long tamanoArchivo = fileItem.getSize();
        // Nombre del archivo en el cliente. Algunos navegadores (por ej. IE 6)
        // incluyen el path completo, lo que puede implicar separar path
        // de nombre.
        String nombreArchivo = fileItem.getName();

        // Content type (tipo MIME) del archivo.
        // Esta información la proporciona el navegador del cliente.
        // Algunos ejemplos: .jpg = image/jpeg, .txt = text/plain
        String contentType = fileItem.getContentType();

        // Obtengo características de campo y archivo
        out.println( "<p>--> Name:" + nombreCampo + "</p>");
        out.println( "<p>--> Tamaño archivo:" + tamanoArchivo + "</p>");
```



```
out.println( "<p>--> Nombre archivo del cliente:" + nombreArchivo + "</p>");  
out.println( "<p>--> contentType:" + contentType + "</p>");  
// Obtengo extensión del archivo de cliente  
String extension = nombreArchivo.substring(nombreArchivo.indexOf("."));  
out.println( "<p>--> Extensión del archivo:" + extension + "</p>");  
  
// Guardo archivo del cliente en servidor, con un nombre 'fijo' y la  
// extensión que me manda el cliente  
File archivo = new File(directorioArchivos + "/cv" + extension);  
fileItem.write(archivo);  
if ( archivo.exists() )  
    out.println( "<p>--> GUARDADO " + archivo.getAbsolutePath() + "</p>");  
else  
    out.println( "<p>--> FALLO AL GUARDAR. NO EXISTE " +  
    archivo.getAbsolutePath() + "</p>");  
  
} ///// FIN: es un archivo para upload  
} ///// FIN: iteración de FileItems  
} ///// FIN: la request es del tipo multipart  
}  
catch (Exception e) {  
    e.printStackTrace(out);  
    e.printStackTrace();  
}  
finPagina(out);  
}
```

El nombre del archivo (para el cliente) se consigue por medio de **fileItem.getName()**. Este método devuelve en algunos navegadores (por ejemplo, IE 6) la ruta completa que el archivo tenía en el puesto cliente. A partir de aquí se obtiene la extensión del archivo.

Creamos un File, a partir de tres datos que ya tenemos: el directorio (directorioArchivos), el nombre del archivo ("cv") y la extensión. **El punto esencial de la carga es ordenar al FileItem que escriba su contenido en el archivo, por medio de fileItem.write(archivo)**. Terminamos comprobando que el archivo existe usando el método **exists()**.

¿Qué ocurre cuando pulso "Enviar" en el formulario, sin haber indicado archivo.? Ocurre lo mismo que si el archivo estuviera vacío: **su tamaño es cero**. Ya tiene una pequeña mejora que hacer con este ejemplo.



El servlet: tratar el contenido del campo 'file'

Si no se quiere sólo guardar el archivo, sino que además se quiere procesar, podemos usar:

```
byte[] fileData = fileItem.get();
```

Pero si el archivo es demasiado grande, puede interesar cargarlo en memoria como un stream por medio de: `InputStream fileStream = fileItem.getInputStream();` (`InputStream` está en `java.io`)

Consideraciones de seguridad

En PHP y JSP los usuarios maliciosos pueden dar un nombre de archivo que modifique archivos del servidor, normalmente señalando rutas relativas dentro del servidor. Por ejemplo: `../../password/password.dat`, lo cual puede modificar un archivo de passwords. Soluciones:

- Asegurarse que los archivos son del tipo que queremos procesar, mediante `getContentType()`.
- Especialmente importante es no guardar el archivo con el mismo nombre que tiene para el cliente, sino con uno especificado por la aplicación.
- Controlar que el tamaño no es demasiado grande o pequeño.
- Soluciones propias del servidor: permisos de acceso a archivos, etc.



Capítulo 6

Java Server Page

¿QUÉ ES JAVA SERVER PAGES?

Definición

La tecnología Java Server Pages (JSP) aparece para aliviar lo difícil que supone generar páginas web mediante servlets.

JSP es una tecnología que nos permite mezclar HTML estático con HTML generado dinámicamente.

JSP es una especificación de Sun Microsystems. Sirve para crear y gestionar páginas Web dinámicas. Permite mezclar en una página código HTML para generar la parte estática, con contenido dinámico generado a partir de marcas especiales.

El contenido dinámico se obtiene, en esencia, gracias a la posibilidad de incrustar dentro de la página código Java de diferentes formas. Su objetivo final es separar la interfaz (presentación visual) de la implementación (lógica de ejecución).

A continuación tenemos un ejemplo de una página jsp:

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP - Demo 01</title>
  </head>
  <body>
    <%!
      private int n1 = 15;
      private int n2 = 10;
      private int s;
    %>
    <h4>Suma de Dos N&uacute;meros</h4>
    <%
      s = n1 + n2;
      out.println("n1 = " + n1 + "<br>");
      out.println("n2 = " + n2 + "<br>");
      out.println("suma = " + s + "<br>");
    %>
```

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

Características

La página JSP se convierte en un servlet.

La conversión la realiza el servidor o contenedor JEE, la primera vez que se solicita la página JSP.

Este servlet generado procesa cualquier requerimiento para esa página JSP.

Si se modifica el código de la página JSP, entonces se regenera y recompila automáticamente el servlet y se recarga la próxima vez que sea solicitada.

Ciclo de vida a nivel de documentos

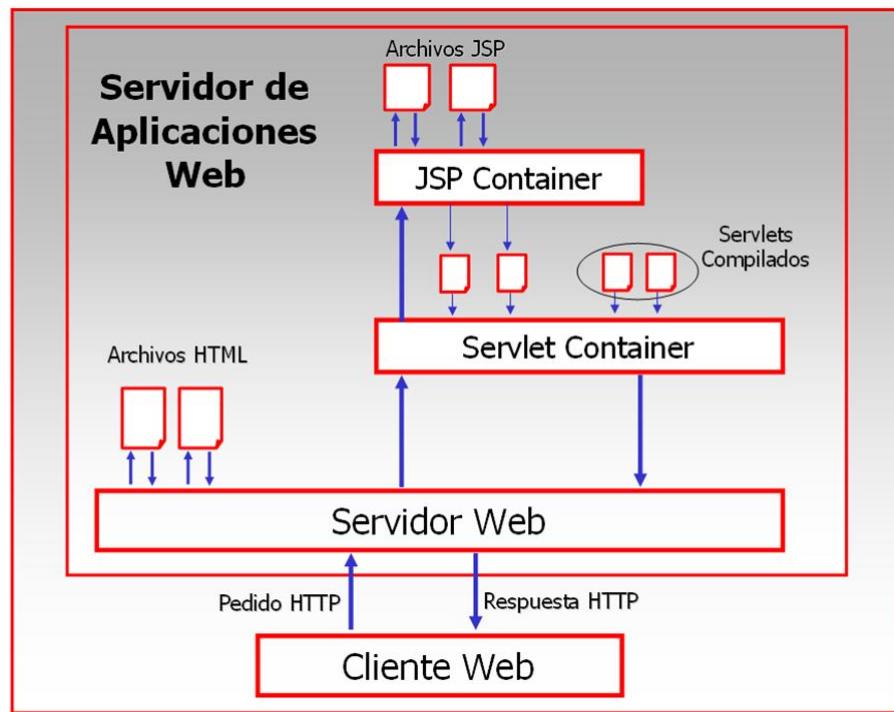


Figura 38: Ciclo de vida a nivel de documentos

Ciclo de vida a nivel de eventos

Primero se ejecuta el método `jspInit()`, que podría ser programado, este método se ejecuta una sola vez, la primera vez que el jsp es requerido, análogamente al método `init()` de un servlet.



La instancia creada para el servlet es también única, y es eliminada por el Garbage Collector del Servlet Engine, pero antes de eliminarse se ejecuta el método `jspDestroy()`, que podría ser también programado.

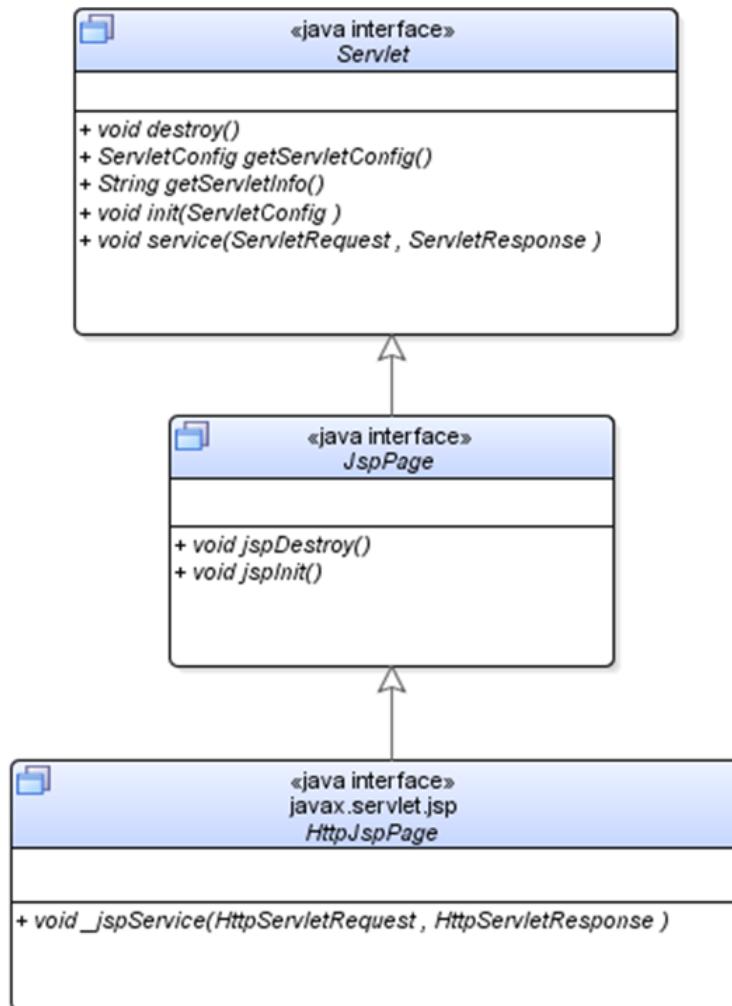


Figura 39: Eventos de una página JSP

ELEMENTOS BÁSICOS

Declaraciones

Sintaxis:

```
<%!
    declaracion;
    [ declaracion; ]
    ...
%>
```

Ejemplo:



```
<%!
    private int cont = 0;
    private int num1, num2, num3;
    private Circulo objCirculo = new Circulo (2.0);
%>
```

Expresiones

Sintaxis 1:

```
<%= expresión_Java %>
```

Sintaxis 2:

```
<jsp:expression>
    Expresión Java
</jsp:expression>
```

Ejemplo:

```
Hora actual: <%= new java.util.Date() %>
```

Scriptlets

Sintaxis 1:

```
<%
    // Código Java
%>
```

Sintaxis 2:

```
<jsp:scriptlet>
    Código Java
</jsp:scriptlet>
```

Ejemplo:

```
if (Math.random() < 0.5) {
    out.write("Este es un <B>agradable</B> día!\n");
} else {
    out.write("Este es un <B>mal</B> día!\n");
}
```



DIRECTIVAS

Introducción

Una directiva JSP afecta a la estructura general de la clase servlet. A continuación tenemos la sintaxis:

```
<%@ directiva atributo="valor" %>
```

Sin embargo, también podemos combinar múltiples atributos para una sola directiva, a continuación tenemos la sintaxis:

```
<%@ directiva atributo1="valor1"
    atributo2="valor2"
    ...
    atributoN="valorN" %>
```

Hay tres tipos principales de directivas:

Directiva	Descripción
include	Se usa para insertar un archivo dentro de la clase servlet, en el momento que la página JSP es traducida a un servlet
page	Se usa para definir atributos que se aplican toda la página jsp y cualquier archivo que se incluya con la directiva include.
taglib	Mediante esta directiva se puede ampliar el conjunto de etiquetas que el interprete de jsp es capaz de entender, de manera que la funcionalidad del mismo sea prácticamente ilimitada,

Directiva: include

Esta directiva nos permite incluir archivos en el momento en que la página JSP es traducida a un servlet.

Sintaxis:

```
<%@ include file="URL relativa" %>
```

Directiva: page

Define atributos que se aplican a una página JSP entera, y los archivos estáticos incluidos con la directiva include.



Sintaxis

En **negrita** tenemos los atributos por defecto.

```
<%@ page
    [ language="java"]
    [ extends="package .class"]
    [ import="{package .class | package.*}, ..."]
    [ session="true|false"]
    [ buffer="none|8kb|sizekb"]
    [ autoFlush="true|false"]
    [ isThreadSafe="true|false"]
    [ info="text" ]
    [ errorPage="relativeURL"]
    [ contentType="{MimeType [;charset=characterSet ]" |
      text/html ; charset=ISO-8859-1}"]
    [ isErrorPage="true|false"]
%>
```

Directiva: taglib

Indica al contenedor JEE que la página jsp utilizará librerías de etiquetas. Estas librerías contienen etiquetas creadas por el propio programador con sus correspondientes atributos que encapsulan determinada funcionalidad. Lo habitual es utilizar librerías públicas que han diseñado otros programadores y han sido profusamente probadas.

Sintaxis:

```
<%@ taglib uri="uriLibreriaEtiquetas" prefix="prefijoEtiqueta" %>
```

El contenido de cada atributo se describe a continuación:

Atributo	Descripción
uri	Permite localizar el archivo descriptor de la librería de etiquetas (TLD, Tag Library Descriptor).
prefix	Especifica el identificador que todas las etiquetas de la librería deben incorporar.



ACCIONES

Las acciones JSP usan sintaxis XML para controlar el comportamiento del motor de Servlets. Podemos insertar un fichero dinámicamente, reutilizar componentes JavaBeans, reenviar al usuario a otra página, o generar HTML.

Hay que tener cuidado con las mayúsculas y las minúsculas ya que el motor es sensible (case sensitive) con respecto a como se escriben las etiquetas.

Acción: <jsp:forward>

Esta acción permite redirigir la ejecución de la página JSP actual hacia otro recurso de forma permanente, si antes de utilizar esta etiqueta ya se ha enviado algún contenido del búfer del flujo de salida del cliente se producirá un error.

Sintaxis:

```
<jsp:forward page="URLLocal"/>
```

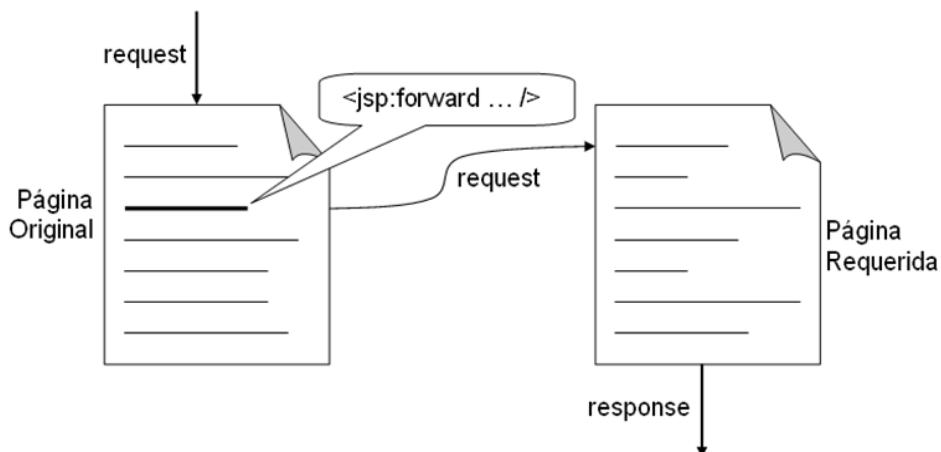


Figura 40: Funcionamiento de la acción asp:forward

Acción: <jsp:param>

Esta acción se utiliza en colaboración con cualquiera de las siguientes acciones: <jsp:forward>, <jsp:include> o <jsp:plugin>.

Sintaxis:

```
<jsp:param name="nombreParametro" value="valorParametro"/>
```

Ejemplo:

```
<jsp:forward page="demo.jsp">
  <jsp:param name="nombre" value="Gustavo"/>
  <jsp:param name="apellido" value="Coronel"/>
</jsp:forward>
```

Acción: <jsp:include>

La acción <jsp:include> permite insertar en el contenido generado por la página actual, el contenido de otro recurso distinto, resultando la salida final que se envía al usuario una combinación de ambos contenidos. Al contrario de lo que ocurría en la acción <jsp:forward>, el control de la ejecución vuelve a la página original una vez que se ha terminado la ejecución de la página incluida.

Sintaxis:

```
<jsp:include page="URLLocal" flush="true|false"/>
```

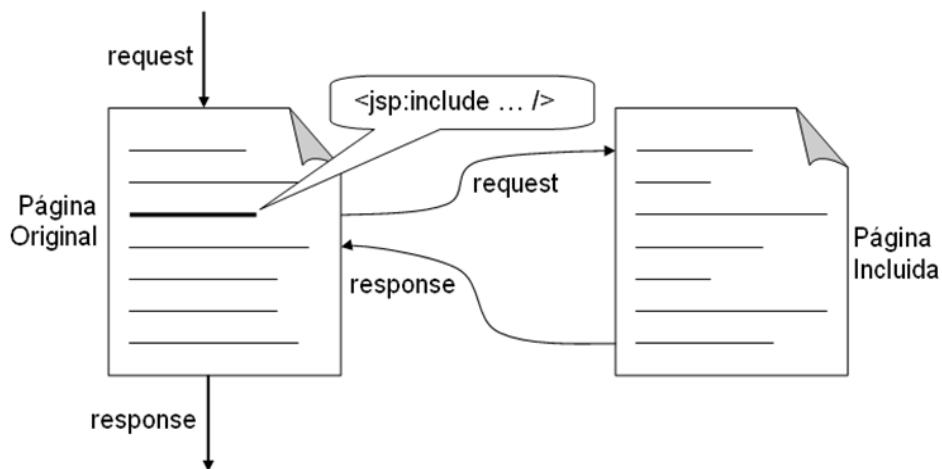


Figura 41: Funcionamiento de la acción jsp:include

Acción: <jsp:useBean>

Esta acción se utiliza para poder utilizar dentro de una página JSP un componente JavaBean en un ámbito determinado. El componente JavaBean podrá ser utilizado dentro de la página JSP haciendo referencia al nombre indicado dentro de la acción <jsp:useBean>, teniendo siempre en cuenta el ámbito al que pertenece el Bean, y que se indica también en la acción <jsp:useBean>.

La acción <jsp:useBean> indica a la página JSP que queremos tener un Bean determinado disponible, el contenedor de páginas JSP creará el Bean correspondiente o bien lo recuperará del ámbito correspondiente.

Sintaxis:

```
<jsp:useBean
    id="beanInstanceName"
    scope="page | request | session | application"
    {
        class="package.class" |
        type="package.class" |
```



```
class="package.class" type="package.class" |
beanName="{package.class | <%= expression %>}"
type="package.class"
}
{
/>
> other elements </jsp:useBean>
}
```

Para localizar o instanciar el Bean, <jsp:useBean> sigue los siguientes pasos, en este orden:

1. El contenedor trata de localizar un objeto que posea el identificador y ámbito indicados en la acción <jsp:useBean>.
2. Si se encuentra el objeto, y se ha especificado un atributo type, el contenedor trata de utilizar la conversión de tipos del objeto encontrado con el tipo (type) especificado, si la conversión falla se lanzará una excepción ClassCastException. Si únicamente se ha indicado un atributo class se creará una nueva referencia al objeto en el ámbito indicado utilizando el identificador correspondiente.
3. Si el objeto no se encuentra en el ámbito especificado y no se ha indicado un atributo class o beanName, se lanzará una excepción InstantiationException.
4. Si el objeto no se ha encontrado en el ámbito indicado, y se ha especificado una clase concreta con un constructor público sin argumentos, se instanciará un objeto de esa clase. Un nuevo objeto se asociará con el identificador y ámbitos indicados. Si no se da alguna de las condiciones comentadas, se lanzará una excepción InstantiationException.
5. Si el objeto no se localiza en el ámbito indicado, y se ha especificado un atributo beanName, se invocará al método instantiate(...) de la clase java.beans.Beans, pasándole como parámetro el valor de la propiedad beanName. Si el método tiene éxito un nuevo objeto se asociará con el identificador y ámbitos indicados.
6. Si la acción <jsp:useBean> no posee un cuerpo vacío, se procesará el cuerpo de la etiqueta, únicamente si el objeto se crea como un objeto nuevo, es decir, no se había encontrado en el ámbito indicado.



Acción: <jsp:getProperty>

Esta acción forma parte de las acciones que nos permiten utilizar componentes Beans dentro de nuestras páginas JSP, en este caso la acción <jsp:getProperty> nos va a permitir obtener el valor de la propiedad de un Bean creado en la página con el ámbito correspondiente.

La sintaxis de esta acción es muy sencilla, no posee cuerpo y únicamente presenta dos atributos, como se puede observar en la sintaxis.

Sintaxis:

```
<jsp:getProperty  
    name="nombreBean"  
    property="nombrePropiedad"/>
```

Acción: <jsp:setProperty>

Esta acción permite modificar las propiedades de los Beans a los que hacemos referencia en nuestras páginas JSP, es la acción complementaria a la acción <jsp:getProperty>.

Sintaxis:

```
<jsp:setProperty  
    name=" nombreBean"  
    {  
        property= "*" |  
        property="nombrePropiedad" [  
            param="nombreParametro" ] |  
        property="nombrePropiedad"  
            value="{cadena | <%= expresión %>}"  
    }  
/>
```

El atributo **name** indica el identificador del Bean que hemos creado con la acción <jsp:useBean>.

Los detalles del atributo **property** son una serie de atributos que combinados entre sí permiten asignar el valor a la propiedad del Bean de distinta forma. Así por ejemplo la forma de establecer el valor de la propiedad de un Bean puede ser cualquiera de las que aparecen a continuación:

```
property="*"  
  
property="nombrePropiedad"
```



```
property="nombrePropiedad" param="nombreParámetro"
```

```
property="nombrePropiedad" value="valorPropiedad"
```

El valor de una propiedad de un Bean se puede establecer a partir de varios elementos:

- En el momento del requerimiento de la página a partir de los parámetros existentes en el objeto integrado request.
- En el momento de ejecución de la página a partir de la evaluación de una expresión válida de JSP.
- A partir de una cadena de caracteres indicada o como una constante en la propia página.

OBJETOS IMPLÍCITOS

Para simplificar el código en expresiones y scriplets, tenemos ocho variables definidas automáticamente, a estas variables también se les denomina "Objetos Implícitos".

Estos objetos corresponden con objetos útiles del API de servlets (request, response, session, ...) y que en realidad son variables instanciadas de manera automática en el servlet generado a partir del JSP.

Objeto: request

Este es el HttpServletRequest asociado con el requerimiento del cliente, y nos permite tener acceso a los parámetros asociados, para lo cual debemos utilizar el método getParameter(..), el tipo de petición (GET, POST, HEAD, etc.), y las cabeceras HTTP entrantes (cookies, Referer, etc.).

Ejemplo:

```
<%
  String nombre = request.getParameter("nombre");
%>
```

Objeto: response

Este es el HttpServletResponse asociado con la respuesta al cliente.



Objeto: out

Este es el PrintWriter usado para enviar la salida al cliente. Sin embargo, para poder hacer útil el objeto response, esta es una versión con buffer de PrintWriter llamada JspWriter.

Ejemplo:

```
Mi Nombre: <% out.println("Gustavo Coronel"); %>
```

Objeto: session

Este es el objeto HttpSession asociado con el requerimiento. Recuerde que las sesiones se crean automáticamente, por lo tanto esta variable es creada incluso si no hubiera una sesión de referencia entrante.

La única excepción es cuando usamos el atributo session de la directiva page para desactivar las sesiones, en cuyo caso los intentos de referenciar la variable session causarán un error en el momento de traducir la página JSP a un servlet.

El objeto session también permite almacenar objetos en forma de atributos dentro del ámbito de la sesión, y por lo tanto presenta los mismos métodos que el objeto request para este propósito.

Ejemplo:

```
<%
    Empleado empleado = new egcc.dao.to.Empleado();
    empleado.setCodigo("12345");
    empleado.setNombre("Gustavo Coronel");
    session.setAttribute( "empleado", empleado );
%>
```

Objeto: application

Este objeto integrado corresponde con el **ServletContext** obtenido mediante **getServletConfig().getContext()** y representa la aplicación Web a la que pertenece la página JSP actual.

El objeto **application** se encuentra disponible en cualquier página JSP y como instancia del interfaz **ServletContext** ofrece todos los métodos de esta interfaz, además de los métodos ya conocidos que se utilizan para almacenar y recuperar los atributos con ámbito de aplicación, no debemos olvidar que el objeto **application** define otro ámbito para los objetos que se almacenan como atributos, y en este caso es el ámbito más general posible.



Objeto: config

Este objeto integrado es una instancia del interfaz **javax.servlet.ServletConfig**, y su función es la de almacenar información de configuración del servlet generado a partir de la página JSP correspondiente.

Normalmente las páginas JSP no suelen interactuar con parámetros de inicialización del servlet generado, por lo tanto el objeto **config** en la práctica no se suele utilizar.

Objeto: pageContext

JSP presenta una nueva clase llamada **PageContext** para encapsular características de uso específicas del servidor.

Permite acceder al espacio de nombres de la página JSP actual, ofrece también acceso a varios atributos de la página así como una capa sobre los detalles de implementación.

Además el objeto **pageContext** ofrece una serie de métodos que permiten obtener el resto de los objetos integrados de JSP, también nos va a permitir acceder a los atributos pertenecientes a distintos ámbitos.

Objeto: page

El objeto **page** es una instancia de la clase **java.lang.Object**, y representa la página JSP actual, o para ser más exactos, una instancia de la clase del servlet generada a partir de la página JSP actual. Se puede utilizar para realizar una llamada a cualquiera de los métodos definidos por la clase del servlet.

Utilizar el objeto **page**, es similar a utilizar la referencia a la clase actual, es decir, la referencia **this**. Este objeto pertenece a la categoría de objetos integrados relacionados con los servlets, en este caso representa una referencia al propio servlet.

En nuestras páginas JSP no es muy común utilizar el objeto **page**.

MANEJO DE ESTADOS

En muchas aplicaciones es normal que haya un conjunto de datos introducidos por el usuario disponibles en más de una página, por ejemplo preferencias de configuración o carritos de la compra deben permanecer y ser accesibles a todas las páginas de la aplicación a las que acceda un determinado usuario, esto es lo que se conoce como mantenimiento del estado. Cualquier aplicación web actual soporta mantenimiento de estado, y quizás más importante, los usuarios esperan que las aplicaciones que van a usar tengan esta funcionalidad, sin mantenimiento de estado, algo tan simple como un login a una aplicación dejaría de funcionar siendo necesario introducir los datos de



usuario cada vez que se accediese a una página de la aplicación. Con mantenimiento de estado estos datos son almacenados y compartidos por todas las páginas que lo necesiten.

Posibles soluciones

- Usar Cookies
- Añadir información en la URL (reescribir url – url rewriting)
- Usar campos ocultos Hidden (<input type="hidden" ... />)
- Usar HttpSession

Cookies

Las cookies son pequeñas piezas de información que se envían con las peticiones (request y response) y que almacenan en cliente datos, almacenando en estas cookies los datos necesarios el servidor podía tener acceso a toda la información necesaria. Sin embargo por motivos de seguridad muchos navegadores restringen el uso de cookies, también son fácilmente interceptables y un uso excesivo conlleva un mayor gasto de ancho de banda.

Algunas actividades más usadas:

- Evitar el nombre de usuario y la password.
- Personalizar una Site.
- Publicidad enfocada.

URL Rewriting

Usando URL Rewriting lo que se hacía era modificar las URLs de las peticiones para que incluyeran como parámetros GET los datos del estado del cliente (por ejemplo los datos de usuario), de esta forma el servidor tenía un fácil acceso a ellos, en contrapartida esos datos eran fácilmente interceptables por terceros y simplemente con copiar la URL de las peticiones (almacenada por ejemplo en un historial) se podía sobrepasar la seguridad de muchos sitios aunque no fuéramos el usuario deseado.

Crear un cookie

Un cookie almacenado en el ordenador de un usuario está compuesto por un nombre y un valor asociado al mismo. Además, asociada a este cookie pueden existir una serie de atributos que definen datos como su tiempo de vida, alcance, dominio, etc.



Cabe reseñar que los cookies, no son más que ficheros de texto, que no pueden superar un tamaño de 4Kb, además los navegadores tan sólo pueden aceptar 20 cookies de un mismo servidor web (300 cookies en total).

Para crear un objeto de tipo Cookie se utiliza el constructor de la clase *Cookie* que requiere su nombre y el valor a guardar. El siguiente ejemplo crearía un objeto Cookie que contiene el nombre "nombre" y el valor "objetos".

```
Cookie miCookie=new Cookie("nombre","objetos");
```

Una vez que se ha creado un cookie, es necesario establecer una serie de atributos para poder ser utilizado. El primero de esos atributos es el que se conoce como tiempo de vida.

Por defecto, cuando creamos un cookie, se mantiene mientras dura la ejecución del navegador. Si el usuario cierra el navegador, los cookies que no tengan establecido un tiempo de vida serán destruidos.

Por tanto, si se quiere que un cookie dure más tiempo y esté disponible para otras situaciones es necesario establecer un valor de tiempo (en segundos) que será la duración o tiempo de vida del cookie. Para establecer este atributo se utiliza el método *setMaxAge()*. El siguiente ejemplo establece un tiempo de 31 días de vida para el cookie "miCookie":

```
miCookie.setMaxAge(60*60*24*31);
```

Existe un método dentro de la clase *Cookie* que permite establecer el dominio desde el cual se ha generado el cookie. Este método tiene su significado porque un navegador sólo envía al servidor los cookies que coinciden con el dominio del servidor que los envió. Si en alguna ocasión se requiere que estén disponibles desde otros subdominios se especifica con el método *setDomain()*. Por ejemplo, si existe el servidor web en la página www.paginasjsp.com, pero al mismo tiempo también existen otros subdominios como usuario1.paginasjsp.com, usuario2.paginasjsp.com, etc.

Si no se establece la propiedad domain se entiende que el cookie será visto sólo desde el dominio que lo creó, pero sin embargo si se especifica un nombre de dominio se entenderá que el cookie será visto en aquellos dominios que contengan el nombre especificado. En el siguiente ejemplo hace que el cookie definido en el objeto "miCookie" esté disponible para todos los dominios que contengan el nombre ".paginasjsp.com". Un nombre de dominio debe comenzar por un punto.

```
miCookie.setDomain(".paginasjsp.com");
```

Igualmente, para conocer el dominio sobre el que actúa el cookie, basta con utilizar el método *getDomain()* para obtener esa información. Una vez que se ha creado el objeto Cookie, y se ha establecido todos los atributos necesarios es el momento de



crear realmente, ya que hasta ahora sólo se tenía un objeto que representa ese cookie.

Para crear el fichero cookie real, se utiliza el método `addCookie()` de la interfaz `HttpServletResponse`:

```
response.addCookie(miCookie);
```

Una vez ejecutada esta línea es cuando el cookie existe en el disco del cliente que ha accedido a la página JSP. Es importante señalar que si no se ejecuta esta última línea la cookie no habrá sido grabada en el disco, y por lo tanto, cualquier aplicación o página que espere encontrar dicho cookie no lo encontrará.

Recuperar un cookie

El proceso de recuperar un cookie determinado puede parecer algo complejo, ya que no hay una forma de poder acceder a una cookie de forma directa. Por este motivo es necesario recoger todos los cookies que existen hasta ese momento e ir buscando aquél que se quiera, y que al menos, se conoce su nombre.

Para recoger todos los cookies que tenga el usuario guardados se crea un array de tipo `Cookie`, y se utiliza el método `getCookies()` de la interfaz `HttpServletRequest` para recuperarlos:

```
Cookie [] todosLosCookies=request.getCookies();

/* El siguiente paso es crear un bucle que vaya leyendo todos los cookies. */

for(int i=0;i<todosLosCookies.length;i++) {

    Cookie unCookie=todosLosCookies[i];

    /* A continuación se compara los nombres de cada uno de los cookies con el que
       se está buscando. Si se encuentra una cookie con ese nombre se ha dado con el
       que se está buscando, de forma que se sale del bucle mediante break. */

    if(unCookie.getName().equals("nombre"))
        break;
}

/* Una vez localizado tan sólo queda utilizar los métodos apropiados
   para obtener la información necesaria que contiene. */

out.println("Nombre: "+unCookie.getName()+"<BR>");
out.println("Valor: "+unCookie.getValue()+"<BR>");
```



```
out.println("Path: "+unCookie.getPath()+"<BR>");  
out.println("Tiempo de vida:"+unCookie.getMaxAge()+"<BR>");  
out.println("Dominio: "+unCookie.getDomain()+"<BR>");
```

Utilizar los cookies

Para realizar un ejemplo práctico se vera un ejemplo de Sesiones. El objetivo será modificar las páginas necesarias para que si el usuario selecciona un campo de tipo checkbox (que será necesario añadir) el nombre de usuario le aparezca por defecto cuando vuelva a entrar a esa página. Este nombre de usuario estará guardado en un cookie en su ordenador.

El primer paso es añadir el checkbox en la página login.jsp:

```
<%@ page session="true" import="java.util.*"%>  
<%  
    String usuario = "";  
    String fechaUltimoAcceso = "";  
    /*Búsqueda del posible cookie si existe para recuperar  
    su valor y ser mostrado en el campo usuario */  
    Cookie[] todosLosCookies = request.getCookies();  
    for (int i=0; i<todosLosCookies.length; i++) {  
        Cookie unCookie = todosLosCookies[i];  
        if (unCookie.getName().equals("cokieUsu")) {  
            usuario = unCookie.getValue();  
        }  
    }  
    /* Para mostrar la fecha del último acceso a la página.  
    Para ver si el cookie que almacena la fecha existe, se busca en los  
    cookies existentes. */  
    for (int i=0; i<todosLosCookies.length; i++) {  
        Cookie unCookie = todosLosCookies[i];  
        if (unCookie.getName().equals("ultimoAcceso")) {  
            fechaUltimoAcceso = unCookie.getValue();  
        }  
    }  
    /* Se comprueba que la variable es igual a vacío, es decir  
    no hay ningún cookie llamado "ultimoAcceso", por lo que se  
    recupera la fecha, y se guarda en un nuevo cookie . */  
    if (fechaUltimoAcceso.equals("")) {  
        Date fechaActual = new Date();  
        fechaUltimoAcceso = fechaActual.toString();
```



```
Cookie cookieFecha = new
Cookie("ultimoAcceso",fechaUltimoAcceso);
cookieFecha.setPath("/");
cookieFecha.setMaxAge(60*60*24);
response.addCookie(cookieFecha);
}

%>
<html>
<head>
<title>Proceso de login</title>
</head>
<body>
<b>PROCESO DE IDENTIFICACIÓN</b>
<br>Última vez que accedió a esta
página:<br><%=fechaUltimoAcceso%>
<%
if (request.getParameter("error") != null) {
    out.println(request.getParameter("error"));
}
%>
<form action="checklogin.jsp" method="post">
    usuario:
    <input type="text" name="usuario" size="20" value="<%=usuario%>"><br>
    clave:
    <input type="password" name="clave" size="20"><br>
    Recordar mi usuario:
    <input type="checkbox" name="recordarUsuario" value="on"><br>
    <input type="submit" value="enviar">
</form>
</body>
</html>
```

Teniendo un aspecto final similar a la siguiente Figura 42.

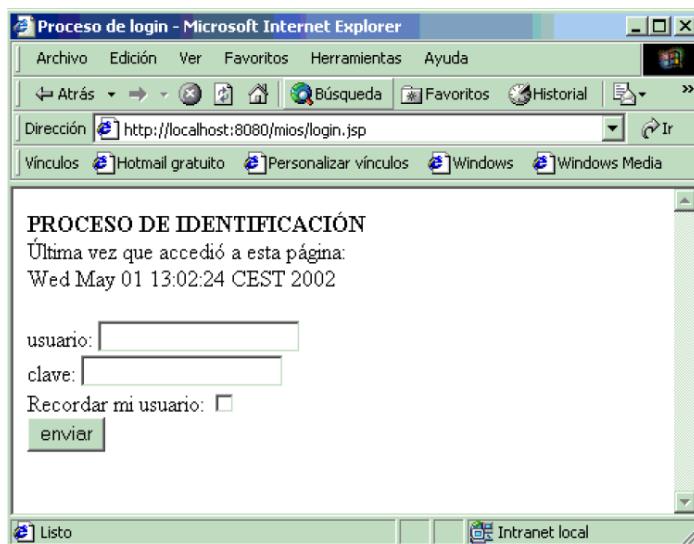


Figura 42: Página login utilizando cookie

El siguiente paso es modificar la página checklogin.jsp que recoge el usuario y clave introducidos y por lo tanto ahora también la nueva opción de "**Recordar mi usuario**". Dentro de la condición que se cumple si el usuario y la clave son correctos, y después de crear la sesión, escribimos el código que creará el cookie con el usuario. El primer paso es comprobar que el usuario ha activado esta opción, es decir, ha seleccionado el checkbox. También se realiza la comprobación de que el campo "recordarUsuario" no llegue con el valor nulo y produzca un error en la aplicación, en caso de que el usuario deje sin seleccionar el checkbox:

```
<%@ page session="true" import="java.util.*"%>
<%
String usuario = "";
String clave = "";
if (request.getParameter("usuario") != null)
    usuario = request.getParameter("usuario");
if (request.getParameter("clave") != null)
    clave = request.getParameter("clave");
if (usuario.equals("spiderman") && clave.equals("librojsp")) {
    out.println("checkbox: " +
        request.getParameter("recordarUsuario") + "<br>");
    HttpSession sesionOk = request.getSession();
    sesionOk.setAttribute("usuario",usuario);
    if ((request.getParameter("recordarUsuario") != null) &&
        (request.getParameter("recordarUsuario").equals("on")))
    {
        out.println("entra");
    }
}
```



```
Cookie cookieUsuario = new Cookie("cokieUsu",usuario);
cookieUsuario.setPath("/");
cookieUsuario.setMaxAge(60*60*24);
response.addCookie(cookieUsuario);
}
/* Se realiza un proceso similar a la creación de cookie de
recordar el usuario. En este caso se trata de crear un nuevo cookie
con el nuevo valor de la fecha y guardarla con el mismo nombre. De
esta forma será borrado el anterior y prevalecerá el valor del último.
*/
Date fechaActual = new Date();
String fechaUltimoAcceso = fechaActual.toString();
Cookie cookieFecha = new
Cookie("ultimoAcceso",fechaUltimoAcceso);
cookieFecha.setPath("/");
cookieFecha.setMaxAge(60*60*24);
response.addCookie(cookieFecha);
%>
<jsp:forward page="menu.jsp" />
<% } else { %>
<jsp:forward page="login.jsp">
<jsp:param name="error" value="Usuario y/o clave
incorrectos.<br>Vuelve a intentarlo."/>
</jsp:forward>
<% } %>
```

Señalar que los cookies están envueltos en una mala fama relacionada con la intromisión de la privacidad de los usuarios por parte de determinados websites e incluso con la seguridad. Se trata de pequeños ficheros de texto que poco daño pueden causar, y que como es lógico no pueden contener ningún código ejecutable.

Es cierto que determinados sites pueden saber que clase de palabras se consultan en un buscador o cual es nuestra frecuencia de visita a un determinado web. En cualquier caso son datos que no descubren la verdadera identidad del usuario, aunque si bien por este motivo, muchos de ellos deciden desactivar la recepción de cookies. Si esto es así las páginas en las que se utilicen cookies no funcionarán de forma correcta al no permitir ser recibidas por los navegadores.

Las sesiones en el protocolo HTTP

HTTP es un protocolo sin estado (stateless), cada petición al servidor web y su correspondiente respuesta es manejada como una transacción aislada. El servidor



(HTTP) no tiene manera de determinar que una serie de peticiones provienen del mismo cliente.

Las sesiones, en una aplicación web, se utilizan para asociar las peticiones con el cliente. Una sesión comienza cuando un cliente desconocido envía la primera petición (que requiera crear una sesión) a la aplicación web. La misma finaliza, cuando el cliente la termina explícitamente o cuando se expira el tiempo de vida de la sesión (debido a que no se ha recibido una petición en un tiempo determinado).

La manera en que una aplicación web mantiene una sesión con el cliente es:

- Cuando el servidor web recibe la primera petición de un cliente, el servidor inicia una sesión y le asigna un identificador único.
- El cliente debe incluir este identificador único en cada petición subsecuente. El servidor revisa el identificador y asocia la petición con su correspondiente sesión.

Existen dos formas de implementar el soporte para sesiones, cookies y la sobreescritura de la URL.

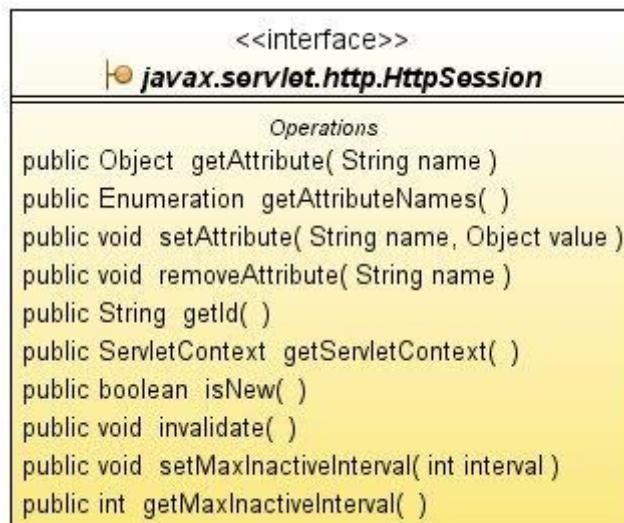


Figura 43: Interfaz HttpSession

javax.servlet.http.HttpSession

La API de servlets abstrae el concepto de sesión a través de esta interface. Cuando se inicia una sesión para un cliente, el contenedor de servlets instancia un objeto HttpSession. Los distintos servlets pueden utilizar este objeto para almacenar información relacionada al usuario y de esta manera mantener el estado del mismo. Existe un objeto HttpSession por cada sesión (o usuario). El contenedor de servlets se encarga de asociar una petición con su correspondiente objeto de sesión.



La interface `javax.servlet.http.HttpServletRequest` provee dos métodos para obtener la sesión:

- `HttpSession getSession (boolean create)`: Devuelve el correspondiente objeto sesión, o si no existe ninguna sesión y el parámetro sesión es true, crea una nueva sesión.
- `HttpSession getSession ()`: Método equivalente a invocar `getSession (true)`.

Métodos de la interface `HttpSession` para el manejo de sus atributos:

- `void setAttribute (String name, Object value)`: Asocia el objeto pasado como parámetro a la sesión, y lo identifica a través del nombre.
- `Object getAttribute (String name)`: Devuelve el objeto asociado al nombre, o null en caso de no existir ninguno.

Cerrar sesiones

La interface `HttpSession` provee el método:

- `void invalidate()`: Cierra una sesión y desenlaza todos los objetos asociados a ella. Lanza la excepción `IllegalStateException` si la sesión ya ha sido cerrada.

Session timeout

El protocolo HTTP no provee ninguna señal de la terminación de una sesión en el servidor. Por este motivo, cuando el usuario no realiza ninguna acción en un período de tiempo determinado, el servidor cierra dicha sesión.

En el archivo `web.xml`, el elemento `<session-timeout>` contiene en minutos el tiempo de vida de una sesión. Un valor 0 o menor significa que la sesión nunca expirará.

```
<web-app ...="">
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
</web-app>
```

La interface `HttpSession` también posee los siguientes métodos:

- `void setMaxInactiveInterval (int seconds)`: Afecta solo a la sesión en la cual se invoca. Las demás, mantienen el tiempo especificado en el archivo `web.xml`.
- `void getMaxInactiveInterval ()`



Existen dos inconsistencias entre estos modos de setear el timeout:

- El valor de elemento `<session-timeout>` se especifica en minutos, mientras que `setMaxInactiveInterval()` en segundos.
- El valor 0 o menos en el elemento `<session-timeout>` significa que la sesión nunca expirará, mientras que al método `setMaxInactiveInterval()` se le debe pasar un valor negativo (no 0) para conseguir el mismo comportamiento.

El HttpSession trabaja de dos formas usando Cookies y Url Writting esta última alternativa permite añadir, dentro de una respuesta, el identificador de sesión en todas las URL's. De esta manera, cuando el usuario interactúe con el sitio, enviará automáticamente su identificador de sesión como parte de la petición.

Este mecanismo no es transparente al desarrollador (a diferencia de las cookies). La interface `HttpServletResponse` provee los métodos:

- `String encodeURL (String url)`: Devuelve la url con el identificador de sesión adjuntado.
- `String encodeRedirectURL (String url)`: Idem anterior, pero se utiliza junto con el método `sendRedirect()`.

Ambos métodos verifican si es necesario adjuntar el identificador. Si la petición contiene un encabezado de cookie, entonces las mismas están habilitadas en el browser cliente, y la sobreescritura de la url no es necesaria.

Se debe tener en cuenta que en la URL, el ID de la sesión se agrega luego del carácter ';'. Por ejemplo:

```
http://localhost:8080/scwcd/LoginServlet;jsessionid=1C9B3BAE94BE88AD45AA54B2A8AC0246?  
user=matias
```

Esto se debe a que `jsessionid` es parte del path info de la petición, y no es un parámetro de la misma. Este valor NO puede ser accedido a través del método `HttpServletRequest.getParameter("jsessionid")`.

MANEJANDO LOS EVENTOS DE SESIÓN

En una aplicación web, cuando ocurren eventos importantes, se pueden recibir las notificaciones correspondientes a los mismos a través de las interfaces listener. Se crea una clase que implemente la interface listener correspondiente, y el contenedor de servlets invocará los métodos apropiados sobre un objeto (de esta clase) cuando ocurra algún evento.



La API de servlets define cuatro interfaces listener y dos eventos relacionados a las sesiones dentro de javax.servlet.http:

HttpSessionAttributeListener

Permite recibir notificaciones cuando se agrega, reemplaza, o elimina un atributo de un objeto HttpSession. La clase se debe especificar en el archivo web.xml. El contenedor de servlets solo crea una instancia, sobre la cual se invocaran los métodos para todas las sesiones.

- **void attributeAdded (HttpSessionBindingEvent sbe):** Invocado cuando se agrega un atributo a una sesión.
- **void attributeRemoved (HttpSessionBindingEvent sbe):** Invocado cuando se elimina un atributo.
- **void attributeReplaced (HttpSessionBindingEvent sbe):** Invocado cuando se reemplaza un atributo.

HttpSessionBindingListener

Esta interface es implementada por aquellas clases cuyos objetos necesiten recibir notificaciones de cuando ellos son agregados y eliminados de una sesión. Esta clase no se especifica en el archivo web.xml. Cuando se agrega o elimina un objeto de una sesión, el contenedor revisa las interfaces que implementa. En caso de que el objeto implemente la interface HttpSessionBindingListener, el contenedor invoca el método correspondiente:

- **void valueBound (HttpSessionBindingEvent event):** Notifica al objeto que está siendo asociado a una sesión.
- **void valueUnbound (HttpSessionBindingEvent event):** Notifica al objeto que está siendo desasociado de una sesión.

HttpSessionListener

Esta interface se utiliza para recibir notificaciones de cuando una sesión se crea o se destruye. La clase que implemente dicha interface debe ser especificada en el archivo web.xml. Se crea una nueva instancia de esta clase, por cada nueva sesión creada. Posee los métodos:

- **void sessionCreated (HttpSessionEvent se):** Se invoca cuando se crea una sesión.



- **void sessionDestroyed (HttpSessionEvent se):** Se invoca cuando se destruye una sesión. Si dentro de este método se invoca el método getAttribute(), se lanza la excepción IllegalStateException.

HttpSessionActivationListener

Esta interface se utiliza por los atributos de una sesión, para recibir notificaciones de cuando la sesión está siendo migrada a través de JVM's en un ambiente distribuido. De esta manera, los atributos contenidos en una sesión no se pierden, más allá de que exista más de una JVM ejecutando la aplicación. Posee los métodos:

- **void sessionDidActivate (HttpSessionEvent se):** Se invoca justo después de que la sesión se activa.
- **void sessionWillPassivate (HttpSessionEvent se):** Se invoca antes de desactivar la sesión.

Capítulo 7

JavaServer Pages

Standard Tag Library

¿QUÉ SON TAG LIBRARIES?

La tecnología JavaServer Pages Standard Tag Library (JSTL) es un componente de Java EE. Extiende las ya conocidas JavaServer Pages (JSP) proporcionando cinco librerías de etiquetas (Tag Libraries) con utilidades ampliamente utilizadas en el desarrollo de páginas web dinámicas.

Estas librerías de etiquetas extienden de la especificación de JSP (la cual a su vez extiende de la especificación de Servlet). Su API nos permite además desarrollar nuestras propias librerías de etiquetas.

CARACTERÍSTICAS

- Las páginas JSTL son también páginas JSP. JSTL es un superconjunto de JSP.
- JSTL provee un conjunto de cinco librerías estándar: Core (Internationalization/format), XML, SQL y Funciones.
- Además JSTL define un nuevo lenguaje de expresiones llamado EL, que ha sido luego adoptado por JSP 2.0
- Una etiqueta JSTL corresponde a una acción; llamándolas acción nos indica que añaden comportamiento dinámico a una, de otra manera, página estática.



¿QUÉ ES “EL”?

El lenguaje de expresiones “EL” simplemente define un poderoso mecanismo para expresar expresiones simples en una sintaxis muy sencilla.

- Es algo entre JavaScript y Perl.
- Su combinación con las etiquetas de las 4 librerías antes mencionadas proveen mucha flexibilidad y poder para el desarrollo de páginas dinámicas.
- En “EL” las expresiones están delimitadas por \${ }.

Ejemplos:

```
 ${anExpression}
```

```
 ${aList[4]}
```

```
 ${aList[someVariable]} -> acceso a un elemento de una colección
```

```
 ${anObject.aProperty} -> acceso a la propiedad de un objeto
```

```
 ${anObject["aPropertyName"]} -> entrada en un mapa con propiedad a propertyName
```

```
 ${anObject[aVariableContainingPropertyName]}
```

Variables implícitas definidas en EL:

- **pageContext**: el contexto del JSP actual
- **pageScope, requestScope, sessionScope, y applicationScope**: colecciones de mapas que mapean nombres de variables en esos contextos a valores.
- **param y paramValues**: parámetros pasados con la petición de la página, lo mismo que en JSP.
- **header y headerValues**: cabeceras pasadas en la petición de la página.
- **cookie**: mapa que mapea nombres de cookies a los valores de las mismas.



JSTL TAG LIBRARIES

Librería	URI	Prefijo Librería
Core	http://java.sun.com/jsp/jstl/core	c
Internationalization I18N formateo	http://java.sun.com/jsp/jstl/fmt	fmt
SQL/DB support	http://java.sun.com/jsp/jstl/sql	sql
Procesamiento XML	http://java.sun.com/jsp/jstl/xml	x
Functions	http://java.sun.com/jsp/jstl/functions	fn

Declarar Tag Libraries

Las directivas han de incluirse al comienzo de la página.

Ejemplo:

```
<%@ taglib prefix="c" uri=http://java.sun.com/jsp/jstl/core %>
```

Para utilizar una etiqueta de una librería simplemente se ha de preceder con el prefijo de la librería utilizada.

Ejemplo:

```
<c:out value="${anExpression}"/>
```

O simplemente su uso a partir de JSTL 1.2 es

```
${anExpression}
```

¿QUÉ ES JSTL?

JSTL es una biblioteca que implementa funciones de uso frecuente en aplicaciones JSP. En concreto, JSTL proporciona:

- Cinco bibliotecas de etiquetas JSP
- Funciones comunes de iteración sobre datos, operaciones condicionales, e importación de otras páginas.
- Internacionalización y formateo de texto.
- Funciones de manipulación de cadenas.
- Procesamiento de XML.



- Acceso a bases de datos.
- Un lenguaje de expresión para referenciar objetos y sus propiedades sin necesidad de código Java.
- Validadores de bibliotecas de etiquetas (Tag Library Validators, TLVs).
- Por su simplicidad, JSTL solo requiere conocimientos rudimentarios de Java, JSP, y aplicaciones Web. Cualquier desarrollador puede comenzar a usarlo de forma casi inmediata.
- JSTL facilita la referencia a objetos, ejemplo:

```
<%-- con JSP --%>
<%= session.getAttribute("username").getFirstName()%>
<%-- con JSTL --%>
${sessionScope.username.firstName}
```

- JSTL requiere un contenedor de JSP 2.0. o superior.

USO DE LAS LIBRERÍAS ESTÁNDARES DE JSTL

Como vimos anteriormente, JSTL provee un conjunto de cinco librerías: Core, Internationalization/format, XML, SQL y Funciones

Core:

Permiten llevar a cabo las siguientes acciones:

- Visualizar/asignar valores y manejar excepciones
- Control de flujo
- Otras acciones de utilidad

Para visualizar valores utilizamos:

```
<c:out:value="${applicationScope.product.inventoryCount}" escapeXml="true" default="0" />
of those items in stock
```

Nota: escapeXml indica si hay que aplicar códigos de escape a los caracteres <, > y &.

Asignar una variable en una página:

```
<c:set var="customerID" value="$param.customerNumber" scope="session" />
```



Nota: scope indica el contexto en el que se define la variable

También podemos asignar el contenido de una etiqueta a una variable:

```
<c:set var="cellContents">
  <td>
    <c:out value="${myCell}" />
  </td>
</c:set>
```

Normalmente en un JSP o incluimos un bloque try/catch o usamos la directiva errorPage:

```
<c:catch>
  <!-- . . . -->
</c:catch>
```

Para borrar una variable se puede utilizar <c:remove>.

Para llevar a cabo simples condiciones (c:if):

```
<c:if test="${status.totalVisits == 1000000}" var="visits">
  You are the millionth visitor to our site! Congratulations!
</c:if>
```

El switch de un lenguaje de programación se puede emular con c:choose:

```
<c:choose>
  <c:when test="${item.type == 'book'}">
    ...
  </c:when>
  <c:when test="${item.type == 'electronics'}">
    ...
  </c:when>
  <c:when test="${item.type == 'toy'}">
    ...
  </c:when>
  <c:otherwise>
    ...
  </c:otherwise>
</c:choose>
```

Para iterar sobre una colección se define **c:foreach**. Se pueden especificar índice de comienzo, final e incremento con los atributos begin, end y step.



```
<table>
    <c:forEach var="name" items="${customerNames}">
        <tr><td><c:out value="${name}" /></td></tr>
    </c:forEach>
</table>
```

Funcionalidad similar a StringTokenizer puede ser obtenida en JSTL con c:forTokens:

```
<table>
    <c:forTokens items="47,52,53,55,46,22,16,2" delims="," var="dailyPrice">
        <tr><td><c:out value="${dailyPrice}" /></td></tr>
    </c:forTokens>
</table>
```

Ejemplo práctico:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Parameter Listing Example</title>
    </head>
    <body>
        <br>
        <b>Parameter values passed to this page for each parameter: </b>
        <table border="2">
            <c:forEach var="current" items="${param}">
                <tr>
                    <td>
                        <b><c:out value="${current.key}" /></b>
                    </td>
                    <c:forEach var="aVal" items="${paramValues[current.key]}">
                        <td>
                            <c:out value="${aVal}" />
                        </td>
                    </c:forEach>
                </tr>
            </c:forEach>
        </table>
    </body>
</html>
```

Para codificar URLs se puede utilizar c:url:

```
<c:url value="http://acme.com/exec/register" var="myUrl">
```



```
<c:param name="name" value="${param.name}"/>
<c:param name="country" value="${param.country}"/>
</c:url>
<a href='<c:out value="${myUrl}"/>'>Register</a>
```

Se pueden importar otros JSPs o incluso otros recursos en una URL arbitraria usando c:import (análogo to jsp:include)

Para manejar redireccionamiento se puede utilizar la etiqueta c:redirect

Cuando EL se migró de la especificación JSTL a JSP, se le añadió una nueva funcionalidad: la invocación a funciones. Su uso es trivial: nombre-func(lista-params)

```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
${fn:length(myCollection)}
```

Internacionalizacion / Formateo

Cubre dos áreas:

- Etiquetas (acciones) de formateo
- Acciones de internacionalización

Acciones de formateo:

- Inspiradas en el funcionamiento de las clases DateFormat y NumberFormat
- Para formatear un número usamos formatNumber con los atributos number para el número y pattern para el patrón de formateo a aplicar.

```
<fmt:formatNumber value="1000.001" pattern="#,#00.0#"/>
```

- Si queremos parsear un número a partir de un string usamos parseNumber:

```
<fmt:parseNumber value="${currencyInput}" type="currency" var="parsedNumber"/>
```

- Para formatear una fecha usamos formatDate y para parsear un string parseDate:

```
<jsp:useBean id="now" class="java.util.Date" />
<fmt:formatDate value="${now}" timeStyle="long" dateStyle="long"/>
<fmt:parseDate value="${dateInput}" pattern="MM dd, YYYY" />
```



Acciones de internacionalización:

Una pieza importante de la localización en Java es la clase ResourceBundle. Las acciones JSTL que permiten trabajar con esta clase son:

- fmt:bundle para obtener un ResourceBundle correspondiente al Locale actual y
- fmt:message para hacer lookups en el ResourceBundle.

Ejemplo:

```
<fmt:bundle basename="myBundle">
    <%-- Use values in myBundle --%>
    <fmt:message key="Introduction">
        <fmt:param value="${loginName}" />
        <fmt:param value="${loginCount}" />
    </fmt:message>
    <fmt:formatDate value="${now}" var="parsedDate"/>
</fmt:bundle>
```

PROCESAMIENTO XML

- El soporte de XML que lleva a cabo JSTL conforma con la especificación XPath.
- Xpath provee una sintaxis clara para acceder a partes jerárquicas de un documento.
- Accion c:import es utilizada para importar un documento, mientras x:parse para generar un árbol DOM a partir de él. x:set crea una variable a partir de un extracto de XML

```
<c:import url="http://www.cheapstuff.com/orderStatus?id=2435" var="xml"/>
<x:parse xml="${xml}" var="doc"/>
<!-- access XML data via XPath expressions -->
<x:out select="$doc/name"/>
<x:out select="$doc/shippingAddress"/>
<x:out select="$doc/deliveryDate"/>
<!-- Set a scoped variable -->
<x:set var="custName" scope="request" select="$doc/name"/>
```



SQL/DB

JSTL permite una fácil integración con bases de datos.

Seleccionar y visualizar un conjunto de elementos.

Ejemplo:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
<sql:query var="orderItems" dataSource="jdbc/sistemaweb">
    SELECT * FROM items
    WHERE order_id = <cout value="${orderID}" />
    ORDER BY price
</sql:query>
<table>
    <c:forEach var="row" items="${orderItems.rows}">
        <tr>
            <td><c:out value="${row.itemName}" /></td>
            <td><c:out value="${row.price}" /></td>
            <td><c:out value="${row.weight}" /></td>
        </tr>
    </c:forEach>
</table>
```

También se soportan acciones para manejar transacciones (sql:transaction), sql:update soporta no sólo updates sino que también insert y delete e incluso create, es decir todas las acciones SQL que no devuelven un resultado: ejemplo.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<sql:query var="rs" dataSource="jdbc/sistemaweb">
    select id, foo, bar from testdata
</sql:query>
<html>
    <head>
        <title>DB Test</title>
    </head>
    <body>
        <h2>Results</h2>
        <c:forEach var="row" items="${rs.rows}">
            Foo ${row.foo}<br/>
            Bar ${row.bar}<br/>
        </c:forEach>
    </body>
</html>
```



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

DESARROLLAR ETIQUETAS JSP PERSONALIZADAS

Introducción

La tecnología JavaServer Pages (JSP) hace fácil embeber trozos de código Java (o scriptlets) en documento HTML. Sin embargo, esta solución, podría no ser adecuada para todos los desarrolladores de contenido HTML, quizás porque no conocen Java o no les interesa aprender su sintaxis. Aunque los JavaBeans pueden usarse para encapsular mucho código Java, usándolos en páginas JSP todavía requieren que los desarrolladores de contenido tengan algún conocimiento sobre su sintaxis.

La tecnología JSP nos permite introducir nuevas etiquetas personalizadas a través de una librería de etiquetas. Como desarrollador Java, podemos ampliar las páginas JSP introduciendo etiquetas personalizadas que pueden ser desplegadas y usadas en una sintaxis al estilo HTML. Las etiquetas personalizadas también nos permiten proporcionar un mejor empaquetamiento, mejorando la separación entre la lógica del negocio y su representación.

Básicamente hay dos tipos de etiquetas, y ambos pueden tener atributos (información sobre cómo la etiqueta debería hacer su trabajo):

- **Etiquetas sin cuerpo:** Una etiqueta sin cuerpo es una etiqueta que tiene una etiqueta de apertura pero no tiene su correspondiente etiqueta de cierre. Tiene la sintaxis:

```
<tagName attributeName="value" anotherAttributeName="anotherValue"/>
```

Las etiquetas sin cuerpo se usan para representar ciertas funciones, como la presentación de un campo de entrada o para mostrar una imagen. Aquí tenemos un ejemplo de una etiqueta sin cuerpo en HTML:

```
<IMG SRC=".//fig10.gif">
```

- **Etiquetas con cuerpo:** Una etiqueta con cuerpo tiene una etiqueta de inicio y una etiqueta de fin. Tiene la sintaxis:

```
<tagName attributeName="value" anotherAttributeName="anotherValue">  
  
tag body...  
  
</tagName>
```



Las etiquetas con cuerpo se usan para realizar operaciones sobre el contenido del cuerpo, como formatearlo. Aquí tenemos un ejemplo de una etiqueta con cuerpo en HTML:

```
<H2>Custom Tags</H2>
```

Etiquetas JSP Personalizadas

Las etiquetas JSP personalizadas son simplemente clases Java que implementan unos interfaces especiales. Una vez que se han desarrollado y desplegado, sus acciones pueden ser llamadas desde nuestro HTML usando sintaxis XML. Tienen una etiqueta de apertura y otra de cierre. Y podrían o no podrían tener un cuerpo. Las etiquetas sin cuerpo pueden expresarse como:

```
<tagLibrary:tagName />
```

Y una etiqueta con cuerpo, podría expresarse de esta forma:

```
<tagLibrary:tagName>  
  
    body  
  
</tagLibrary:tagName>
```

De nuevo, ambos tipos podrían tener atributos que sirven para personalizar el comportamiento de la etiqueta. La siguiente etiqueta tiene un atributo llamado name, que acepta un valor String obtenido validando la variable yourName:

```
<mylib:hello name="<% yourName %>" />
```

O podría escribirse como una etiqueta con cuerpo:

```
<mylib:hello>  
    <%= yourName %>  
</mylib:hello>
```

Nota:

Cualquier dato que sea un simple String, o pueda ser generado evaluando una expresión simple, debería ser pasado como un atributo y no como contenido del cuerpo.

Beneficios de las Etiquetas Personalizadas

Algo muy importante a observar sobre las etiquetas JSP personalizadas es que no ofrecen más funcionalidad que los scriptlets, simplemente proporcionan un mejor empaquetamiento, ayudandonos a mejorar la separación entre la lógica del negocio y su representación. Algunos de sus beneficios son:



- Pueden reducir o eliminar los scriptlets en nuestras aplicaciones JSP. Cualquier parámetro necesario para la etiqueta puede pasarse como atributo o contenido del cuerpo, por lo tanto, no se necesita código Java para inicializar o seleccionar propiedades de componentes.
- Tienen una sintaxis muy simple. Los scriptlets están escritos en Java, pero las etiquetas personalizadas pueden usar una sintaxis al estilo HTML.
- Pueden mejorar la productividad de los desarrolladores de contenido que no son programadores, permitiéndoles realizar tareas que no pueden hacer con HTML.
- Son reutilizables. Ahorran tiempo de desarrollo y de prueba. Los Scriptlets no son reusables, a menos que llamemos reutilización a "copiar-y-pegar".

Definir una etiqueta

Una etiqueta es una clase Java que implementa un interface especializado. Se usa para encapsular la funcionalidad desde una página JSP. Como mencionamos anteriormente, una etiqueta puede o no tener cuerpo. Para definir una sencilla etiqueta sin cuerpo, nuestra clase debe implementar el interface *Tag*. El desarrollo de etiquetas con cuerpo se discute más adelante.

A continuación se muestra el código fuente de la interface *Tag* que debemos 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. La Tabla 1 proporciona una descripción de los métodos del interface Tag:

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.

Primera etiqueta

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. En este ejemplo, la etiqueta implementa el interface Tag como se muestra en el siguiente ejemplo:

HelloTag.java

```
package tags;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HelloTag implements Tag {

    private PageContext pageContext;
    private Tag parent;

    public HelloTag() {
        super();
    }

    public int doStartTag() throws JspException {
        try {
            pageContext.getOut().print("This is my first tag!");
        } catch (IOException ioe) {
            throw new JspException("Error: IOException while writing to client" +
                ioe.getMessage());
        }
        return SKIP_BODY;
    }

    public int doEndTag() throws JspException {
        return SKIP_PAGE;
    }

    public void release() {
    }

    public void setPageContext(PageContext pageContext) {
        this.pageContext = pageContext;
    }

    public void setParent(Tag parent) {
```



```
    this.parent = parent;  
}  
  
public Tag getParent() {  
    return parent;  
}  
  
}
```

Los dos métodos importantes a observar en HelloTag son doStartTag y doEndTag. El primero es invocado cuando se encuentra la etiqueta de inicio. En este ejemplo, este método devuelve SKIP_BODY porque es una simple etiqueta sin cuerpo. El método doEndTag es invocado cuando se encuentra la etiqueta de cierre. En este ejemplo devuelve SKIP_PAGE porque no queremos evaluar el resto de la página, de otro modo debería devolver EVAL_PAGE.

Crear el descriptor de librería de etiquetas

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. El siguiente ejemplo muestra un TLD de ejemplo:

mytaglib.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">  
  
<!-- a tag library descriptor -->  
<taglib>  
  
    <tlibversion>1.0</tlibversion>  
    <jspversion>1.1</jspversion>  
    <shortname>first</shortname>  
    <uri></uri>  
    <info>A simple tag library for the examples</info>  
  
    <tag>  
        <name>hello</name>  
        <tagclass>tags.HelloTag</tagclass>
```



```
<bodycontent>empty</bodycontent>
<info>Say Hi</info>
</tag>

</taglib>
```

Primero especificamos la versión de la librería de etiquetas y la versión de JSP. La etiqueta `<shortname>` indica como se va a referencia la librería de etiquetas desde la página JSP. La etiqueta `<uri>` puede usarse como un identificador único para nuestra librería de etiquetas.

En este TLD, sólo tenemos una etiqueta llamada `hello` cuya clase se especifica usando la etiqueta `<tagclass>`. Sin embargo, una librería de etiquetas puede tener tantas etiquetas como queramos. El `<bodycontent>` nos dice que esta etiqueta no tiene cuerpo; de otro modo se produciría un error. Por otro lado, si queremos evaluar el cuerpo de la etiqueta, el valor debería ser:

- **tagdependent:** lo que significa que cualquier cuerpo de la etiqueta sería manejado por la propia etiqueta, y puede estar vacío.
- **JSP:** lo que significa que el contenedor JSP evaluaría cualquier cuerpo de la etiqueta, pero también podría estar vacío.

Grabamos **mytaglib.tld** en el directorio **WEB-INF**

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:

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

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

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

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

6. 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 **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/mytaglib.tld</taglib-location>
</taglib>
```

Ahora, escribimos un JSP y usamos la primera sintaxis. Lo podremos ver en el siguiente ejemplo:

Hello.jsp

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

<HTML>
    <HEAD>
        <TITLE>Hello Tag</TITLE>
    </HEAD>
    <BODY bgcolor="#ffffcc">
        <B>My first tag prints</B>:
        <first:hello/>
    </BODY>
</HTML>
```

El **taglib** se usa para decirle al motor de ejecución JSP donde encontrar el descriptor para nuestra etiqueta, y el **prefix** especifica cómo se referirá a las etiquetas de esta librería. Con esto correcto, el motor de ejecución JSP reconocerá cualquier uso de nuestra etiqueta a lo largo del JSP, mientras que la precedamos con el prefijo **first** como en **<first:hello/>**.

Alternativamente, podemos usar la segunda opción de referencia creando un fichero JAR. O podemos usar la tercera opción simplemente reemplazando la primera línea del ejemplo anterior con la siguiente línea:

```
<%@ taglib uri="mytags" prefix="first" %>
```

Básicamente, hemos usado el nombre mytags que se ha añadido a web.xml, para referenciar a la librería de etiquetas. Para el resto de los ejemplos de esta página utilizaremos este tipo de referencia.

Ahora, si solicitamos **Hello.jsp** desde nuestro navegador, veríamos algo similar a la Figura 44.

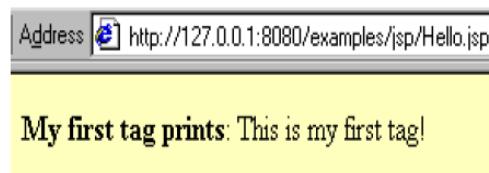


Figura 44: Ejecución de un tag personalizado



La etiqueta personalizada desarrollada en el ejemplo es una etiqueta sencilla, el objetivo era sólo ofrecernos uno poco del sabor del esfuerzo que implica el desarrollo de etiquetas personalizadas. Podríamos haber observado que incluso está simple etiqueta requiere que implementemos un gran número de métodos, algunos de los cuales tienen implementaciones muy simples. Para minimizar el esfuerzo implicado, los diseñadores de JSP proporcionaron una plantilla a utilizar en la implementación de etiquetas sencillas. La plantilla es la clase abstracta TagSupport. Es una clase de conveniencia que proporciona implementaciones por defecto para todos los métodos del interface Tag.

Por lo tanto, la forma más fácil de escribir etiquetas sencillas es extender la clase TagSupport en vez de implementar el interface Tag. Podemos pensar en la clase abstracta TagSupport como en un adaptador. Habiendo dicho esto, la clase HelloTag del ejemplo 4 podría implementarse más fácilmente como se ve en el siguiente ejemplo.

```
package tags;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HelloTag extends TagSupport {

    public int doStartTag() throws JspException {
        try {
            pageContext.getOut().print("This is my first tag!");
        } catch (IOException ioe) {
            throw new JspException("Error: IOException while writing to client" +
                ioe.getMessage());
        }
        return SKIP_BODY;
    }

    public int doEndTag() throws JspException {
        return SKIP_PAGE;
    }
}
```



Etiquetas parametrizadas

Hemos visto como desarrollar etiquetas sencillas. Ahora veamos cómo desarrollar etiquetas parametrizadas--etiquetas que tienen atributos. Hay dos nuevas cosas que necesitamos añadir al ejemplo anterior para manejar atributos:

1. Añadir un método set
2. Añadir una nueva etiqueta a mytagslib.tld

Añadir un método set y cambiar el mensaje de salida resulta en el siguiente ejemplo:

```
package tags;

import java.io.*;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HelloTagParam extends TagSupport {

    private String name;
    public void setName(String name) {
        this.name = name;
    }

    public int doStartTag() throws JspException {
        try {
            pageContext.getOut().print("Welcome to JSP Tag Programming, " +name);
        } catch (IOException ioe) {
            throw new JspException("Error: IOException while writing to client");
        }
        return SKIP_BODY;
    }

    public int doEndTag() throws JspException {
        return SKIP_PAGE;
    }
}
```

Lo siguiente que necesitamos hacer es añadir una nueva etiqueta a mytaglib.tld. La nueva etiqueta se muestra en el siguiente ejemplo. Este fragmento de código debería añadirse a mytaglib.tld justo antes de la línea, </taglib>:



```
<tag>

    <name>helloparam</name>
    <tagclass>tags.HelloTagParam</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Tag with Parameter</info>
    <attribute>
        <name>name</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>

</tag>
```

Hemos añadido una nueva etiqueta llamada helloparam. Observa la nueva etiqueta `<attribute>`, que especifica que la etiqueta helloparam acepta un atributo cuyo nombre es name. La etiqueta `<required>` se selecciona a false, significando que el atributo es opcional; la etiqueta `<rtexprvalue>` se selecciona a false especificando que no se hará evaluación en el momento de la ejecución.

No necesitamos añadir nada al fichero descriptor de aplicación web web.xml porque estamos usando la misma librería de etiquetas: mytaglib.tld.

Ahora, podemos probar la nueva etiqueta. El código fuente del siguiente ejemplo muestra cómo probarla usando un atributo name de "JavaDuke".

```
<%@ taglib uri="mytags" prefix="first" %>

<HTML>
    <HEAD>
        <TITLE>Hello Tag with Parameter</TITLE>
    </HEAD>
    <BODY bgcolor="#ffffcc">
        <B>My parameterized tag prints</B>:
        <P><first:helloparam name="JavaDuke"/></P>
    </BODY>
</HTML>
```

Si solicitamos HelloTagParam.jsp desde un navegador web, veremos una salida similar a la de la Figura 45.

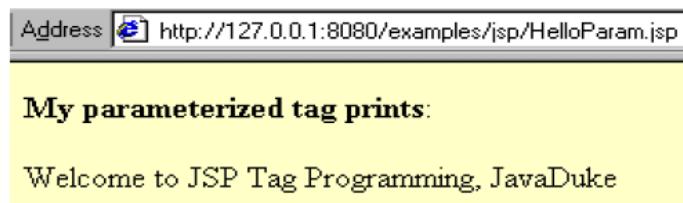


Figura 45: Ejecución de etiquetas con parámetros

Librerías de etiquetas

Una librería de etiquetas es una colección de etiquetas personalizadas JSP. El Jakarta Taglibs Project proporciona varias librerías de etiquetas útiles para analizar XML, transformaciones, email, bases de datos, y otros usos. Pueden descargarse y usarse muy fácilmente.

Aquí desarrollamos nuestra librería de etiquetas. Como un ejemplo, desarrollamos una sencilla librería matemática que proporciona dos etiquetas, una para sumar dos números y la otra para restar un número de otro. Cada etiqueta está representada por una clase. El código fuente de las dos clases, Add.java y Subtract.java, se muestra en el siguiente ejemplo.

Add.java

```
package tags;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class Add extends TagSupport {

    private int num1, num2;

    public void setNum1(int num1) {
        this.num1 = num1;
    }

    public void setNum2(int num2) {
        this.num2 = num2;
    }

    public int doStartTag() throws JspException {
        try {
            pageContext.getOut().print("Welcome to First Grade Math! ");
            pageContext.getOut().print("The sum of: " + num1 + " and " + num2 +
                " is " + (num1 + num2));
        } catch (Exception e) {
            e.printStackTrace();
        }
        return EVAL_BODY_INCLUDE;
    }
}
```



```
    " is: " + (num1+num2));
} catch (IOException ioe) {
    throw new JspException("Error: IOException while writing to client");
}
return SKIP_BODY;
}

}
```

Subtract.java

```
package tags;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class Subtract extends TagSupport {

    private int num1, num2;

    public void setNum1(int num1) {
        this.num1 = num1;
    }

    public void setNum2(int num2) {
        this.num2 = num2;
    }

    public int doStartTag() throws JspException {
        try {
            pageContext.getOut().print("Welcome to First Grade Math! ");
            pageContext.getOut().print("If you subtract: " + num2 + " from " + num1 +
                ", you get: "+ (num1 - num2));
        } catch (IOException ioe) {
            throw new JspException("Error: IOException while writing to client");
        }
        return SKIP_BODY;
    }

}
```



El código fuente es fácil de entender. Observa una cosa que hemos repetido en Add.java y Subtract.java es la llamada a pageContext.getOut.print. Una forma mejor de hacer esto sería obtener un objeto JspWriter y luego usarlo para imprimir hacia el cliente:

```
JspWriter out = pageContext.getOut();
out.print("first line");
out.print("second line");
```

El siguiente paso es revisar el fichero descriptor de librería de etiquetas, mytaglib.tld, y añadimos las descripciones para las dos nuevas etiquetas. El ejemplo 9 muestra la descripción de las nuevas etiquetas. Añadimos el siguiente fragmento de XML a mytaglib.tld, justo antes de la última línea.

```
<tag>

    <name>add</name>
    <tagclass>tags.Add</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Tag with Parameter</info>
    <attribute>
        <name>num1</name>
        <required>true</required>
        <rteprvalue>false</rteprvalue>
    </attribute>
    <attribute>
        <name>num2</name>
        <required>true</required>
        <rteprvalue>false</rteprvalue>
    </attribute>
</tag>

<tag>
    <name>sub</name>
    <tagclass>tags.Subtract</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Tag with Parameter</info>
    <attribute>
        <name>num1</name>
        <required>true</required>
        <rteprvalue>false</rteprvalue>
    </attribute>
    <attribute>
        <name>num2</name>
```



```
<required>true</required>
<rteprvalue>false</rteprvalue>
</attribute>
</tag>
```

Como podemos ver, cada etiqueta requiere dos atributos que deben llamarse num1 y num2.

Ahora podemos probar nuestra nueva librería de etiquetas matemáticas usando el probador mostrado en el siguiente ejemplo.

math.jsp

```
<%@ taglib uri="mytags" prefix="math" %>

<HTML>
  <HEAD>
    <TITLE>Hello Tag with Parameter</TITLE>
  </HEAD>
  <BODY bgcolor="#ffffcc">
    <B>Calling first tag</B>
    <P><math:add num1="1212" num2="121"/></P>
    <P><B>Calling second tag</B></P>
    <P><math:sub num1="2123" num2="3"/></P>
  </BODY>
</HTML>
```

Si solicitamos math.jsp desde un navegador web, veríamos una salida similar a la de la Figura 46:

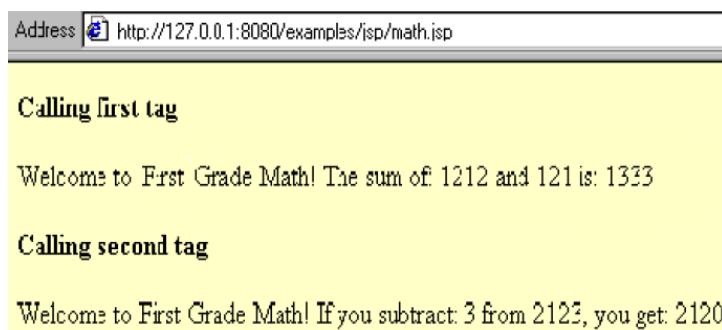


Figura 46: Ejecución de math.jsp

Etiquetas con cuerpo

Un manejador de etiquetas para una etiqueta con cuerpo se implementa de forma diferente dependiendo de si se necesita evaluar el cuerpo sólo una vez o varias veces.



- **Una Evaluación:** Si el cuerpo necesita evaluarse sólo una vez, el controlador de etiqueta debería implementar el interface Tag, o extender la clase abstracta TagSupport; el método doStartTag necesita devolver EVAL_BODY_INCLUDE, y si no necesita ser evaluado en absoluto debería devolver BODY_SKIP.
- **Multiple Evaluación:** Si el cuerpo necesita evaluarse varias veces, debería implementarse el interface BodyTag. Este interface extiende el interface Tag y define métodos adicionales (setBodyContent, doInitBody, y doAfterBody) que le permiten al controlador inspeccionar y posiblemente cambiar su cuerpo. De forma alternativa, similarmente a la clase TagSupport, podemos extender la clase BodyTagSupport, que proporciona implementaciones por defecto para los métodos del interface BodyTag. Típicamente, necesitaremos implementar los métodos doInitBody y doAfterBody. doInitBody es llamado después de que se haya seleccionado el contenido del cuerpo pero antes de que sea evaluado, y el doAfterBody es llamado después de que el contenido del cuerpo sea evaluado.

Una evaluación

Aquí tenemos un ejemplo de una sola evaluación donde hemos extendido la clase BodyTagSupport. Este ejemplo lee el contenido del cuerpo, lo convierte a minúsculas, y luego lo reescribe de vuelta hacia el cliente. El siguiente ejemplo muestra el código fuente. El contenido del cuerpo es recuperado como un String, convertido a minúsculas, y luego escrito de vuelta al cliente.

ToLowerCaseTag.java

```
package tags;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class ToLowerCaseTag extends BodyTagSupport {

    public int doAfterBody() throws JspException {
        try {
            BodyContent bc = getBodyContent();
            // get the bodycontent as string
            String body = bc.getString();
            // getJspWriter to output content
            JspWriter out = bc.getEnclosingWriter();
            out.print(body.toLowerCase());
        }
    }
}
```



```
if(body != null) {  
    out.print(body.toLowerCase());  
}  
} catch(IOException ioe) {  
    throw new JspException("Error: "+ioe.getMessage());  
}  
return SKIP_BODY;  
}  
  
}
```

El siguiente paso es añadir una etiqueta al fichero descriptor de la librería de etiquetas, mytaglib.tld. El nuevo descriptor de etiqueta es:

```
<tag>  
    <name>tolowercase</name>  
    <tagclass>tags.ToLowerCaseTag</tagclass>  
    <bodycontent>JSP</bodycontent>  
    <info>To lower case tag</info>  
</tag>
```

Observa que cuando escribimos una etiqueta con cuerpo, el valor de la etiqueta <bodycontent> debe ser JSP o jspcontent, como se explicó anteriormente. En el siguiente ejemplo, podemos ver un probador para este caso.

lowercase.jsp

```
<%@ taglib uri="mytags" prefix="first" %>  
  
<HTML>  
<HEAD>  
    <TITLE>Body Tag</TITLE>  
</HEAD>  
<BODY bgcolor="#ffffcc">  
    <first:tolowercase>  
        Welcome to JSP Custom Tags Programming.  
    </first:tolowercase>  
</BODY>  
</HTML>
```

Si solicitamos lowercase.jsp desde un navegador web, veríamos algo similar a la Figura 47.



Figura 47: Etiqueta personalizada con cuerpo

Multiples evaluaciones

Ahora veamos un ejemplo de un cuerpo de etiqueta evaluado múltiples veces. El ejemplo acepta un string e imprime el string tantas veces como se indique en el JSP. El código fuente se muestra en el siguiente ejemplo.

LoopTag.java

```
package tags;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class LoopTag extends BodyTagSupport {

    int times = 0;
    BodyContent bodyContent;

    public void setTimes(int times) {
        this.times = times;
    }

    public int doStartTag() throws JspException {
        if (times>0) {
            return EVAL_BODY_TAG;
        } else {
            return SKIP_BODY;
        }
    }

    public void setBodyContent(BodyContent bodyContent) {
        this.bodyContent = bodyContent;
    }

    public int doAfterBody() throws JspException {
        if (times >1) {
            times--;
        }
    }
}
```



```
        return EVAL_BODY_TAG;
    } else {
        return SKIP_BODY;
    }
}

public int doEndTag() throws JspException {
    try {
        if(bodyContent != null) {
            bodyContent.writeOut(
                bodyContent.getEnclosingWriter());
        }
    } catch(IOException e) {
        throw new JspException("Error: "+e.getMessage());
    }
    return EVAL_PAGE;
}

}
```

En este ejemplo, los métodos implementados juegan los siguientes papeles:

- El método doStartTag obtiene la llamada al inicio de la etiqueta. Chequea si se necesita realizar el bucle.
- El método setBodyContent es llamado por el contenedor JSP para chequear por más de un bucle.
- El método doAfterBody es llamado después de cada evaluación; el número de veces que se necesite realizar el bucle es decrementado en uno, luego devuelve SKIP_BODY cuando el número de veces no es mayor que uno.
- El método doEndTag es llamado al final de la etiqueta, y el contenido (si existe) se escribe en el writer encerrado.

Similarmente a los ejemplos anteriores, el siguiente paso es añadir un nuevo descriptor de etiqueta a mytaglib.tld. Las siguientes líneas muestran lo que necesitamos añadir:

```
<tag>
<name>loop</name>
<tagclass>tags.LoopTag</tagclass>
<bodycontent>JSP</bodycontent>
<info>Tag with body and parameter</info>
```



```
<attribute>
    <name>times</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

Observa que la etiqueta `<rtexprvalue>` especifica que las evaluaciones se ejecutarán en tiempo de ejecución.

En el siguiente ejemplo podemos ver un JSP probador:

loops.jsp

```
<%@ taglib uri="mytags" prefix="first" %>

<HTML>
<HEAD>
    <TITLE>Body Tag</TITLE>
</HEAD>
<BODY bgcolor="#ffffcc">
    <first:loop times="4">
        Welcome to Custom Tags Programming.<BR>
    </first:loop>
</BODY>
</HTML>
```

Finalmente, si solicitamos loops.jsp desde un navegador, veríamos una salida similar a la de la Figura 48.

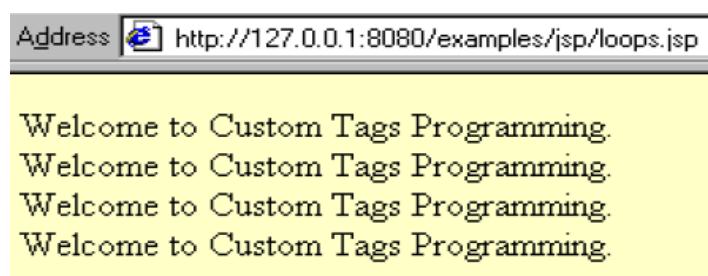


Figura 48: Bucle con etiquetas personalizadas

Guías de programación

Aquí hay unas cuantas guías a tener en mente cuando desarrollemos librerías de etiquetas JSP:

- Mantenerla simple: si una etiqueta requiere muchos atributos, debemos intentar dividirla en varias etiquetas.



- Hacerla utilizable: consultemos a los usuarios de las etiquetas (desarrolladores HTML) para conseguir un alto grado de utilizabilidad.
- No inventemos un lenguaje de programación en JSP: no desarrollemos etiquetas personalizadas que permitan a los usuarios escribir programas explícitos.
- Intentemos no re-inventar la rueda: ya existen varias librerías de etiquetas JSP a nuestra disposición, como las del Jakarta Taglibs Project. Deberemos chequearlas para ver si ya existe alguna etiqueta que nos pueda servir y no tengamos que re-inventar la rueda.

Capítulo 8

Patrón: Model View Controller

INTRODUCCIÓN

¿QUÉ ES UN PATRÓN DE DISEÑO?

Son soluciones simples y elegantes a problemas específicos y comunes del diseño orientado a objetos. Son soluciones basadas en la experiencia y que se ha demostrado que funcionan.



Figura 49: Introducción a patrones de diseño.

¿POR QUÉ DEBEMOS UTILIZAR PATRONES DE DISEÑO?



Figura 50: ¿Por qué debemos usar patrones?

Un patrón de diseño es el trabajo de una persona que ya se encontró con el problema anteriormente, intentó muchas soluciones posibles, escogió y describió una de las mejores. Y esto es algo que deberíamos aprovechar.



En otras palabras, brindan una solución ya probada y documentada a problemas de desarrollo de software que están sujetos a contextos similares. Debemos tener presente los siguientes elementos de un patrón:

- Su nombre.
- El problema (cuando aplicar un patrón).
- La solución (descripción abstracta del problema).
- Las consecuencias (costos y beneficios).

CLASIFICACIÓN DE LOS PATRONES

Los patrones se clasifican de la siguiente manera:

- **Patrones Creacionales:** Inicialización y configuración de objetos.
- **Patrones Estructurales:** Separan la interfaz de la implementación. Se ocupan de cómo las clases y objetos se agrupan, para formar estructuras más grandes.
- **Patrones de Comportamiento:** Más que describir objetos o clases, describen la comunicación entre ellos.

En este caso nos ocuparemos del patrón Model View Controller (MVC) que se encuentra dentro del grupo de patrones creacionales.

PATRÓN: MODEL VIEW CONTROLLER

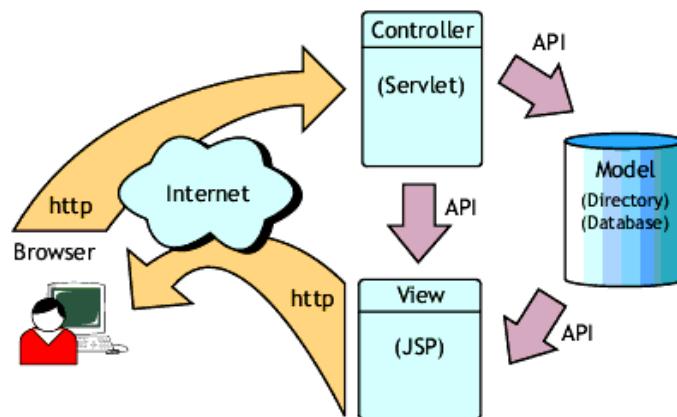


Figura 51: Patrón MVC

Introducción

En cada programa podemos encontrar tres responsabilidades principales:

- La lógica de la aplicación (qué hacer).
- Los datos sobre los que se opera (sobre qué datos se opera y cómo se organizan esto).
- La presentación de la aplicación (como se expone a sus usuarios, pueden ser otras aplicaciones).

Cada una de estas responsabilidades puede ser implementada por una o más clases. Normalmente más de una.

Orígenes del patrón MVC

Este patrón fue descrito por primera vez por Trygve Reenskaug en 1979, y la implementación original fue realizada en Smalltalk en los laboratorios Xerox.

El patrón MVC le da nombre a las 3 responsabilidades anteriores:

- Controlador: lógica.
- Modelo: datos.
- Vista: representación.
 - No decimos “interfaz” porque, si bien la GUI es una forma de vista (y probablemente la más común), no toda vista es una GUI.

Además nos dice como debiesen ser las relaciones entre las tres componentes que las implementan:

Se usa (él o alguna de sus variantes) en la gran mayoría de las interfaces de usuario.

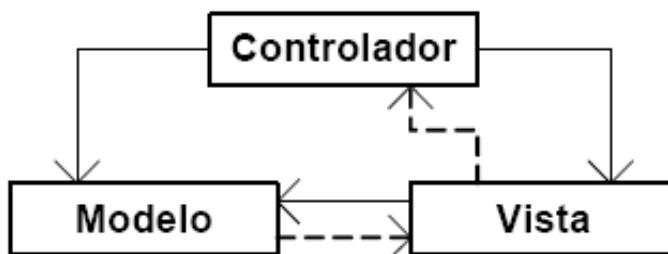


Figura 52: Relación entre los componentes del patrón MVC



El controlador

- El controlador es el que dirige la orquesta de la aplicaci\'On.
 - Este define que se ha de hacer y cuando.
 - Como se ha de responder ante un evento de la interfaz gr\'afica (si es que la hay).
 - Como se ha de responder ante un mensaje por la red.
 - Como se han de producir los resultados.
 - Que vistas se han de usar.
 - Etc.

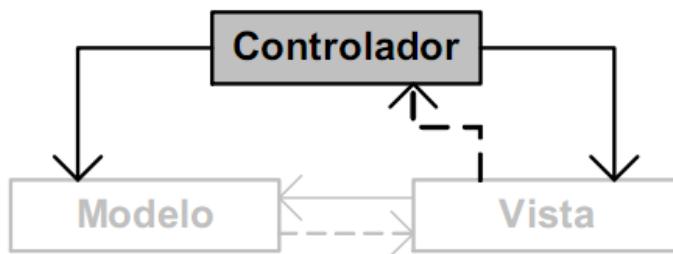


Figura 53: MVC – El Controlador

- El controlador conoce tanto al modelo de datos subyacente como a la vista que se est\'a usando.
 - Necesita que \'as\' sea, dado que tiene que orquestarlos.
 - Es posible que no conozca directamente a la vista misma, sino que s\'olo a una interfaz de esta, de tal manera de pueda cambiarla sin problemas.
- Adem\'as recibe notificaciones por parte de la vista sobre cambios que han ocurrido.
 - Un usuario que hace click en un bot\'on, por ejemplo.
 - La vista notifica (evento) al controlador, y es \'este el que toma la acci\'on que corresponda, potencialmente actualizando al modelo o a la vista.

El modelo

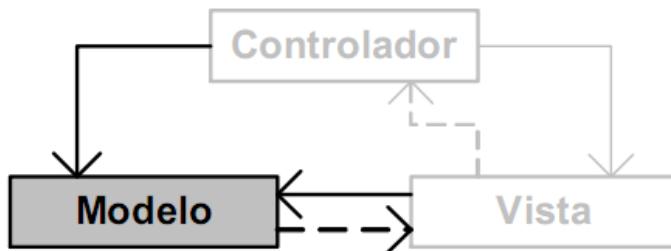


Figura 54: MVC - El Modelo

- El modelo es la parte de la aplicación que representa los datos sobre los cuales se va a operar.
 - Este puede ser tan complejo como se necesite para satisfacer las necesidades de representación de los datos.
 - Entrega valor agregado a los datos (por ejemplo, calcular la edad a partir de la fecha de nacimiento).
 - Encapsula cualquier forma de almacenamiento de los datos (por ejemplo, si salen de una base de datos o de un archivo).
- El modelo es actualizado/modificado por el controlador.
- Además es capaz de levantar notificaciones cuando sus datos cambian, de manera de que, quien sea que lo esté observando, se entere de los cambios.
- Nótese que el modelo no conoce ni al controlador ni a la vista.
 - Evitamos acoplamiento innecesario.
- El mismo modelo puede ser compartido por aplicaciones diferentes.
 - Por ejemplo, en un juego, tanto el juego mismo como el editor de niveles tienen que poder operar sobre los mismos datos, por lo que tiene sentido que el modelo sea el mismo.
 - Como no conoce a nadie, no está amarrado a nada.



La vista

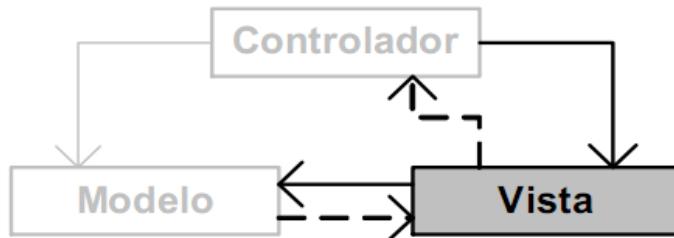


Figura 55: MVC – La Vista

- La vista es la frontera de comunicación con el mundo exterior.
 - Es quién se encarga de dar una representación al modelo.
- Normalmente su responsabilidad se confunde con la de la GUI.
 - Aunque no son estrictamente lo mismo, en adelante hablaremos indistintamente de la vista y de la GUI.
- La vista está expuesta al usuario.
 - Este interactúa con ella para lograr las metas que se propone.
 - La vista notifica al controlador.
 - El controlador toma las acciones que corresponda, potencialmente actualizando del modelo de datos.
 - El modelo, de cambiar, notifica (en este caso a la vista) de manera que actualice su estado (si corresponde).

Resumen

- Acoplamiento vista/lógica/datos es un problema.
- Son todas responsabilidades diferentes e independientes.
 - Hay que separarlas.
- El patrón MVC nos dice cómo separarlas.
 - Y quienes tienen que conocer a quienes para que la solución sea extensible, escalable, etc.
- El controlador se encarga de la lógica.

- Es el que define qué se tiene que hacer, cuándo y cómo.
- El modelo se encarga de los datos.
 - Es capaz de avisar cuando cambia.
- La vista se encarga de la interacción con el usuario.
 - Muestra una representación del modelo.

VARIANTES DEL PATRÓN MVC

Variante inicial

Variante en la cual no existe ninguna comunicación entre el Modelo y la Vista y esta última recibe los datos a mostrar a través del Controlador.

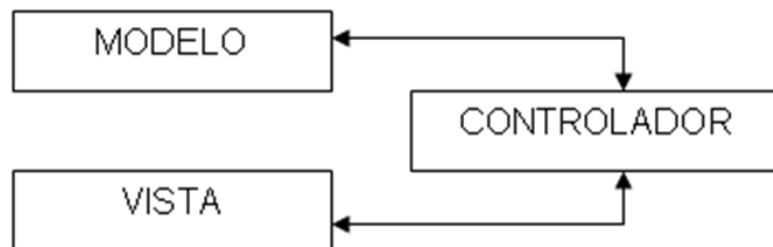


Figura 56: Variante inicial del patrón MVC.

Variante intermedia

Variante en la cual se desarrolla una comunicación entre el Modelo y la Vista, donde esta última para mostrar los datos los busca directamente en el Modelo, dada una indicación del Controlador, disminuyendo el conjunto de responsabilidades de este último.

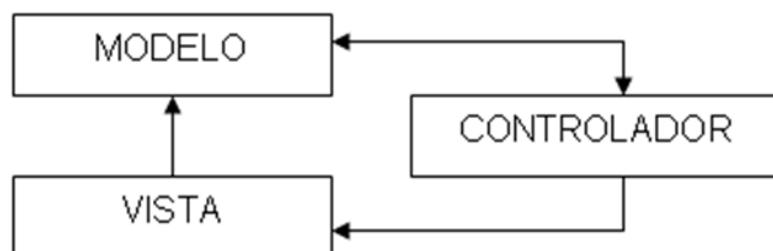


Figura 57: Variante intermedia del patrón MVC.



Capítulo 9

jQuery

INTRODUCCIÓN

El mejor resumen de lo que es jQuery lo podemos encontrar en el lema de su propia página web: “La librería JavaScript para escribir menos y hacer más”. Ampliando algo más esta definición, dejémoslo en que es una forma de convertir el desarrollo de la parte de cliente de una aplicación web en algo mucho más divertido, rápido y sencillo, facilitando la interacción con los elementos del árbol de documento, el manejo de eventos, el uso de animaciones, etc.

jQuery se ha convertido en la librería JavaScript más utilizada actualmente, y es que, además, es gratuita, de código abierto (bajo licencia MIT y GPL v2) e increíblemente ligera. Entre sus usuarios podemos encontrar a Google, Microsoft, IBM, Amazon, Twitter, WordPress, Mozilla o Drupal.

Para poder utilizar esta librería lo primero que tendremos que hacer será incluir su código en nuestro proyecto. Podemos descargar el script desde su página web, subirlo a nuestro servidor, y ejecutarlo con la etiqueta script:

```
<script type="text/javascript" src="jquery.js"></script>
```

También podemos cargarla directamente desde el CDN que mantiene Google:

```
<script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js"></script>
```

El de Microsoft:

```
<script type="text/javascript"
src="http://ajax.microsoft.com/ajax/jquery/jquery-1.4.2.min.js"></script>
```

O el del propio jQuery

```
<script type="text/javascript" src="http://code.jquery.com/jquery-1.4.2.min.js"></script>
```

Independientemente de la opción elegida, ya estaremos listos para comenzar a trabajar con la librería. Sin embargo, y aunque no es estrictamente necesario, os aconsejaría tener unos conocimientos básicos de JavaScript primero. Y ahora, manos a la obra.

El corazón de jQuery es la función sobrecargada del mismo nombre, **jQuery**, que ofrece distinta funcionalidad dependiendo de los parámetros utilizados. Además, como JavaScript también toma conceptos del paradigma funcional, y las funciones no son



más que otro tipo de objeto, la función jQuery cuenta a su vez con distintas propiedades y métodos. La intención de esta decisión de diseño es la de evitar llenar el espacio de nombres global con montones de nombres inútiles esperando a colisionar.

Si queremos escribir menos y que nuestros archivos sean más pequeños, y si no utilizamos ninguna otra librería que pueda causar algún conflicto con este símbolo, también podemos utilizar el alias \$ en sustitución de **jQuery**.

SELECTORES JQUERY

El primer paso a la hora de trabajar con jQuery es seleccionar los elementos del árbol de documento sobre los que queremos trabajar. Para ello se utiliza la función jQuery, pasando como argumento a la función una cadena con un selector, la mayoría de los cuales utilizan una sintaxis similar a la de CSS 3. El engine que utiliza jQuery para seleccionar elementos se llama Sizzle, y puede descargarse y utilizarse de forma aislada.

Selectores básicos

Selector universal: selecciona todos los elementos de la página:

```
jquery("*")
```

Selector de tipo o etiqueta: selecciona todos los elementos con el tipo de etiqueta indicado:

```
jQuery("div")
```

Selector de clase: selecciona todos los elementos con la clase indicada (atributo class):

```
jQuery("div.entrada")
```

Selector de identificador: selecciona el elemento con el identificador (atributo id) indicado:

```
jQuery("div#cabeza")
```

Grupos de selectores: se puede combinar el resultado de varios selectores distintos separándolos con comas:

```
jquery("p,div,a")
```

Selectores de atributos

Selector de atributo: selecciona elementos que tengan un cierto atributo:



```
jquery("a[rel]")
```

También se puede seleccionar aquellos que tengan un cierto valor para un atributo:

```
jquery("a[rel='nofollow']")
```

O que tengan un valor distinto del indicado (jQuery):

```
jQuery("a[rel!='nofollow']")
```

Aquellos cuyo valor empieza por una cierta cadena:

```
jquery("a[href^='http://mundogeek.net/]")
```

Los que terminan con una cierta cadena:

```
jquery("a[href$='.com']")
```

Y los que contienen una cierta cadena:

```
jquery("a[href*='geek']")
```

Por último, podemos hacer combinaciones de todo lo anterior:

```
jquery("a[rel='nofollow'][href']")
```

Selectores de widgets

Pseudo clase botón: selecciona todos los botones (jQuery):

```
jquery(":button")
```

Pseudo clase checkbox: selecciona todos los checkboxes (jQuery):

```
jquery(":checkbox")
```

Pseudo clase archivo: selecciona todos los widgets de tipo archivo (jQuery):

```
jquery(":file")
```

Pseudo clase cabeceras: selecciona todas las cabeceras (jQuery);

```
jquery(":header")
```

Pseudo clase imagen: selecciona todas las imágenes (jQuery):

```
jquery(":image")
```

Pseudo clase input: selecciona todos los widgets de tipo input (jQuery):

```
jquery(":input")
```

Pseudo clase contraseña: selecciona todos los elementos password (jQuery):

```
jquery(":password")
```

Pseudo clase radiobutton: selecciona todos los elementos radiobutton (jQuery):

```
jquery(":radio")
```



Pseudo clase reset: selecciona todos los elementos reset (jQuery):

```
jquery(":reset")
```

Pseudo clase seleccionado: selecciona las opciones seleccionadas en un select (jQuery):

```
jquery(":select")
```

Pseudo clase submit: selecciona todos los elementos submit (jQuery):

```
jquery(":submit")
```

Pseudo clase texto: selecciona todos las cajas de texto (jQuery):

```
jquery(":text")
```

Pseudo clase marcado: selecciona todos los radiobuttons y checkboxes marcados:

```
jquery(":checked")
```

Pseudo clase activo: selecciona todos los elementos que estén activos:

```
jquery("input:enabled")
```

Pseudo clase inactivo: selecciona todos los elementos que no estén activos:

```
jquery("input:disabled")
```

Pseudo clase ocultos: selecciona todos los elementos ocultos (jQuery):

```
jquery(":hidden")
```

Pseudo clase visible: selecciona todos los elementos visibles (jQuery):

```
jquery(":visible")
```

Otros selectores

Pseudo clase animado: selecciona todos los elementos que están en proceso de animación en este momento (jQuery):

```
jquery(":animated")
```

Pseudo clase contiene: selecciona todos los elementos que contengan el texto indicado, directamente o en uno de los hijos (jQuery):

```
jquery("div:contains('Mundo geek')")
```

Pseudo clase tiene: selecciona todos los elementos que contengan al menos un elemento que responda al selector indicado (jQuery):

```
jquery("div:has(h2)")
```

Pseudo clase negación: selecciona todos los elementos que no cumplan con el selector dado:



```
jquery("div:not(.entrada)")
```

GESTIONANDO UNA COLECCIÓN JQUERY

Al llamar a la función *jQuery* con un selector como argumento, el valor devuelto será otro objeto *jQuery* representando la colección de elementos DOM seleccionados. Tanto es así, que podremos obtener uno de los elementos utilizando el operador `[]`, como si de un array normal se tratara:

```
jQuery("div.entrada")[0]
```

Y también tenemos acceso a una propiedad *length* con el número de elementos que contiene la colección:

```
jQuery("div.entrada").length
```

Otras cosas que podemos hacer con este objeto son añadir elementos a la colección:

```
jQuery("div.entrada").add("div.comentario")
```

Eliminar elementos:

```
jQuery("div.entrada").not("div.destacada")
```

Filtrar elementos con un selector o basándonos en el valor devuelto por una función:

```
jQuery("div.entrada").filter(":has(h2)")
```

Quedarnos con los elementos que contengan otro cierto elemento:

```
jQuery("div.entrada").has("h2")
```

Obtener un sólo elemento, por su índice:

```
jQuery("div.entrada").eq(3)
```

Obtener el primer elemento de la colección:

```
jQuery("div.entrada").first()
```

Obtener el último elemento de la colección:

```
jQuery("div.entrada").last()
```

Crear una sub colección a partir de la original:

```
jQuery("div.entrada").slice(0,5);  
jQuery("div.entrada").slice(3);
```

Obtener los descendientes directos:

```
jQuery("div.entrada").children()  
jQuery("div.entrada").children("p")
```



Obtener los descendientes directos, incluyendo el texto plano

```
jQuery("div.entrada").contents()
```

Obtener los hijos que cumplan con un determinado selector:

```
jQuery("div.entrada").find("p")
```

Obtener el hermano siguiente:

```
jQuery("div.entrada").next()
```

Obtener los hermanos siguientes

```
jQuery("div.entrada").nextAll()
```

Obtener el hermano anterior:

```
jQuery("div.entrada").prev()
```

Obtener los hermanos anteriores:

```
jQuery("div.entrada").prevAll()
```

Obtener todos los hermanos:

```
jQuery("div.entrada").siblings()
```

Obtener el parente de cada elemento:

```
jQuery("div.entrada").parent()
```

Obtener todos los ancestros de cada elemento:

```
jQuery("div.entrada").parents()
```

Crear una copia de la colección:

```
jQuery("div.entrada").clone()
```

O buscar la posición que ocupa un elemento en la colección (si existe):

```
jQuery("*").index("div.entrada")
```

MODIFICAR LA PÁGINA CON JQUERY

Veamos ahora cómo utilizar jQuery para modificar nuestra página web. Podemos, por ejemplo, modificar el valor de un atributo:

```
jQuery("a#principal").attr("href", "http://mundogeek.net/")
```

Añadir una nueva clase a uno o varios elementos:

```
jQuery("div.entrada:first").addClass("primera")
```

Añadir una propiedad CSS a uno o varios elementos:



```
jQuery("div.entrada").css("border", "1px solid red")
```

Añadir contenido a un elemento:

```
jQuery("div.entrada:first").before("<strong>Destacada</strong>")
```

```
jQuery("div.entrada:first").prepend("<strong>Destacada</strong>")
```

```
jQuery("div.entrada:first").after("<strong>Destacada</strong>")
```

```
jQuery("div.entrada:first").append("<strong>Destacada</strong>")
```

Modificar el contenido de un elemento:

```
jQuery("p").html("<strong>Sustituido</strong>")
```

Eliminar un elemento de la página:

```
jQuery("div.entrada:first").remove()
```

Ocultar un elemento:

```
jQuery("p").hide()
```

O volver a mostrar un elemento:

```
jQuery("p").show()
```

EVENTOS EN JQUERY

Existen distintas funciones para asignar una función que maneje un evento lanzado por un widget. Para el evento *click*, al hacer clic sobre un elemento:

```
jQuery(":button#pulsame").click(function () {  
    alert("Has hecho clic");  
});
```

Evento *submit*, cuando se pulsa sobre el botón de enviar de un formulario:

```
jQuery("#formulario").submit(function() {  
    alert("Enviando");  
});
```

Evento *dblclick*, al hacer doble clic sobre un elemento:

```
jQuery("p:first").dblclick(function () {  
    alert("Has hecho doble clic");  
});
```



Evento *hover*, cuando al pasar el ratón por encima de un elemento. Podemos utilizar *jQuery(this)* para referirnos a este elemento:

```
jQuery("p:first").hover(function () {  
    jQuery(this).css("border", "1px solid red");  
});
```

Evento *mouseenter*, cuando el cursor entra en un elemento:

```
jQuery("p:first").mouseenter(function () {  
    jQuery(this).css("border", "1px solid red");  
});
```

Evento *mouseout*, cuando el cursor sale de un elemento:

```
jQuery("p:first").mouseenter(function () {  
    jQuery(this).css("border", "1px solid red");  
});  
  
jQuery("p:first").mouseout(function () {  
    jQuery(this).css("border", "0");  
});
```

Evento *change*, cuando se modifica un elemento:

```
jQuery(":text#nombre").change(function () {  
    alert("Cambiado");  
});
```

Evento *load*, cuando se termina de cargar el elemento:

```
jQuery(window).load(function () {  
    alert("Página cargada");  
});
```

Evento *ready*, cuando se termina de cargar el DOM, para no tener que esperar a cargar las imágenes, por ejemplo, de forma que el usuario pueda utilizar nuestra funcionalidad JavaScript cuanto antes:

```
jQuery(document).ready(function () {  
    alert("DOM cargado");  
});
```

Esto último, al ser esto algo muy común, se puede resumir pasando una función a la función *jQuery*, directamente:

```
jQuery(function () {  
    alert("DOM cargado");  
});
```



ANIMACIONES CON JQUERY

jQuery viene con unas pocas animaciones útiles y vistosas por defecto, aunque para sacarle todo el partido probablemente tendremos que recurrir a plugins.

Para hacer un fundido a opaco:

```
jQuery(function () {  
    jQuery("p").hide();  
    jQuery("p").delay(200).fadeIn();  
});
```

En el ejemplo anterior se utiliza delay para hacer pasar un par de segundos y que se vea más claramente el efecto. A esta función se le puede pasar un valor numérico con el número de milisegundos a esperar, o una cadena, como "slow" (lento) o "fast" (rápido).

Para hacer un fundido a transparente:

```
jQuery(":button").click(function () {  
    jQuery("p").fadeOut();  
});
```

También podemos cambiar la opacidad de un elemento a cualquier valor intermedio

```
jQuery(":button").click(function () {  
    jQuery("p").fadeTo("slow", 0.5);  
});
```

Mostrar los elementos con una animación de deslizamiento de arriba a abajo:

```
jQuery(function () {  
    jQuery("p").hide().delay(200).slideDown();  
});
```

Ocultarlos deslizándolos hacia arriba:

```
jQuery(function () {  
    jQuery("p").delay(200).slideUp();  
});
```

Mostrarlos u ocultarlos, dependiendo de si se estaban mostrando o no:

```
jQuery(":button").click(function () {  
    jQuery("p").delay(200).slideToggle();  
});
```

Por último, para cualquier otro tipo de animación con propiedades CSS cuyos valores sean numéricos, utilizaríamos:



```
jQuery(":button").click(function () {
    jQuery("p").animate({opacity:0.50,width:100}, 'slow');
});
```

JQUERY Y AJAX

La forma más sencilla de enviar una petición HTTP de forma asíncrona y mostrar el resultado en la página actual es utilizar la función *load*. Esta se ejecuta sobre el elemento al que se va a añadir la respuesta, y se le pasa como argumento una cadena con el archivo a cargar. Esta cadena puede contener también un selector con el que seleccionar qué elementos queremos mostrar de la respuesta.

```
jQuery(":button").click(function () {
    $("#citas").load("citas.html li");
});
```

También se pueden enviar parámetros al documento (se utiliza GET a menos que los datos se manden en forma de objeto):

```
jQuery(":button#login").click(function () {
    $("#mensaje").load("login.jsp", {nombre: "zootropo", pass: "contraseña"});
});
```

E indicar una función a ejecutar cuando se termine de llevar a cabo la petición

```
jQuery(":button#login").click(function () {
    jQuery("#mensaje").load("login.jsp", {nombre: "zootropo", pass: "contraseña"}, 
        function(responseText, textStatus, XMLHttpRequest) {
            alert("cargado");
        });
});
```

También se pueden utilizar los métodos *get* y *post* del objeto jQuery, en cuyo caso se nos devolverá unos ciertos datos con los que nosotros mismos tendremos que trabajar para generar la respuesta y mostrarla en nuestro documento actual:

```
jQuery.get("login.php", {nombre: "zootropo", pass:"contraseña"}, 
    function(data, textStatus, XMLHttpRequest){
        jQuery("#mensaje").html("Han devuelto: " + data);
    });
});
```



```
jQuery.post("login.php", {nombre: "zootropo", pass:"contraseña"},  
    function(data, textStatus, XMLHttpRequest){  
        jQuery("#mensaje").html("Han devuelto: " + data);  
    });
```

Si la respuesta del servidor va a estar en formato JSON (JavaScript Object Notation), muy utilizado actualmente, podemos utilizar el método `jQuery.getJSON()`, al que se le pasa la URL de la petición y, opcionalmente, cualquier parámetro que se necesite, además de una función de callback que ejecutar cuando se termine con la petición. Este método se encargará de parsear la estructura del objeto JSON devuelta utilizando `jQuery.parseJSON()`, objeto que estará disponible como primer parámetro de la función de callback.

```
$.getJSON('tareas.php', function(data, textStatus){  
    $.each(data, function(i, tarea){  
        $("<li></li>").html(tarea.nombre + " - " + tarea.hora).appendTo("ul#tareas");  
    });  
});
```

El código de este `tareas.php` podría tener este aspecto:

```
<?php  
header('Content-type: text/javascript');  
$bbdd = new mysqli('servidor.com', 'usuario', 'pass', 'tareas');  
$query = 'SELECT * FROM tareas';  
$tareas = array();  
if($resultado = $bbdd->query($query))  
    while($tarea = $resultado->fetch_object())  
        $tareas[] = $tarea;  
    $bbdd->close();  
    echo json_encode($tareas);  
?>
```

Como se puede observar, se utiliza la función `json_encode` para convertir el array u objeto PHP a formato JSON. Esta función, junto con su complemento, `json_decode`, se introdujo en PHP en la versión 5.2.0.



Capítulo 10

Fuentes de Datos

FUENTES DE DATOS BÁSICAS

Uno de los principales beneficios de usar el API JDBC es facilitar una programación independiente de la base de datos, así muchas de las aplicaciones JDBC pueden transferirse fácilmente a otra base de datos diferente. Sin embargo, todavía hay dos ítems que permanecen unidos a una base de datos particular, la clase JDBC Driver y la URL JDBC. Con la introducción de las fuentes de datos en el API JDBC 2.0, se eliminaron incluso estas dependencias.

Esencialmente un objeto *DataSource* representa una fuente de datos particular en una aplicación Java. Además de encapsular la información específica de la base de datos y del driver JDBC en un sólo objeto estándarizado, las fuentes de datos pueden actuar como una factoría de *Connection* y proporcionar métodos para seleccionar y obtener propiedades particulares que requiere el objeto *DataSource* para una operación satisfactoria. Algunas propiedades estándar que podría requerir un objeto *DataSource* incluyen:

- databaseName
- serverName
- portNumber
- userName
- password

Un beneficio adicional de usar una *DataSource*, que podrías haber adivinado de la lista anterior, es que la información sensible relacionada con la seguridad como el nombre de usuario, el password, e incluso el servidor de la base de datos están codificados sólo en un lugar, lo que puede hacer un administrador de sistemas. Mientras que la interacción con un objeto *DataSource* se puede hacer con una aplicación gráfica, es instructivo ver como trabajan realmente los ejemplos. Aunque los conceptos de un objeto *DataSource* son bastante simples, para usarlo dentro de una aplicación Java, un objeto *DataSource* es referenciado usando Java Naming and Directory Interface, o JNDI. Antes de saltar dentro del código de ejemplo de



DataSource , el siguiente punto presenta conceptos relevantes de JNDI que son necesarios para usar apropiadamente un objeto DataSource .

REPASO RÁPIDO DE JNDI

JNDI es un API Java que encapsula el concepto de servidores de nombres y directorios de la misma forma que JDBC encapsula los conceptos que hay detrás de la comunicación con una base de datos. Aunque podría parecer confuso, es bastante sencillo, todos los usuarios de ordenadores usan servicios de nombres y directorios todos los días. Por ejemplo, los discos duros trabajan con pistas y sectores, aunque un usuario sólo se preocupa de nombres de ficheros y directorios. El sistema de ficheros maneja el servicio de nombrado que asocia un nombre de fichero dado con una localización específica en el disco duro. Otro ejemplo es la Web, donde la mayoría de los usuarios sólo se preocupan del nombre de la Web site, como www.isil.pe, y no de la dirección IP subyacente. Sin embargo, la comunicación TCP/IP se hace usando la dirección IP y no usando el nombre que leemos los humanos. La transformación entre las dos representaciones la realiza el DNS, o Domain Name System.

Aunque JDNI proporciona un API amplio y útil por sí mismo, nuestras necesidades son considerablemente simples. Necesitamos conocer como hacer cuatro cosas:

- Crear un nombre y unirlo a un objeto Java.
- Buscar un nombre para recuperar un objeto Java.
- Borrar un nombre.
- Re-unir un nombre a nuevo objeto Java.

POOL DE CONEXIONES

Usando Tomcat 5.x, 6.x

1. Crear el archivo llamado **context.xml** dentro de la carpeta **META-INF**.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/demopool" reloadable="true" crossContext="true">

<Resource
    name="jdbc/poolTest"
    auth="Container"
    type="javax.sql.DataSource"
```



```
maxActive="100"
maxIdle="30"
maxWait="10000"
username="root"
password="adminadmin"
driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/Test?autoReconnect=true"/>
</Context>
```

2. Modifique **el web.xml** y agregue lo siguiente.

```
<resource-ref>
  <res-ref-name>jdbc/poolTest</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

3. Usar el Pool de conexiones

```
//Usar JNDI
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("java:comp/env/jdbc/poolTest");
//Obtener la conexión del Pool
Connection cn = ds.getConnection();
System.out.println("Conexión OK");
//Devolver la conexión al Pool
cn.close();
```

4. Recomendación

Copie el **mysql-connector-java-xxx-bin.jar**, en la carpeta **lib** de donde está instalado tomcat.



Usando GlassFish v2, v3

1. Crear el archivo llamado **sun-web.xml** dentro de la carpeta **META-INF**.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc./DTD Application Server
9.0 Servlet 2.5//EN" "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-
0.dtd">

<sun-web-app error-url="">
    <context-root>/demopool</context-root>
    <class-loader delegate="true"/>
        <jsp-config>
            <property name="keepgenerated" value="true">
                <description></description>
            </property>
        </jsp-config>
</sun-web-app>
```

2. Para ello entramos a la consola administrativa de GlassFish <http://localhost:4848> (usuario: **admin** y la contraseña: **adminadmin** por default). Luego, en el panel de la izquierda seleccionamos **Recursos > JDBC > Conjunto de conexiones** y hacemos clic en el botón superior **Nuevo**.



3. Damos clic en **Siguiente**. Y en la siguiente ventana buscaremos las propiedades necesarias para establecer la conexión a la base de datos.

- databaseName: Test
- ServerName: localhost
- user: root
- password: adminadmin

Propiedades adicionales (155)		
	Agregar propiedad	Eliminar propiedades
Nombre	Valor	
User	root	
EnableQueryTimeouts	true	
AutoReconnectForPools	false	
UseInformationSchema	false	
NoAccessToProcedureBodies	false	
LoggerClassName	com.mysql.jdbc.log.StandardLogger	
UltraDevHack	false	
AllowMultiQueries	false	
ServerName	localhost	

4. Hacemos clic en **Finalizar** y para probar la conexión haga clic en **Sondeo**.

Recursos > JDBC > Conjuntos de conexiones > poolTest

General	Avanzado	Propiedades adicionales
 Ping realizado con éxito		
Editar conjunto de conexiones Modifique los conjuntos de conexiones JDBC existentes. Un conjunto de conexiones JDBC es un grupo de conexiones que comparten una determinada base de datos. <input type="button" value="Cargar predeterminados"/> <input type="button" value="Sondeo"/>		
Configuración general Nombre: poolTest		

5. Ahora, debemos crear el **Recurso JDBC**. Para ello, en el panel izquierdo entramos a **Recursos > JDBC > Recursos JDBC** y le damos clic en **Nuevo**. Asignamos el nombre **JNDI** para invocarlo desde Java y hacemos la referencia al **Pool** que acabamos de crear.

Recursos > JDBC > Recursos JDBC

Nuevo recurso JDBC

Especifique un nombre JNDI exclusivo que identifique el recurso JDBC que desea crear. El nombre no debe contener espacios ni caracteres de subrayado, guiones o puntos.

Nombre JNDI: *	<input type="text" value="jdbc/poolTest"/>
Nombre de conjunto: *	<input type="text" value="poolTest"/>
Descripción:	<input type="text"/>
Estado:	<input checked="" type="checkbox"/> Activado

6. Usar el Pool de conexiones



```
//Usar JNDI
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/poolTest");
//Obtener la conexión del Pool
Connection cn = ds.getConnection();
System.out.println("Conexión OK");
//Devolver la conexión al Pool
cn.close();
```

7. Recomendación

Copie el **mysql-connector-java-xxx-bin.jar**, en la carpeta **lib\ext** de donde está instalado **GlassFish**.



Usando JBoss 5.x, 6.x

1. Crear el archivo llamado **mysql-ds.xml** dentro de la carpeta **server\default\deploy**.

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

<datasources>
    <local-tx-datasource>
        <jndi-name>poolTest</jndi-name>
        <connection-url>
            jdbc:mysql://localhost:3306/test?autoReconnect=true
        </connection-url>
        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <user-name>root</user-name>
        <password>adminadmin</password>
        <min-pool-size>5</min-pool-size>
        <max-pool-size>20</max-pool-size>
        <idle-timeout-minutes>5</idle-timeout-minutes>
    </local-tx-datasource>
</datasources>
```

2. Usar el Pool de conexiones

```
//Usar JNDI
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("java:poolTest");
//Obtener la conexión del Pool
Connection cn = ds.getConnection();
System.out.println("Conexión OK");
//Devolver la conexión al Pool
cn.close();
```

3. Recomendación

Copie el **mysql-connector-java-xxx-bin.jar**, en la carpeta **server\default\lib** de donde está instalado **jboss**.