



Lenguaje de Programación II

Faltan Créditos

ÍNDICE

Presentación	5
Red de contenidos	6
Unidad de Aprendizaje 1: <i>Servlet y JSP</i>	
SEMANA 1 : Arquitectura JEE y <i>Tomcat</i>	7
SEMANA 2 : <i>Servlet</i> y JSP	21
SEMANA 3 : <i>Servlet</i> y JSP	33
SEMANA 4 <i>HttpSession</i>	53
SEMANA 5 Patrones de Diseño de <i>Software</i>	67
SEMANA 6 Patrón <i>Data Access Object</i>	85
SEMANA 7 : Examen parcial de teoría	
SEMANA 8 : Examen parcial de laboratorio	
Unidad de Aprendizaje 2: <i>Seguridad de Aplicaciones</i>	
SEMANA 9 : Manejo de Seguridad con <i>Java</i>	103
Unidad de Aprendizaje 3: <i>Standard Actions</i>	
SEMANA 10 : Manejo de Etiquetas <i>Standard Actions</i>	113
Unidad de Aprendizaje 4: <i>Custom Tag</i>	
SEMANA 11 : <i>Custom Tag</i> I	126
SEMANA 12 : <i>Custom Tag</i> II	
Unidad de Aprendizaje 5: <i>JSTL y Lenguaje de Expresiones</i>	
SEMANA 13 : Etiquetas estándares para <i>Java</i> (JSTL)	139
SEMANA 14 : Lenguaje de Expresiones (EL)	
SEMANA 15 : Examen final de laboratorio	
SEMANA 16 : Sustentación comercial de sistemas	
SEMANA 17 : Examen final de teoría	

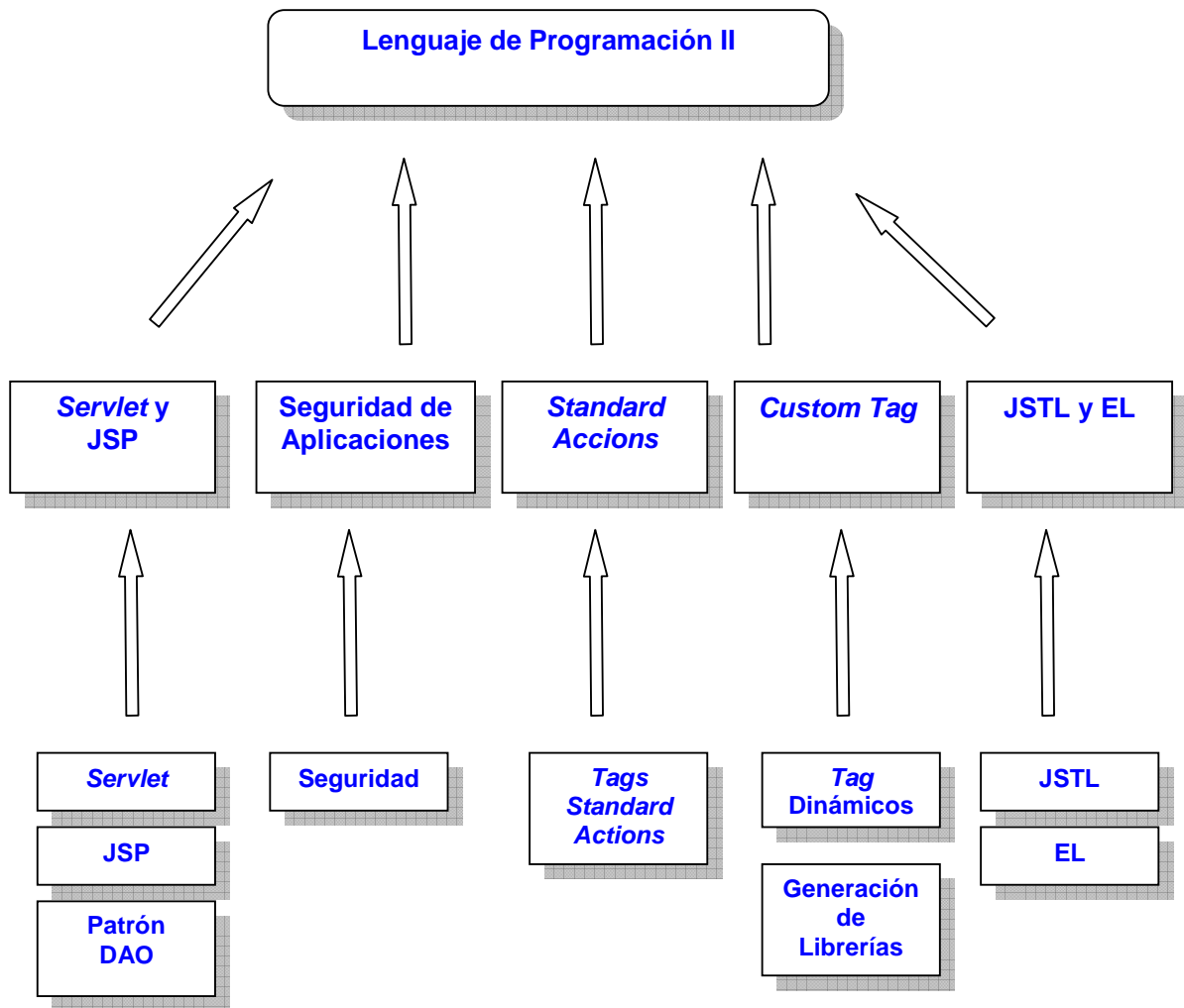
PRESENTACIÓN

Lenguaje de Programación II pertenece a la línea de cursos de programación y se dicta en la carrera de Computación e Informática. El curso brinda un conjunto de herramientas del lenguaje *Java* que permite a los alumnos utilizar para la implementación de aplicaciones web en *Java*.

El manual para el curso ha sido diseñado bajo la modalidad de unidades de aprendizaje, las que se desarrollan durante semanas determinadas. En cada una de ellas, hallará los logros, que debe alcanzar al final de la unidad; el tema tratado, el cual será ampliamente desarrollado; y los contenidos, que debe desarrollar, es decir, los subtemas. Asimismo, encontrará las actividades que deberá desarrollar en cada sesión, que le permitirán reforzar lo aprendido en la clase.

El curso es eminentemente práctico: consiste en desarrollar una aplicación web. En primer lugar, se inicia con crear programas utilizando la tecnología de *Servlet* y *JSP*. Continúa con la presentación de nuevas tecnologías como el patrón *Data Access Object*. Luego, se desarrollan los componentes reutilizables mediante el *CustomTag*. Finalmente, se concluye con el uso del lenguaje de expresiones (EL).

RED DE CONTENIDOS





SERVLET Y JSP

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos implementarán una aplicación web en *Java* que realiza operaciones *SELECT*, *INSERT*, *UPDATE*, *DELETE* a la base de datos que contenga *Servlets*, *JSP*, *JavaBeans* y procese los objetos *request* y *response*.

TEMARIO

- Arquitectura *Java EE* y el contenedor *Tomcat*

ACTIVIDADES PROPUESTAS

- Los alumnos reconocen la estructura de carpetas del contenedor de *Servlet* llamado *Tomcat*.
- Los alumnos reconocen la estructura de una aplicación web.

1. JEE

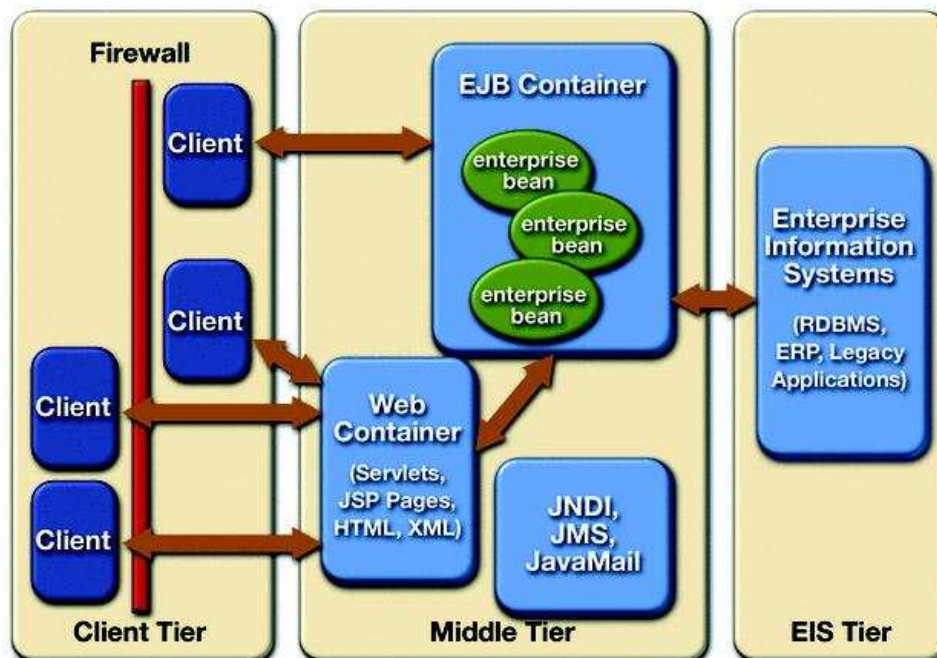
Java Platform, Enterprise Edition o Java EE (anteriormente conocido como *Java 2 Platform, Enterprise Edition o J2EE hasta la versión 1.4*), es una plataforma de programación—parte de la Plataforma Java—para desarrollar y ejecutar *software* de aplicaciones en Lenguaje de programación Java con arquitectura de N niveles distribuida, basándose ampliamente en componentes de *software* modulares ejecutándose sobre un servidor de aplicaciones. La plataforma Java EE está definida por una especificación. Similar a otras especificaciones del Java Community Process, Java EE es también considerada informalmente como un estándar debido a que los suministradores deben cumplir ciertos requisitos de conformidad para declarar que sus productos son conformes a Java EE; estandarizado por *The Java Community Process / JCP*.

Java EE incluye varias especificaciones de API, tales como JDBC, RMI, *e-mail*, JMS, Servicios Web, XML, etc. y define cómo coordinarlos. Java EE también configura algunas especificaciones únicas para Java EE para componentes. Estas incluyen *Enterprise JavaBeans*, *servlets*, *portlets* (siguiendo la especificación de *Portlets Java*), *JavaServer Pages* y varias tecnologías de servicios web. Esto permite al desarrollador crear una Aplicación de Empresa portable entre plataformas y escalable, a la vez que integrable con tecnologías anteriores. Otros beneficios añadidos son, por ejemplo, que el servidor de aplicaciones puede manejar transacciones, la seguridad, escalabilidad, concurrencia y gestión de los componentes desplegados, significando que los desarrolladores pueden concentrarse más en la lógica de negocio de los componentes en lugar de en tareas de mantenimiento de bajo nivel.

Las razones que empujan a la creación de la plataforma JEE son las siguientes:

- **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**.



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*.

1.1 Servidor de aplicaciones JEE

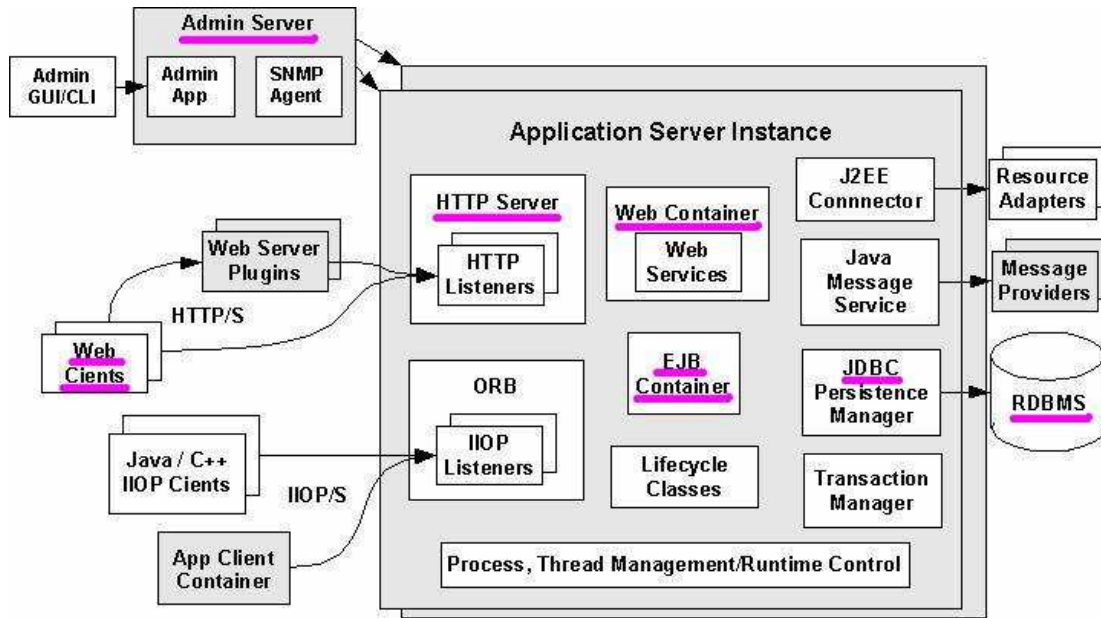
A continuación, vamos a entrar en más detalle. Para ello, hemos **subrayado en el siguiente gráfico los elementos más importantes** y usuales. La **arquitectura de un servidor de aplicaciones** incluye una serie de subsistemas:

- **Servidor HTTP** (también denominado servidor Web o servidor de páginas). Un ejemplo es 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).

Sin embargo, conviene empezar por el principio, es decir, el lenguaje básico de interconexión: el protocolo **HTTP**. Este 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.



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 CGI, 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 la aplicación).

Otros aspectos del contenedor web:

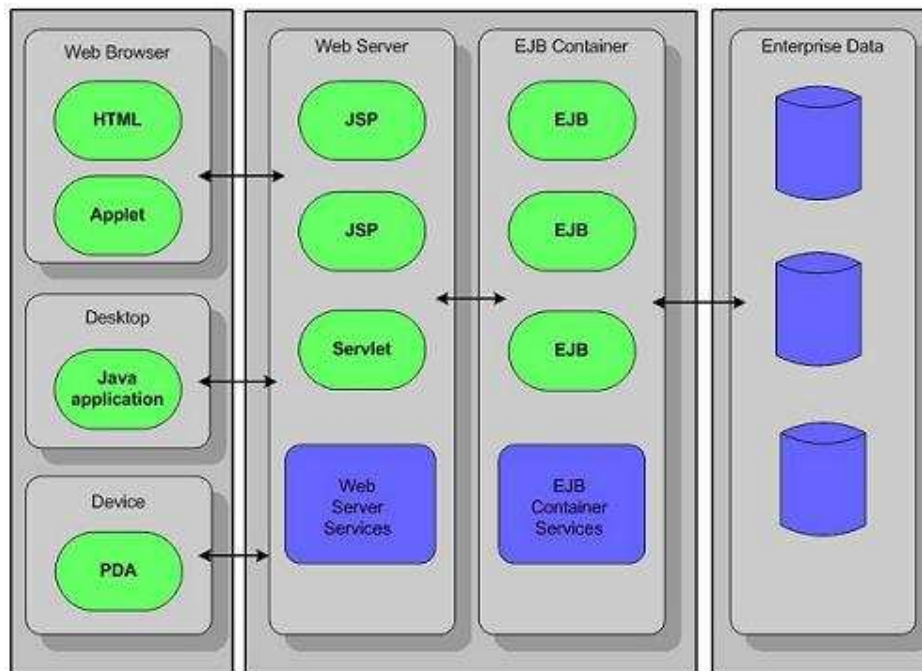
- El contenedor necesita conectores que sirven de intermediarios para comunicarse con elementos externos. Los **conectores** capacitan al AS para acceder a sistemas empresariales (backends). Por ejemplo:
 - El **Java Message Service** ofrece conectividad con sistemas de mensajería como MQSeries.
 - El 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**.

1.2 Las capas de la arquitectura

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

- 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 pueden 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**.

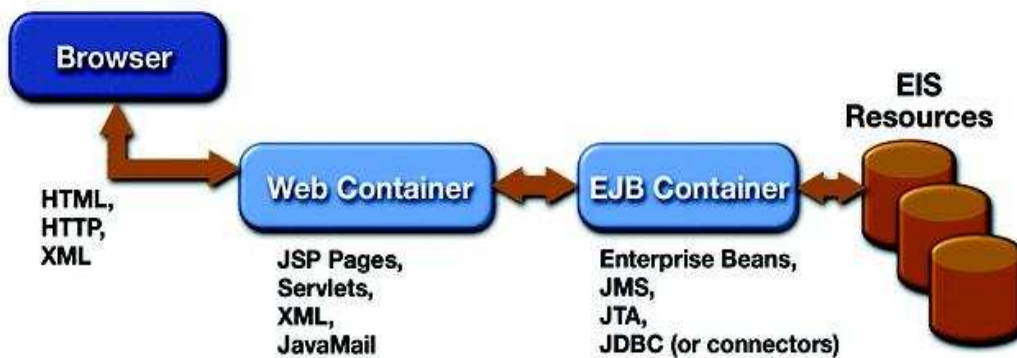
La visión de la arquitectura es un **esquema lógico**, no físico. Cuando hablamos de capas nos referimos sobre todo a **servicios** diferentes (que pueden estar físicamente dentro de la misma máquina e incluso compartir servidor de aplicaciones y JVM).



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

1.3 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:

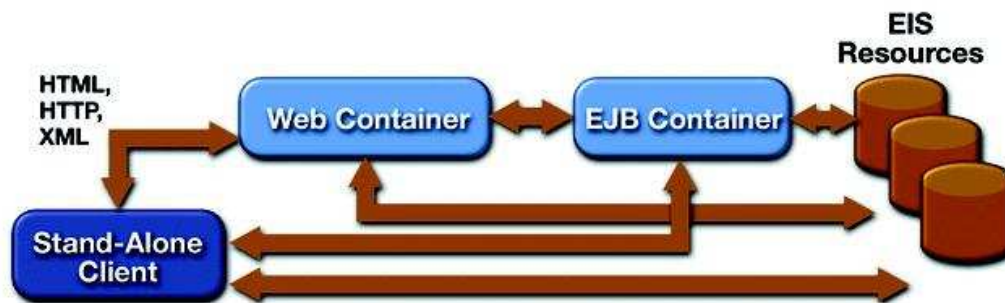


XML utiliza HTTP como su soporte para el transporte.

Una pregunta muy común es **cuándo usar *servlets* y cuándo 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).

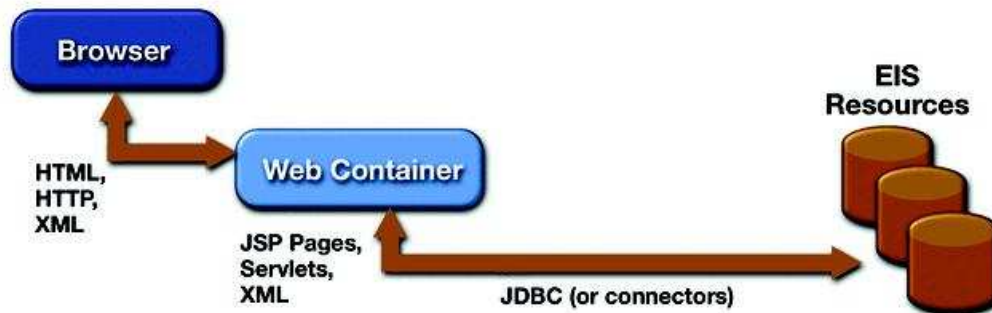
1.4 Escenario desde una aplicación

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):

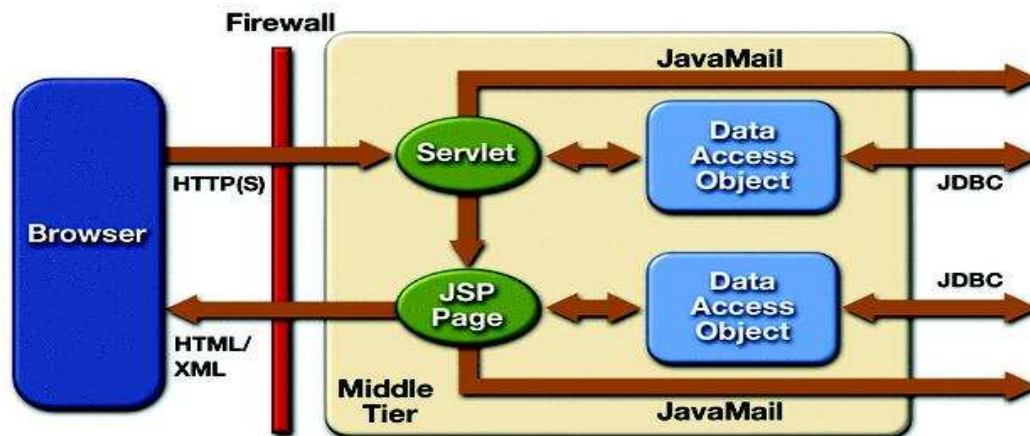


1.5 Escenario basado en la web (web-centric application)

La plataforma JEE no obliga a usar todas las capas en un sistema. 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.



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 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:

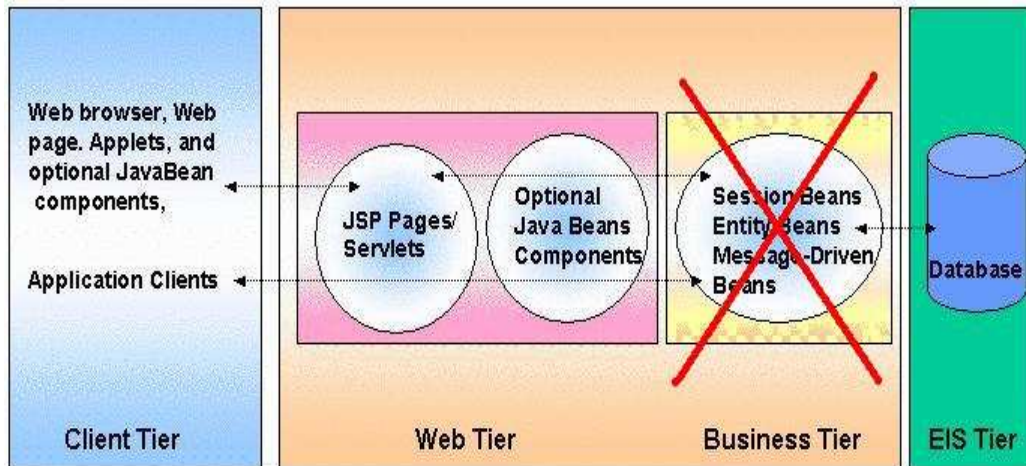


El escenario web-centric es el más ampliamente utilizado actualmente.

1.6 Nota sobre MVC

Es muy usual utilizar el patrón MVC como patrón arquitectónico. En este patrón, ya sabemos que el modelo representa los datos y las reglas de negocio que determinan el acceso y modificación de los datos. La vista traduce los contenidos del modelo a uno o unos modos de presentación. Cuando el modelo cambia, es responsabilidad de la vista mantener la consistencia de la presentación. El controlador define el comportamiento de la aplicación, es decir, asocia (map) las peticiones del usuario (captadas por botones, ítems de menú, etc.) con acciones realizadas por componentes del modelo. En una aplicación web, las peticiones aparecen como peticiones HTTP que usan normalmente el método POST o GET para invocar a la capa web. El controlador es el responsable de seleccionar la vista que debe responder a la petición realizada por el usuario.

Ya dijimos en el escenario web-centric que en bastantes ocasiones las aplicaciones no requieren acceder a múltiples sistemas empresariales, es decir, la capa de lógica de negocio no requiere fuerte conectividad distribuida con sistemas empresariales. No se contemplan EJBs, pero esto no significa que desaparezcan los componentes del modelo, sino que los servicios del modelo se implementan en *JavaBeans* (no *Enterprise JavaBeans*) para ser utilizados por *Servlets*/JSP dentro de la capa web:



En una aplicación centrada en la capa Web, sigue existiendo, aunque sea ligera, un modelo que contiene entidades y reglas de negocio; es decir, **el que no sean necesarios los EJB no implica no modularizar, mezclarlo todo y eliminar los componentes del modelo.**

Nota: La especificación JEE no considera como componentes JEE a los *Java Beans* ya que son diferentes de los EJB (no confundirlos). La arquitectura de componentes *JavaBeans* se pueden utilizar tanto en la capa de cliente como de servidor, mientras que los componentes *Enterprise JavaBeans* sólo se utilizan en la capa de negocio como parte de los servicios del servidor.

2. TOMCAT

2.1 Tomcat - Definición

Jakarta-Tomcat es un servidor de aplicaciones que extiende la funcionalidad de un servidor web. Por sí solo, un servidor web sólo puede mostrar páginas estáticas html. *Tomcat* extiende y mejora esta funcionalidad al permitir ejecutar componentes *Java* tales como JSPs, *Servlets*, etc.

Tomcat no soporta *Enterprise Java Beans* (EJBs). Es básicamente un contenedor de *Servlets* y JSPs.

Ejemplos de otros servidores de aplicaciones que soportan *Servlets* y JSPs son los siguientes:

- iPlanet
- Jetty
- JRun
- Bluestone

- Borland Enterprise Server

Se puede trabajar con *Tomcat* como contenedor de JSPs y *Servlets* y algún otro servidor de aplicaciones como contenedor de EJBs. A continuación, se listan ejemplos de servidores de aplicaciones que soportan EJBs:

- Oracle 9iAS
- JBoss (gratuito)
- JOnAS (gratuito)
- Web Logic
- IBM WebSphere

Tomcat se constituye en uno de los proyectos más interesantes de código abierto (open source) liderado por la *Apache Software Foundation* y es distribuido junto con el servidor web *Apache*.

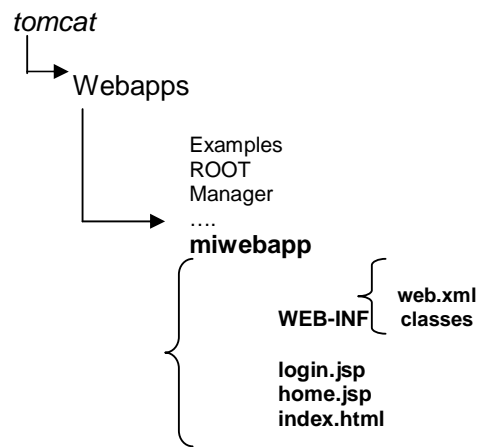
Tomcat cumple parcialmente (no soporta EJBs) con los estándares JEE e implementa las especificaciones *Servlet 2.3* y *JavaServer Pages 1.2*.

2.2 Aplicaciones web en *Tomcat*

Aplicación web:

“Una aplicación web es una colección de *Servlets*, páginas JSP, clases *Java*, archivos de descripción de la aplicación y documentos estáticos tales como HTML, XHTML, imágenes, etc.”¹

Las aplicaciones web en *Tomcat* deben ser alojadas en un contenedor. El contenedor que alberga una aplicación web es la estructura de directorios en donde están colocados todos los archivos necesarios para la ejecución de la aplicación web. Esta estructura de directorios en la que colocamos los componentes se crea dentro del directorio **Webapps**.



¹ Definición extraída de la *Api Servlet 2.2*

Principales directorios de una aplicación web:

/miwebapp. Directorio raíz de la aplicación web en donde se colocan todos los archivos HTML, JSP, GIF, JPG que utiliza la aplicación. Se pueden crear subdirectorios adicionales para mantener cualquier otro recurso de tipo estático que forme parte de la aplicación web.

/miwebapp/WEB-INF. Directorio que contiene todos los recursos relacionados con la aplicación web que no son de acceso directo para un cliente (browser). En este directorio, se coloca el archivo descriptor web.xml, donde se define la configuración de la aplicación web.

/miwebapp/WEB-INF/classes. Directorio que contiene todos los *Servlets* y cualquier otra clase de utilidad o complementaria que se necesite para la ejecución de la aplicación web. Por lo general, contiene solo archivos compilados .class

Archivo de configuración web.xml

Conocido como archivo descriptor de la aplicación web, este archivo xml, ubicado dentro del directorio WEB-INF, contiene la descripción de la configuración correspondiente a la aplicación web. La información que contiene puede incluir lo siguiente:

- Configuración de la sesión
- Definiciones de *Servlets* y Registro de *Servlets*
- Registro de tipos MIME
- Páginas de error
- Páginas de bienvenida (tag <welcome-file-list>)

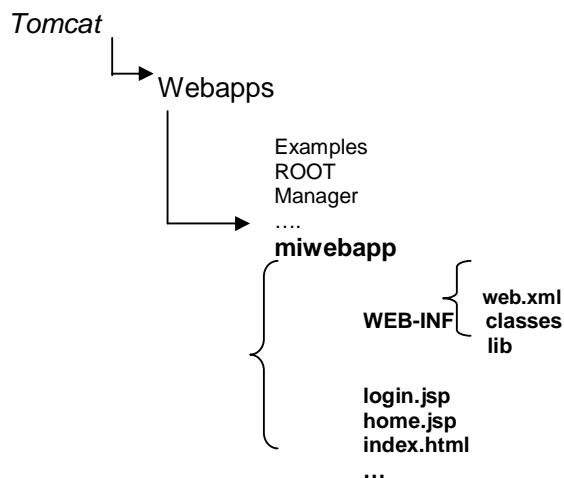
Archivos War

Un archivo WAR (web *Archive*) es la representación de una aplicación web en una unidad única distribuible. Es el método estándar empleado para empaquetar una aplicación web y dejarla lista para su distribución y acceso a través de servidores web con soporte para *Servlets* y páginas JSP.

No importa el número o tipo de recursos (*Servlets*, JSPs, HTMLs, etc.); un archivo WAR agrupa una aplicación web completa, en una única unidad de distribución, en un único archivo.

Resumen

La estructura de una aplicación WEB



La arquitectura de un servidor de aplicaciones incluye una serie de subsistemas:

- **Servidor HTTP** (también denominado servidor Web o servidor de páginas); por ejemplo, el servidor *Apache*
- **Contenedor de aplicaciones** o contenedor Servlet/JSP; por 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).

Si desea saber más acerca de estos temas, puede consultar la siguiente página.

<http://pdf.coreservlets.com/>

Aquí hallará lo referente a *Servlets*.

**UNIDAD DE
APRENDIZAJE**

1

SEMANA

2

SERVLET Y JSP

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos implementarán una aplicación web en *Java* que realiza operaciones *SELECT*, *INSERT*, *UPDATE*, *DELETE* a la base de datos que contenga *Servlets*, *JSP*, *JavaBeans* y procese los objetos *request* y *response*.

TEMARIO

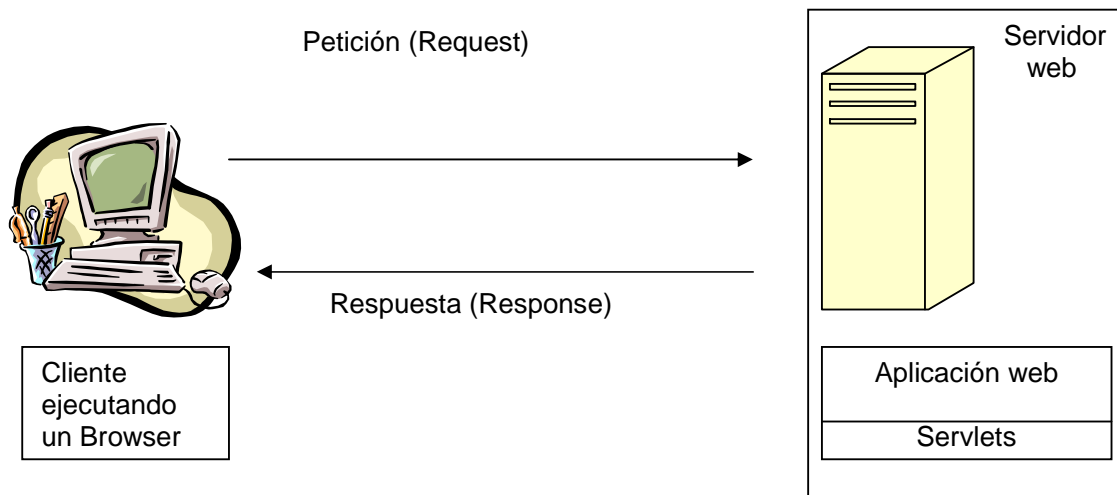
- *Servlets*
- Objeto *RequestDispatcher*

ACTIVIDADES PROPUESTAS

- Los alumnos realizarán programas utilizando la tecnología de *Servlets*.

1. **SERVLETS**

Un *servlet* es un componente JEE que extiende la capacidad de proceso de un servidor que emplea el paradigma *request – response*. Por lo tanto, “Un *servlet* es una clase *Java* que recibe requerimientos de un cliente para cumplir con un servicio; luego de cumplir con el servicio, envía la respuesta hacia el cliente.”



La API *Servlet* se constituye de dos paquetes básicos:

- `javax.servlet`
- `javax.servlet.http`

2. **Ciclo de Vida de un SERVLET**

El ciclo de vida del *servlet* está compuesto de tres fases:

- a) El método `init`. Este método es llamado por el servidor de aplicaciones cuando el *servlet* se está cargando en memoria.
- b) El método `service`. Este método es llamado por cada petición de cliente. Para las peticiones HTTP, este método se ha especializado para enviar la petición al método `doGet` o `doPost`.
- c) El método `destroy`. Este método es llamado por el servidor de aplicaciones cuando el *servlet* es descargado de memoria.

Los *Servlets* son cargados en memoria en la primera petición de un cliente o cuando el servidor de aplicaciones arranca.

Cada petición de cliente es servida sobre un diferente hilo (thread); por lo tanto, muchos clientes pueden acceder al mismo código en paralelo. Es responsabilidad del desarrollador sincronizar los accesos a los recursos compartidos.

Se pueden crear *Servlets* de un solo hilo (SingleThread) implementando la interfaz:

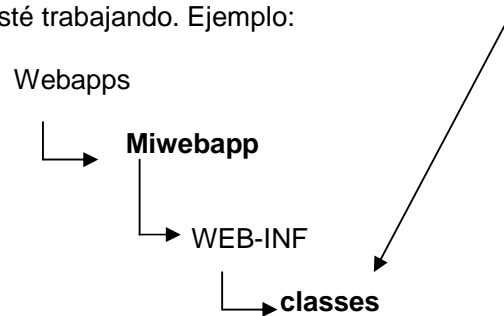
`javax.servlet.SingleThreadModel`

3. La clase HTTPSERVLET

Los clientes web (browsers) interactúan con los *Servlets* usando el protocolo HTTP (*Request-Response*); por lo tanto, para crear *Servlets* que soporten este protocolo, se debe heredar de la clase `javax.servlet.http.HttpServlet`.

La clase `HttpServlet` provee una estructura de trabajo adecuada para manipular el protocolo HTTP junto con los métodos GET y POST.

El *servlet* debe ser creado dentro de la carpeta *classes* del web application con el que se esté trabajando. Ejemplo:



4. Métodos de la clase HttpServlet

protected void	<code>doDelete</code> (<code>HttpServletRequest</code> req, <code>HttpServletResponse</code> resp) Called by the server (via the <code>service</code> method) to allow a servlet to handle a DELETE request.
protected void	<code>doGet</code> (<code>HttpServletRequest</code> req, <code>HttpServletResponse</code> resp) Called by the server (via the <code>service</code> method) to allow a servlet to handle a GET request.
protected void	<code>doHead</code> (<code>HttpServletRequest</code> req, <code>HttpServletResponse</code> resp) Receives an HTTP HEAD request from the protected <code>service</code> method and handles the request.
protected void	<code>doOptions</code> (<code>HttpServletRequest</code> req, <code>HttpServletResponse</code> resp) Called by the server (via the <code>service</code> method) to allow a servlet to handle a OPTIONS request.
protected void	<code>doPost</code> (<code>HttpServletRequest</code> req, <code>HttpServletResponse</code> resp) Called by the server (via the <code>service</code> method) to allow a servlet to handle a POST request.

protected void	<code>doPut</code> (<code>HttpServletRequest</code> req, <code>HttpServletResponse</code> resp) Called by the server (via the <code>service</code> method) to allow a servlet to handle a PUT request.
protected void	<code>doTrace</code> (<code>HttpServletRequest</code> req, <code>HttpServletResponse</code> resp) Called by the server (via the <code>service</code> method) to allow a servlet to handle a TRACE request.
protected long	<code>getLastModified</code> (<code>HttpServletRequest</code> req) Returns the time the <code>HttpServletRequest</code> object was last modified, in milliseconds since midnight January 1, 1970 GMT.
protected void	<code>service</code> (<code>HttpServletRequest</code> req, <code>HttpServletResponse</code> resp) Receives standard HTTP requests from the public <code>service</code> method and dispatches them to the <code>doXXX</code> methods defined in this class.
void	<code>service</code> (<code>ServletRequest</code> req, <code>ServletResponse</code> res) Dispatches client requests to the protected <code>service</code> method.

5. Registro del *Servlet*

Una vez compilado el *servlet*, debe registrarlo dentro del archivo descriptor `web.xml`:

```
...

<!--=====
<!-- La etiqueta <servlet> define el nombre físico de un servlet -->
<!-- =====>
<servlet>
  <servlet-name>
    SampleServlet
  </servlet-name>
  <servlet-class>
    edu.cibertec.servlet.SampleServlet
  </servlet-class>
</servlet>

<!--=====
<!-- La etiqueta <servlet-mapping> define el nombre lógico de un servlet -->
<!-- =====>

<servlet-mapping>
  <servlet-name>
    SampleServlet
  </servlet-name>
  <url-pattern>
    /servlet1
  </url-pattern>
</servlet-mapping>

...
```


Object	getAttribute (String name) <i>Returns the value of the named attribute as an <code>Object</code>, or <code>null</code> if no attribute of the given name exists.</i>
Enumeration	getAttributeNames () <i>Returns an <code>Enumeration</code> containing the names of the attributes available to this request.</i>
String	getCharacterEncoding () <i>Returns the name of the character encoding used in the body of this request.</i>
int	getContentLength () <i>Returns the length, in bytes, of the request body and made available by the input stream, or -1 if the length is not known.</i>
String	getContentType () <i>Returns the MIME type of the body of the request, or <code>null</code> if the type is not known.</i>
ServletInputStream	getInputStream () <i>Retrieves the body of the request as binary data using a <code>ServletInputStream</code>.</i>
String	getLocalAddr () <i>Returns the Internet Protocol (IP) address of the interface on which the request was received.</i>
Locale	getLocale () <i>Returns the preferred <code>Locale</code> that the client will accept content in, based on the <code>Accept-Language</code> header.</i>
Enumeration	getLocales () <i>Returns an <code>Enumeration</code> of <code>Locale</code> objects indicating, in decreasing order starting with the preferred locale, the locales that are acceptable to the client based on the <code>Accept-Language</code> header.</i>
String	getLocalName () <i>Returns the host name of the Internet Protocol (IP) interface on which the request was received.</i>
int	getLocalPort () <i>Returns the Internet Protocol (IP) port number of the interface on which the request was received.</i>
String	getParameter (String name) <i>Returns the value of a request parameter as a <code>String</code>, or <code>null</code> if the parameter does not exist.</i>
Map	getParameterMap () <i>Returns a <code>java.util.Map</code> of the parameters of this request.</i>
Enumeration	getParameterNames () <i>Returns an <code>Enumeration</code> of <code>String</code> objects containing the names of the parameters contained in this request.</i>
String[]	getParameterValues (String name) <i>Returns an array of <code>String</code> objects containing all of the</i>

	<i>values the given request parameter has, or null if the parameter does not exist.</i>
String	<code>getProtocol()</code> <i>Returns the name and version of the protocol the request uses in the form protocol/majorVersion.minorVersion, for example, HTTP/1.1.</i>
BufferedReader	<code>getReader()</code> <i>Retrieves the body of the request as character data using a <code>BufferedReader</code>.</i>
String	<code>getRealPath(String path)</code> <i>Deprecated. As of Version 2.1 of the Java Servlet API, use <code>ServletContext.getRealPath(java.lang.String)</code> instead.</i>
String	<code>getRemoteAddr()</code> <i>Returns the Internet Protocol (IP) address of the client or last proxy that sent the request.</i>
String	<code>getRemoteHost()</code> <i>Returns the fully qualified name of the client or the last proxy that sent the request.</i>
int	<code>getRemotePort()</code> <i>Returns the Internet Protocol (IP) source port of the client or last proxy that sent the request.</i>
RequestDispatcher	<code>getRequestDispatcher(String path)</code> <i>Returns a <code>RequestDispatcher</code> object that acts as a wrapper for the resource located at the given path.</i>
String	<code>getScheme()</code> <i>Returns the name of the scheme used to make this request, for example, http, https, or ftp.</i>
String	<code>getServerName()</code> <i>Returns the host name of the server to which the request was sent.</i>
int	<code>getServerPort()</code> <i>Returns the port number to which the request was sent.</i>
boolean	<code>isSecure()</code> <i>Returns a boolean indicating whether this request was made using a secure channel, such as HTTPS.</i>
void	<code>removeAttribute(String name)</code> <i>Removes an attribute from this request.</i>
void	<code>setAttribute(String name, Object o)</code> <i>Stores an attribute in this request.</i>
void	<code>setCharacterEncoding(String env)</code> <i>Overrides the name of the character encoding used in the body of this request.</i>

7. Métodos de la clase `HttpServletRequest`

String	<code>getAuthType()</code> <i>Returns the name of the authentication scheme used to protect</i>
------------------------	--

	<i>the servlet.</i>
<u>String</u>	<u>getContextPath()</u> <i>Returns the portion of the request URI that indicates the context of the request.</i>
<u>Cookie[]</u>	<u>getCookies()</u> <i>Returns an array containing all of the Cookie objects the client sent with this request.</i>
long	<u>getDateHeader(String name)</u> <i>Returns the value of the specified request header as a long value that represents a Date object.</i>
<u>String</u>	<u>getHeader(String name)</u> <i>Returns the value of the specified request header as a String.</i>
<u>Enumeration</u>	<u>getHeaderNames()</u> <i>Returns an enumeration of all the header names this request contains.</i>
<u>Enumeration</u>	<u>getHeaders(String name)</u> <i>Returns all the values of the specified request header as an Enumeration of String objects.</i>
int	<u>getIntHeader(String name)</u> <i>Returns the value of the specified request header as an int.</i>
<u>String</u>	<u>getMethod()</u> <i>Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT.</i>
<u>String</u>	<u>getPathInfo()</u> <i>Returns any extra path information associated with the URL the client sent when it made this request.</i>
<u>String</u>	<u>getPathTranslated()</u> <i>Returns any extra path information after the servlet name but before the query string, and translates it to a real path.</i>
<u>String</u>	<u>getQueryString()</u> <i>Returns the query string that is contained in the request URL after the path.</i>
<u>String</u>	<u>getRemoteUser()</u> <i>Returns the login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated.</i>
<u>String</u>	<u>getRequestedSessionId()</u> <i>Returns the session ID specified by the client.</i>
<u>String</u>	<u>getRequestURI()</u> <i>Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request.</i>
<u>StringBuffer</u>	<u>getRequestURL()</u> <i>Reconstructs the URL the client used to make the request.</i>
<u>String</u>	<u>getServletPath()</u> <i>Returns the part of this request's URL that calls the servlet.</i>
<u>HttpSession</u>	<u>getSession()</u> <i>Returns the current session associated with this request, or if the request does not have a session, creates one.</i>
<u>HttpSession</u>	<u>getSession(boolean create)</u>

	<i>Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.</i>
Principal	getUserPrincipal() <i>Returns a java.security.Principal object containing the name of the current authenticated user.</i>
boolean	isRequestedSessionIdFromCookie() <i>Checks whether the requested session ID came in as a cookie.</i>
boolean	isRequestedSessionIdFromURL() <i>Checks whether the requested session ID came in as part of the request URL.</i>
boolean	isRequestedSessionIdValid() <i>Checks whether the requested session ID is still valid.</i>
boolean	isUserInRole(String role) <i>Returns a boolean indicating whether the authenticated user is included in the specified logical "role".</i>

8. DESPACHADOR DE LA PETICIÓN (*Request Dispatcher*)

El despachador de la petición (*RequestDispatcher*) permite remitir una petición (*request*) a otro servlet o a otro componente JEE.

Un objeto *RequestDispatcher* es obtenido de la siguiente manera:

```
RequestDispatcher rd=getServletContext().getRequestDispatcher("ruta_recurso");
```

Una vez obtenido el objeto *RequestDispatcher*, invocamos al otro recurso (servlet) enviándole el objeto *request*.

```
rd.forward(request, response)
```

Hay que registrar el servlet *MyForwardServlet* para lo cual debemos modificar el archivo *web.xml*.

8. COMPARTIENDO OBJETOS

Existen dos formas de compartir objetos entre *Servlets*:

- El objeto *ServletContext* y
- El objeto *HttpServletRequest*

El *ServletContext* es usado cuando un grupo de *Servlets* necesitan trabajar con un mismo objeto. El servlet que desea compartir el objeto usa el método *setAttribute* del objeto *ServletContext* y otro servlet dentro del contexto de *Servlets* que desea tomar el objeto usa el método *getAttribute*. Ejemplo:

Servlet 1 (compartiendo objeto) :

```
MiClase elObjeto = new MiClase();
getServletContext().setAttribute("objeto",elObjeto);
```

Servlet 2 (tomando objeto):

```
MiClase elObjeto;
ElObjeto =(MiClase)getServletContext().getAttribute("objeto");
```

HttpServletRequest es usado cuando al hacer uso de una remisión (forward) se necesita compartir un objeto entre *Servlets*. El servlet que desea compartir el objeto usa el método *setAttribute* del objeto *HttpServletRequest* y el otro servlet usa el método *getAttribute*. Ejemplo:

```
MiClase elObjeto = new MiClase();
request.setAttribute("objeto",elObjeto);
```

```
ElObjeto =(MiClase)request.getAttribute("objeto");
```

Preguntas de Certificación

1. ¿Cuáles métodos en la clase `HttpServlet` sirve el pedido del HTTP POST? (Seleccione uno)

- a. `doPost(ServletRequest, ServletResponse)`
- b. `doPOST(ServletRequest, ServletResponse)`
- c. `servicePost(HttpServletRequest, HttpServletResponse)`
- d. `doPost(HttpServletRequest, HttpServletResponse)`

2. Considere el siguiente código de la página HTML:

```
<html><body>
<a href="/servlet/HelloServlet">POST</a>
</body></html>
```

¿Cuál método de la clase `HelloServlet` fue invocado cuando el enlace mostrado es pulsado? (Seleccione uno)

- a. `doGet`
- b. `doPost`
- c. `doForm`
- d. `doHref`
- e. `serviceGet`

3. Considere el siguiente código:


```
public void doGet(HttpServletRequest req,
HttpServletResponse res)
{
    HttpSession session = req.getSession();
    ServletContext ctx = this.getServletContext();
    if(req.getParameter("userid") != null)
    {
        String userid = req.getParameter("userid");
        //1
    }
}
```

Si desea usar el parámetro `userid` para que esté disponible en los pedidos del mismo usuario, ¿cuál de las siguientes líneas se deberá insertar en `//1`? (Seleccione uno)


- a. `session.setAttribute("userid", userid);`
- b. `req.setAttribute("userid", userid);`
- c. `ctx.addAttribute("userid", userid);`
- d. `session.addAttribute("userid", userid);`
- e. `this.addParameter("userid", userid);`

- f. `this.setAttribute("userid", userid);`
4. ¿Cuál de las siguientes líneas se deberá usar en la salida del DataServlet para invocar algún otro *servlet*? (Seleccione uno)
- a. `RequestDispatcher rd =
 request.getRequestDispatcher("/servlet/DataServlet");
 rd.include(request, response);`
- b. `RequestDispatcher rd =
 request.getRequestDispatcher("/servlet/DataServlet");
 rd.include(response);`
- c. `RequestDispatcher rd = request.getRequestDispatcher();
 rd.include("/servlet/DataServlet", request, response);`
- d. `RequestDispatcher rd = request.getRequestDispatcher();
 rd.include("/servlet/DataServlet", response);`
- e. `RequestDispatcher rd = request.getRequestDispatcher();
 rd.include("/servlet/DataServlet");`

Resumen

 Para registrar un *Servlet*, es necesario escribir un conjunto de *tags* en el archivo de `web.xml`.

```
<servlet>
  <servlet-name>NombreServlet</servlet-name>
  <servlet-class>rutaDeLaClase </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>NombreServlet </servlet-name>
  <url-pattern>/Alias</url-pattern>
</servlet-mapping>
```

 Si desea saber más acerca de estos temas, puede consultar la siguiente página.

 <http://courses.coreservlets.com/Course-Materials/scwcd.html>

Aquí hallará ejercicios sobre JSP.

UNIDAD DE
APRENDIZAJE

1

SEMANA

3

SERVLET Y JSP

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos implementarán una aplicación web en *Java* que realiza operaciones *SELECT*, *INSERT*, *UPDATE*, *DELETE* a la base de datos que contenga *Servlets*, *JSP*, *JavaBeans* y procese los objetos *request* y *response*.

TEMARIO

- *Java Server Pages*
- Eventos (*Listener*)

ACTIVIDADES PROPUESTAS

- Los alumnos realizarán programas utilizando la tecnología de *Servlets* y *JSPs*.

1. Java Server Pages JSP

Un JSP es un *servlet*; por lo tanto, es un componente web que se encuentra en el lado del servidor. Un JSP tiene código *Java* dentro del código en HTML (*embedded*); a diferencia de los *Servlets* que pueden generar HTML desde código *Java*.

“Los JSPs son utilizados para generar páginas web dinámicas. “

No es muy frecuente encontrar programadores *Java* buenos en diseño html o diseñadores de html buenos programando en *Java*; por ello, JSP es la solución para separar roles:

“Un jsp tiene poco código *Java* que puede ser fácilmente entendido por un diseñador html.”

Ejemplo:

```
<HTML>
<HEAD><TITLE>Saludo</TITLE><HEAD>
<BODY>
<% String miAlias=request.getParameter("miAlias"); %>
<H1>Hola <%=miAlias%></H1>
</BODY>
</HTML>
```

2. CICLO DE VIDA DEL JSP

Antes de poder ser ejecutado, un jsp debe ser convertido en un *servlet* de *Java*. Esto es hecho en dos etapas:

- El texto jsp es *traducido* en código *Java*.
- El código java es *compilado* en *servlet*.

El *servlet* resultante procesa las peticiones http. El proceso de traducido y compilado es realizado una sola vez antes de procesar la primera petición http; luego, el *servlet* resultante tiene el mismo ciclo de vida que cualquier servlet. Los equivalentes a los métodos init, service y destroy son los siguientes:

- jspInit,
- _jspService y
- jspDestroy

3. DIRECTIVAS JSP

Las directivas JSP son usadas para definir información al *traductor java* acerca de la página.

La sintaxis es la siguiente:

```
<%@ directiva[ atributo=valor ] %>
```

- **Directiva include.** Se usa para definir el contenido de otro archivo en el JSP.

Ejemplo:

```
<HTML>
<HEAD><TITLE>Mi primer loguito<TITLE></HEAD>
<BODY>
<%@ include file="loguito.html"%>
```

- **Directiva page.** Se usa para definir las características de las cuales va a depender la página. La directiva aplica a todo el JSP incluso los archivos incluidos con la directiva include.

Directiva page info. Define una cadena de texto que es ubicada en el método `Servlet.getServletInfo()` del código traducido. Ejemplo:

```
<%@ page info="En el mar, la vida es más sabrosa" %>
```

Directiva page import. Se usa para importar una lista de nombres de paquetes separados por comas. Ejemplo:

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

Directiva page errorPage. Se usa para redireccionar un cliente a un URL específico cuando ocurre una excepción que no ha sido capturada en la página. Ejemplo:

```
<%@ page errorPage="/jsps/error.jsp" %>
```

Directiva `page isErrorPage`. Se usa para indicar si la página es un “target” válido (destino) de una directiva `page errorPage`. El valor por defecto es `false`. Ejemplo:

```
<%@ page isErrorPage="true" %>
```

```
<%@ page isErrorPage="false" %>
```

4. JSP SCRIPTING

El *scripting* es utilizado para codificar el JSP. El *scripting* está conformado de:

- Declarations
- Scriptlets
- Expressions

- a) *Declarations*. Son utilizadas para declarar métodos y variables de *instancia* en el servlet JSP.

Sintaxis: `<%! Declaración %>`

Ejemplo:

```
<%! private int contador = 0; %>
```

- b) *Scriptlets*. Se utilizan para escribir código *Java* en el JSP.

Sintaxis: `<% código_java %>`

Ejemplo:

```
<%
    String sexo = request.getParameter("sexo");
    If(sexo.equals("M")){
%>
<H2>Sr.</H2>
<%
    }else{
%>
<H2>Sr.</H2>
<% } %>
```

- c) *Expressions*. Las expresiones son para incluir directamente dentro de la salida de la página cadenas (Strings), que son el resultado de evaluar una expresión de código *Java* y luego convertirla en una cadena.

Sintaxis: `<%= expression %>`

Ejemplo:

La fecha actual es `<%=new java.util.Date() %>`

5. INTERACCIÓN CON UN JSP

Existen tres formas básicas de interactuar con un JSP:

- a) Un JSP puede ser invocado por su URL. Ejemplo:

`http://nombreservidor/aplicacionweb/nombrejsp.jsp`

- b) Un JSP puede ser invocado desde un servlet usando el método *forward* del objeto *RequestDispatcher*. Ejemplo:

```
String miRuta = "/jsps/consultas.jsp";
RequestDispatcher rd =
getServletContext().getRequestDispatcher(miRuta);
Rd.forward(request,response);
```

- c) Un *Servlet* o un *JSP* pueden ser invocados desde un JSP usando la etiqueta `<FORM>` o la etiqueta `<A HREF>`. Ejemplo:

`<FORM ACTION="/aplicacionWeb/URLServlet">`

` texto `

`<FORM ACTION="/aplicacionWeb/nombrejsp.jsp">`

` texto `

6. DETECTANDO ERRORES EN EL JSP

Existen tres categorías de error: las dos primeras categorías son detectadas por el servidor web, mientras que la tercera es detectada por el cliente browser:

- a) JSP Translate. Se genera un error si se escribe mal o se falla al usar los atributos de las etiquetas de JSP.

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

- b) Servlet Compilation. Si se falla al escribir código *Java* o cuando se omite alguna directiva de página que sea obligatoria.

<%= new java.util.Date() %>

- c) HTML Presentation. Si algún elemento HTML está definido incorrectamente.

7. OBJETOS IMPLÍCITOS EN EL JSP

El código JSP puede acceder a información del servlet usando objetos implícitos definidos por cada página. Los objetos implícitos son variables predefinidas (no necesitamos declararlas) que se pueden referenciar en el código *Java* del JSP.

- Objeto *request*. Contiene la información de petición del actual HTTP request.
- Objeto *session*. Contiene la información de la sesión del cliente. Es una *instancia* de la clase `javax.servlet.http.HttpSession`.
- Objeto *out*. Es usado para las salidas de texto que se quieran incluir en la página.
- Objeto *Application*. Contiene información del contexto de todos los componentes web de la misma aplicación web. Es una *instancia* de la clase `javax.servlet.ServletContext`

8. ATRIBUTOS DEL SCOPE (ÁMBITO)

Un JSP puede acceder a objetos en tiempo de ejecución vía uno de los siguientes *scopes* o ámbitos:

- *request* (Es el objeto `HttpServletRequest` Actual)
 - *session* (Es el objeto `HttpSession` actual)
 - *application* (Es el objeto `ServletContext` actual)
- a) **Request Scope**. Este ámbito se constituye en la vía más adecuada para que un servlet pase referencias de objetos al JSP. Las referencias a objetos con ámbito Request son almacenadas en el objeto request.

Se usa lo siguiente:

- `setAttribute(String, Object)` para cargar (setear) el objeto en el request y
 - `getAttribute(String)` para recuperar el objeto
- b) **Session Scope**. Se puede acceder a este ámbito desde *Servlets* y páginas JSP que están procesando peticiones que se encuentran en la misma sesión.

Las referencias a los objetos son perdidas después que la sesión asociada es finalizada.

Las referencias a los objetos con ámbito *Session* son almacenadas en el objeto *session*.

Se usa lo siguiente:

- `setAttribute(String,Object)` para cargar(*setear*) el objeto en la sesión y
- `getAttribute(String)` para recuperar el objeto

- c) **Application Scope.** Se puede acceder a este ámbito desde *Servlets* y páginas JSP que están procesando peticiones que se encuentran en la misma aplicación web. (El mismo contexto)

Las referencias a los objetos con ámbito *Application* son almacenadas en el objeto *application*.

Se usa lo siguiente:

- `setAttribute(String,Object)` para cargar(*setear*) el objeto en el contexto y
- `getAttribute(String)` para recuperar el objeto

9. OBJETOS IMPLÍCITOS EN UN JSP

Objeto	Significado
request	El objeto <code>HttpServletRequest</code> asociado con la petición
response	El objeto <code>HttpServletResponse</code> asociado con la respuesta
out	El <code>Writer</code> empleado para enviar la salida al cliente. La salida de los JSP emplea un <i>buffer</i> que permite que se envíen cabeceras HTTP o códigos de estado aunque ya se haya empezado a escribir en la salida (<code>out</code> no es un <code>PrintWriter</code> , sino un objeto de la clase especial <code>JspWriter</code>).
session	El objeto <code>HttpSession</code> asociado con la petición actual. En JSP, las sesiones se crean automáticamente, de modo que este objeto está <i>instanciado</i> aunque no se cree explícitamente una sesión.
application	El objeto <code>ServletContext</code> , común a todos los <i>Servlets</i> de la aplicación web
config	El objeto <code>ServletConfig</code> , empleado para leer parámetros de inicialización
pageContext	permite acceder desde un único objeto a todos los demás objetos implícitos

page	Referencia al propio <i>servlet</i> generado (tiene el mismo valor que <i>this</i>). Como tal, en <i>Java</i> no tiene demasiado sentido utilizarla, pero está pensada para el caso en que se utilizara un lenguaje de programación distinto.
exception	Representa un error producido en la aplicación. Solo es accesible si la página se ha designado como página de error (mediante la directiva <code>page isErrorPage</code>)

10. EVENTOS EN APLICACIÓN WEB JEE

En aplicación web los eventos son nuevos, apartir en la especificación Servlet 2.3. Ellos le dan mayor grado de control sobre su aplicación web. En este capítulo, se va a estudiar la aplicación de dos importantes eventos:

- Inicio y apagado de una aplicación
- La creación y la invalidación de sesiones

Como sus nombres indican, la aplicación de inicio de evento se produce cuando su aplicación web se carga por primera vez y comenzó por el contenedor de Servlets; y solicitud de cierre se produce cuando la aplicación web se cierra.

El período de sesiones se produce en la creación de una nueva sesión cada vez que se crea en el servidor y de manera similar el período de sesiones se origina con la invalidación de una sesión cada vez que se anula. Para hacer uso de estas aplicaciones web y eventos para hacer algo útil, tendrá que crear y hacer uso de "clases oyentes". De aquí en adelante, vamos a implementar clases oyentes y cómo se pueden utilizar.

10.1 Clases oyentes

Estos son simples clases *Java* que implementan una de las dos siguientes interfaz:

- `javax.servlet.ServletContextListener`
- `javax.servlet.http.HttpSessionListener`

La implementación de `ServletContextListener` permite escuchar la creación o destrucción del contexto. La implementación de `HttpSessionListener` le permite escuchar la creación de una sesión. Veamos cuáles son los diferentes métodos de esta interfaz, que se tendrán que aplicar.

10.2 ServletContextListener:

Esta interfaz contiene dos métodos:

- `public void contextInitialized (ServletContextEvent SCE);` //Creación de un Contexto

- `public void contextDestroyed (ServletContextEvent SCE);` //Destrucción de un contexto

Un ejemplo de esa clase es el siguiente:

```
javax.servlet.ServletContextListener de importación;
javax.servlet.ServletContextEvent de importación;

clase pública ApplicationWatch implementa ServletContextListener (

    public static applicationInitialized largo = 0L;

    /* Aplicación de inicio del evento */
    public void contextInitialized (ServletContextEvent ce) (
        applicationInitialized = System.currentTimeMillis ();
    )

    /* Aplicación del evento de apagado */
    public void contextDestroyed (ServletContextEvent ce) ()

)
```

En el código anterior, una clase *Java ApplicationWatch* implementa *ServletContextListener*. Al implementar sus dos métodos realmente sólo usa uno de ellos y el segundo método sigue teniendo el cuerpo vacío. En esta categoría, se observa en el momento de la solicitud de inicio en `public static` que puede ser llamado desde otra aplicación para saber qué clase fue la última vez que esta solicitud se inició.

Se explicará cómo decirle al servidor de aplicaciones que tienen esta clase de oyente; además, se quiere que se les diga la aplicación de estos acontecimientos en un momento, pero primero vamos a ver cuáles son los diferentes métodos de *HttpSessionListener* interfaz.

10.3 HttpSessionListener

Esta interfaz te permite poder escuchar al crear o destruir una sesión.

- `public void sessionCreated (HttpSessionEvent se);`
- `public void sessionDestroyed (HttpSessionEvent se);`

Al igual que lo que hicimos en el caso de *ApplicationWatch* anterior, tendrá que crear una clase *Java* y aplicar *HttpSessionListener* interfaz. Un ejemplo de esa clase es el siguiente:

```
/*
    Expediente: SessionCounter.java
*/
```

```
javax.servlet.http.HttpSessionListener de importación;
```

```
javax.servlet.http.HttpSessionEvent de importación;
```

```

clase pública SessionCounter implementa HttpSessionListener (

    private static int activeSessions = 0;

    /* Creación de eventos de sesión */
    public void sessionCreated (HttpSessionEvent se) (
        activeSessions + +;
    )

    /* Sesión de invalidación del evento */
    public void sessionDestroyed (HttpSessionEvent se) (
        if (activeSessions > 0)
            activeSessions -;
    )

    public static int getActiveSessions () (
        activeSessions retorno;
    )
)

```

En el código anterior, SessionCounter clase implementa HttpSessionListener para contar el número de sesiones activas.

Se ha aprendido cuáles son los eventos de aplicaciones web, interfaz de lo que está disponible para nosotros y también se han visto ejemplos de la aplicación de la interfaz de las clases. Veamos cómo decirle al servidor de aplicaciones acerca de estas clases oyentes.

10.4 Registro de las clases oyentes en el web.xml

Eso es lo que hacemos poniendo classpath de estas clases en / WEB-INF/web.xml archivo con etiquetas especiales <listener>. Un ejemplo de este tipo de archivo web.xml es el siguiente:

```

<!-- Web.xml -->
<? xml version = "1.0" encoding = "ISO-8859-1">

<! DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD de aplicación Web 2.3//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.3.dtd">

<web-app>

    <!-- Listeners -->
    <listener>
        <listener-class>
            com.stardeveloper.web.listener.SessionCounter

```

```

        </ oyente de clase>
    </ oyente>
    <listener>
        <listener-class>
            com.stardeveloper.web.listener.ApplicationWatch
        </ oyente de clase>
    </ oyente>

</ web-app>

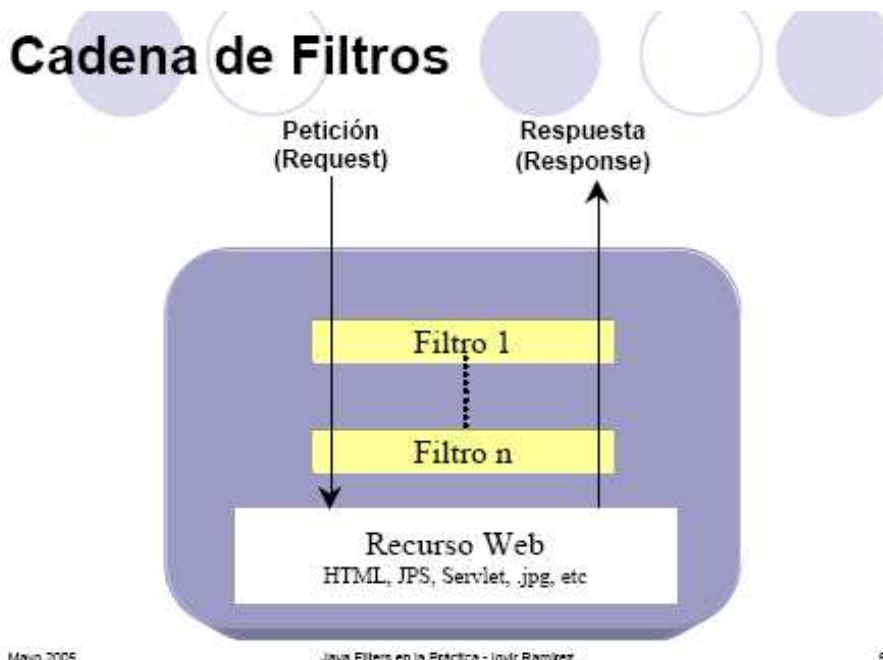
```

Como se indica anteriormente, es muy fácil declarar oyentes en el archivo web.xml. Ahora, cada vez que el servidor se inicia o cierre, una sesión se crea o destruye.

11. FILTROS EN APLICACIÓN WEB JEE

Los filtros son componentes que pueden utilizarse para analizar y/o transformar tanto los datos solicitados como los enviados en una petición web. Pueden trabajar en conjunto con páginas jsp o servlets. Son parte de la especificación de Servlets lo que los hace portables entre los diferentes contenedores disponibles en el mercado. Pueden trabajar “encadenados”.

Un filtro realiza su trabajo y pasa el control al filtro siguiente.



11.1 Ventajas

- Son componentes reutilizables.
- Son parte del estándar.

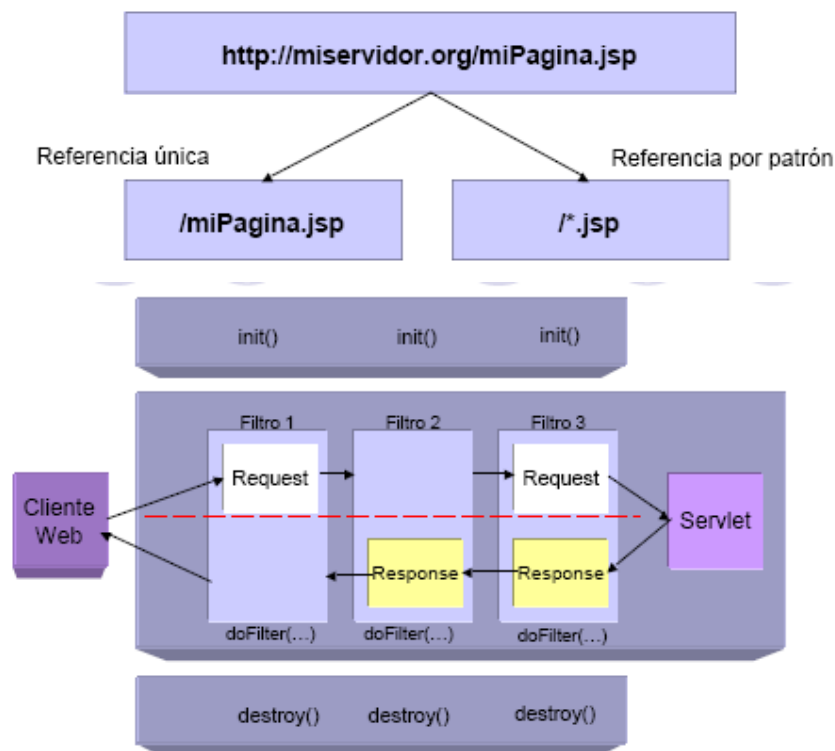
- Son fáciles de implementar.
- Se pueden incorporar y retirar de forma sencilla.
- Pueden ofrecer gran variedad de servicios.

11.2 Usos comunes

- Control de acceso a la aplicación
- Compresión de datos y Cache
- Transformaciones XML / HTML
- Procesamiento de imágenes
- Auditoría/registro de actividades
- Virtualización de recursos
- Cifrado de datos
- Control de acceso a la aplicación
- Compresión de datos y Cache
- Transformaciones XML / HTML
- Procesamiento de imágenes
- Auditoría/registro de actividades
- Virtualización de recursos

11.2 Funcionamiento

- Cada filtro responde a un Servlet o un URL que represente uno o más recursos



Mayo 2005

Java Filters en la Práctica - Iván Ramírez

10

La interfaz `javax.servlet.FilterConfig`

- Define cuatro métodos:
 - `getFilterName()` : Retorna el nombre asociado al filtro en la configuración en un `String`
 - `getInitParameter(String)`: Retorna el valor de un parámetro de configuración
 - `getInitParameterNames()`: Retorna los nombres de los parámetros de configuración.
 - `getServletContext()` : Retorna la referencia al `ServletContext` en el que trabajará el filtro
- Define un único método:
 - `doFilter()` throws `ServletException`
 - El método `doFilter` recibe como argumentos `ServletRequest` y `ServletResponse`.
- Es invocado por el contenedor cuando el filtro forma parte de la cadena creada ante la petición/respuesta de un recurso web.

11.3 Ejemplo de Filtro

- Crear la clase `MiFiltro`

```
public class MiFiltro implements Filter{
    private FilterConfig filterConfig = null;

    public void init(FilterConfig filterConfig)
        throws ServletException {
        this.filterConfig = filterConfig;
    }
    public void doFilter(ServletRequest request,
        ServletResponse response,
        FilterChain chain)
        throws ServletException {
        chain.doFilter(request,response);
    }
    public void destroy(){
        this.filterConfig =null;
    }
}
```

- Definir el filtro y sus parámetros

```
<filter>
<filter-name>Filtro Aniversario Java</filter-name>
<filter-class>MiFiltro</filter-class>
<init-param>
<param-name>aniversarioNumero</param-name>
<param-value>10</param-value>
</init-param>
</filter>
```

- Definir el patrón de recursos para el filtro

```
<filter-mapping>
```

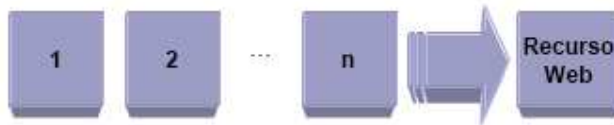
```
<filter-name>Filtro Aniversario Java</filter-name>
<url-pattern>/miPagina.jsp</url-pattern>
</filter-mapping>
Solo miPagina.jsp
<filter-mapping>
<filter-name>Filtro Aniversario Java</filter-name>
<url-pattern>/*.jsp</url-pattern>
</filter-mapping>
```

- Mapping a un *Servlet*

```
<filter-mapping>
<filter-name>Filtro Aniversario Java</filter-name>
<servlet-name>MiServlet</servlet-name>
</filter-mapping>
```

11.3 Cadena de Filtros

- Los filtros se incorporan a la cadena en el mismo orden en que aparecen en web.xml
- El último filtro de la cadena hará entrega de la solicitud al recurso web.



- La regla para seleccionar los miembros de la cadena es el siguiente:
 - Primero obtener los filtros cuyo patrón URL coincida con el recurso solicitado,
 - Luego los que el nombre Servlet coincida con el Servlet solicitado.
 - Una misma implementación, puede ser utilizada para distintos recursos empleado nombres diferentes e incluso parámetros diferentes

```
<filter>
  <filter-name>FiltroJSP</filter-name>
  <filter-class>MiFiltro</filter-class>
  <init-param>
    <param-name>param</param-name>
    <param-value>10</param-value>
  </init-param>
</filter>

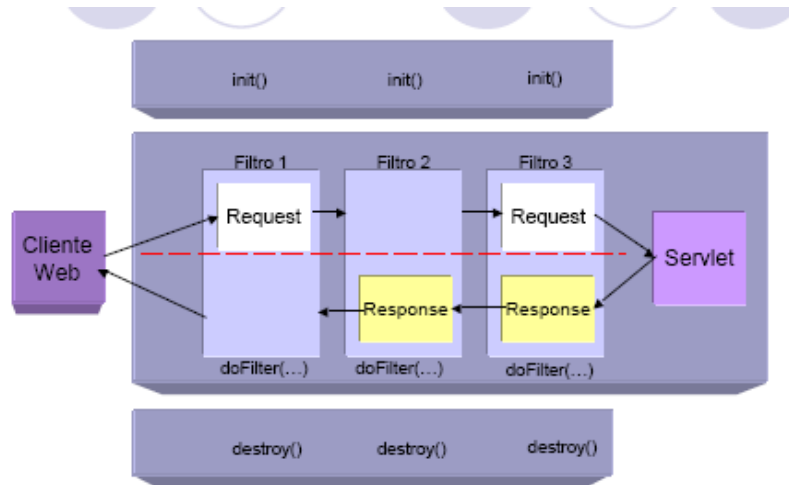
<filter-mapping>
  <filter-name>FiltroJSP</filter-name>
  <url-pattern>/*.jsp</url-pattern>
</filter-mapping>
```

```
<filter>
  <filter-name>FiltroHTM</filter-name>
  <filter-class>MiFiltro</filter-class>
  <init-param>
    <param-name>param</param-name>
    <param-value>20</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>FiltroHTM</filter-name>
  <url-pattern>/*.htm</url-pattern>
</filter-mapping>
```

- Un elemento a considerar es la longitud de la cadena para un recurso o grupo de recursos específico.
- Debe evitarse el uso excesivo de filtros ya que impactarían en el tiempo de respuesta
- Uno o dos filtros suelen ser suficientes en la mayoría de los casos
- Al utilizar la misma clase en diferentes filtros:
 - Cada definición debe tener un nombre diferente en <filter-name>
 - El contenedor creará una instancia por cada definición <filter> en el web.xml
 - Las llamadas concurrentes activarán múltiples Threads para la instancia del filtro asociado
 - La ejecución de un filtro se divide en dos momentos: antes y después de invocar la cadena

```
public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain)
    throws ServletException {
    //Código previo a la cadena
    //Es el mejor lugar para modificar el request
    chain.doFilter(request,response); //Entrega el control al próximo de la cadena
    //Código posterior a la cadena
    //Este es el mejor lugar para modificar la respuesta.
}
```



Mayo 2005

Java Filters en la Práctica - Iván Ramírez

23

11.4 Ejemplo: Control de Credenciales

- Filtro de Control de Credenciales (FCC)
- **Problema:** Un grupo específico de usuarios de una aplicación web requiere obtener sus credenciales desde dos repositorios diferentes; el resto de los usuarios solo requiere uno.
- **Propósito:** Asegurar la obtención de ambas credenciales sólo en caso de que el usuario pertenezca al grupo que lo requiere.

¿Por qué un filtro?

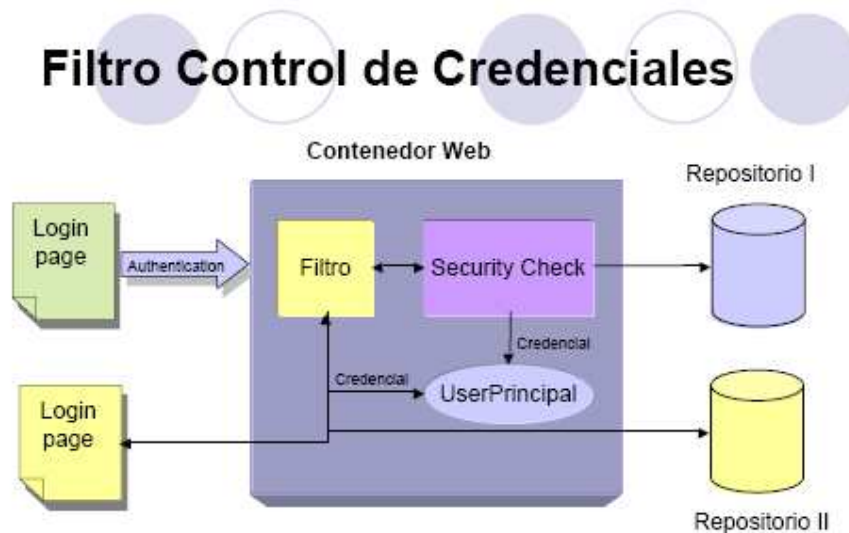
- Premisas de trabajo:
- Debe ser desarrollado en *Java*
- La seguridad debe ser controlada por el contenedor.
- La solución debe ser fácil de implantar y retirar sin afectar el funcionamiento de la aplicación.
- La solución debe ser reutilizable.

Seguridad J2EE



Acciones realizadas por el FCC

- Verifica si el usuario que intenta iniciar sesión pertenece al grupo de control.
- Si pertenece al grupo, desvía la petición a una segunda página de login. Si no, continúa la cadena.
- Intenta obtener las credenciales contra el segundo repositorio.
- Si tiene éxito, la credencial se almacena para uso futuro en la aplicación. Si falla, lo envía al inicio.
- Pasa el control a la cadena de filtros.



Mapping Control de Credenciales

```
<filter>
<filter-name>CtrlCredenciales</filter-name>
<filter-class>FilterClassName</filter-class>
</filter>
<filter-mapping>
<filter-name>CtrlCredenciales</filter-name>
<url-pattern>/j_security_check</url-pattern>
</filter-mapping>
```

Preguntas de Certificación

1 Considere el siguiente código y seleccione la sentencia correcta acerca del siguiente supuesto. (Seleccione uno.)

```
<html><body>
<%! int aNum=5 %>
The value of aNum is <%= aNum %>
</body></html>
```

- a It will print "The value of aNum is 5" to the output.
- b It will flag a compile-time error because of an incorrect declaration.
- c It will throw a runtime exception while executing the expression.
- d It will not flag any compile time or runtime errors and will not print anything to the output.

2. ¿Cuáles de los siguientes *tags* se pueden usar para imprimir un valor? (Seleccione dos.)

- a <%@ %>
- b <%! %>
- c <% %>
- d <%= %>
- e <%-- --%>


3. ¿Cuál de los siguientes métodos es definido por la *JSP engine*? (Seleccione uno.)

- a jspInit()
- b _jspService()
- c _jspService(ServletRequest, ServletResponse)
- d _jspService(HttpServletRequest, HttpServletResponse)
- e jspDestroy()


4. ¿Cuál de las siguientes *exceptions* puede ser disparado por _jspService() métodos? (Seleccione uno.)

- a javax.servlet.ServletException
- b javax.servlet.jsp.JSPException
- c javax.servlet.ServletException and javax.servlet.jsp.JSPException
- d javax.servlet.ServletException and java.io.IOException
- e javax.servlet.jsp.JSPException and java.io.IOException

Resumen

 Existe tres ámbitos:

- **request** (es el objeto *HttpServletRequest* actual)
- **session** (es el objeto *HttpSession* actual)
- **application** (es el objeto *ServletContext* actual)

 Si desea saber más acerca de estos temas, puede consultar la siguiente página.

 <http://courses.coreservlets.com/Course-Materials/scwcd.html>

Aquí hallará ejercicios sobre JSPs.

**UNIDAD DE
APRENDIZAJE**

1

SEMANA

4

SERVLET Y JSP

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos implementarán una aplicación web en *Java* que realiza operaciones *SELECT, INSERT, UPDATE, DELETE* a la base de datos que contenga *Servlets*, *JSP*, *JavaBeans* y procese los objetos *request* y *response*.

TEMARIO

- *HttpSession*

ACTIVIDADES PROPUESTAS

- Los alumnos realizarán programas utilizando sesiones.

1. HTTPSESSION

En una transacción http, cada conexión entre un cliente y un servidor es muy breve. Esta es una característica típica de cualquier protocolo sin estado (stateless). El protocolo http no mantiene el estado, es decir, no tiene un mecanismo para saber que una serie de peticiones (*request*) provienen de un mismo cliente. Ante esta limitación, surge la sesión como el mecanismo adecuado para identificar un usuario que está interactuando con el sitio web.

La sesión puede ser implementada a través de diferentes opciones:

- HttpSession
- Campos ocultos HTML(hidden fields)
- Cookies
- URL Rewriting

La interfaz HttpSession es parte de la API de *Servlets* y proporciona una alternativa para manejar el estado de una aplicación en un servidor web.

2. OBTENIENDO UNA SESIÓN

Las sesiones son compartidas por todos los *Servlets* y jsps accesados por un cliente; por lo tanto, es común que se cree un objeto Sesión cuando nos “logueamos” a un sitio web. Por otro lado, si nos “deslogueamos” de un sitio web, la sesión debería ser destruida.

En *Java*, el objeto HttpSession identifica una sesión y es obtenida utilizando el método getSession del objeto request. Ejemplo:

```
...  
  
HttpSession miSesion = req.getSession(true);  
  
...
```

Sintaxis:

request.getSession() o request.getSession(**true**). Retornan la sesión actual si existe; de lo contrario, crean una nueva.

request.getSession(**false**). Retorna la sesión actual si existe; de lo contrario, retorna un objeto nulo.

3. CONTENIDO DE LA SESIÓN

El objeto HttpSession contiene información acerca de la sesión. Para acceder a esta información, existen una serie de métodos **getters** de los cuales mostraremos los más utilizados:

- session.**getLastAccessedTime**(). La última vez que el cliente envió una petición asociada con la sesión
- session.**getCreationTime**(). Cuando la sesión fue creada
- session.**getMaxInactiveInterval**(). El intervalo de tiempo máximo (en segundos) que el Servlet Container mantiene abierta la sesión entre accesos del cliente
- session.**getAttribute**(). Obtiene un objeto que fue almacenado en la sesión

4. ALMACENANDO Y RECUPERANDO OBJETOS DE LA SESIÓN

La utilidad de una sesión radica en su capacidad de actuar como un repositorio donde se almacenan datos de la aplicación.

Los datos son almacenados como un par <"**nombre**", **valor**> siendo valor cualquier objeto que se desee almacenar en la sesión.

Los métodos para acceder a los datos son los siguientes:

session.**setAttribute**(String, Object). Sirve para almacenar objetos en la sesión.

session.**getAttribute**(String). Sirve para recuperar objetos de la sesión.

Ejemplo:

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

String nombre=" Cesar Vallejo ";

session.setAttribute("miNombre",nombre);
```

La sesión puede ser finalizada de diferentes formas:

- Programáticamente
- Por tiempo de expiración
- Cuando el usuario cierra el browser

Programáticamente:

- a) Usando el método *invalidate()* del objeto session
- b) Removiendo todos los objetos de la sesión con el método *removeAttribute()*

Sintaxis :

```
removeAttribute(String nombreAtributo)
```

nombreAtributo es el nombre del atributo a ser removido.

```
...  
    HttpSession session = req.getSession(true);  
  
    session.removeAttribute("nombre");  
    session.invalidate();  
...
```

Por tiempo de expiración:

La sesión es invalidada cuando se cumple un tiempo de inactividad de la sesión. Podemos definir el tiempo de inactividad máximo de dos formas:

- a) utilizando el método *setMaxInactiveInterval(int)*

Ejemplo:

```
session.setMaxInactiveInterval(int)
```

- b) Utilizando la etiqueta `<session-config>` dentro del archivo web.xml :

```
...  
    <session-config>  
        <session-timeout>30</session-timeout>  
    </session-config>  
...
```


5 Cookies

Las sesiones vistas anteriormente basan su funcionamiento en los *cookies*. Cuando se hace uso de la interfaz HttpSession de forma interna y totalmente transparente al programador se está haciendo uso de los *cookies*. De hecho, cuando a través de una página JSP se comienza una sesión, se crea un *cookie* llamado JSSESSIONID. La diferencia es que este *cookie* es temporal y durará el tiempo que permanezca el navegador ejecutándose, siendo borrada cuando el usuario cierre el navegador.

El objetivo de utilizar *cookies* es poder reconocer al usuario en el momento en el que se conecta al servidor. Una de las páginas que recoge la petición del usuario puede comprobar si existe un *cookie* que ha dejado anteriormente; si es así, sabe que ese usuario ya ha visitado ese website y, por lo tanto, puede leer valores que le identifiquen. Otro de los usos de los *cookies* es ofrecer una personalización al usuario. En muchos sitios web es posible elegir el color de fondo, el tipo de letra utilizado, etc. Estos valores pueden ser almacenados en *cookies* de forma que cuando acceda de nuevo a la web y se compruebe la existencia de esos valores, serán recuperados para utilizarlos en la personalización de la página tal y cómo el usuario estableció en su momento.

Un ejemplo que se ha podido encontrar en muchas webs es en el momento de realizar un registro o solicitar el alta en un área restringida, ya que en muchas ocasiones existe un checkbox que cuando se selecciona permite recordar el nombre de usuario a falta de que sólo se escriba la clave.

Utilizando también el identificador idSession que se genera en una sesión como ya hemos visto y guardándolo en el *cookie*, se pueden mostrar mensajes personalizados en el momento en el que el usuario acceda de nuevo al website. Para trabajar con *cookies*, se utiliza la clase Cookie que está disponible en paquete javax.servlet.http. Por medio de esta clase, se pueden crear *cookies*, establecer sus valores y nombres, alguna de sus propiedades, eliminarlas, leer valores que almacenan, etc.

5.1. 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 archivos 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");
%>
```

También, es posible crear *cookies* con contenido que se genere de forma dinámica. El siguiente código muestra un *cookie* que guarda un texto que está concatenado a la fecha/hora en ese momento:

```
<%@page contentType="text/html; charset=iso-8859-1"
    session="true" language="java" import="java.util.*" %>

<%
    Cookie miCookie=null;
    Date fecha=new Date();
    String texto= "Este es el texto que vamos a guardar en el
    cookie"+fecha;
    miCookie=new Cookie("nombre",texto);
%>
```

En esta ocasión, el contenido del valor a guardar en el *cookie* está en la variable "texto". También, se pueden guardar valores o datos que provengan de páginas anteriores y que hayan sido introducidas a través de un formulario:

```
<%
    Cookie miCookie=null;
    String ciudad= request.getParameter("formCiudad");
    miCookie=new Cookie("ciudadFavorita",ciudad);
%>
```

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* "unCookie":

```
<%
unCookie.setMaxAge( 60*60*24*31 );
%>
```

Si se utiliza un valor positivo, el *cookie* será destruido después de haber pasado ese tiempo; si el valor es negativo, el *cookie* no será almacenado y se borrará cuando el usuario cierre el navegador. Por último, si el valor que se establece como tiempo es cero, el *cookie* será borrado. Otro de los atributos que se incluye cuando se crea un *cookie* es el path desde el que será visto, es decir, si el valor del path es "/" (raíz), quiere decir que en todo el site se podrá utilizar ese *cookie*, pero si el valor es "/datos", quiere decir que el valor del *cookie* sólo será visible dentro del directorio "datos". Este atributo se establece mediante el método `setPath()`.

```
<%
    unCookie.setPath( "/" );
```

```
%>
```

Para conocer el valor de path, se puede utilizar el método `getPath()`.

```
<%
    out.println("cookie visible en: "+unCookie.getPath());
%>
```

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()`, Lo anterior se aplica 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; 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, se hace que el *cookie* definido en el objeto "unCookie" esté disponible para todos los dominios que contengan el nombre ".paginasjsp.com". Un nombre de dominio debe comenzar por un punto.

```
<%
    unCookie.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 archivo *cookie* real, se utiliza el método `addCookie()` de la interfaz `HttpServletResponse`:

```
<%
    response.addCookie(unCookie);
%>
```

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, el *cookie* no habrá sido grabado en el disco y, por lo tanto, cualquier aplicación o página que espere encontrar dicho *cookie* no lo encontrará.

5.2. Recuperar un *cookie*

El proceso de recuperar un *cookie* determinado puede parecer algo complejo, ya que no hay una forma de poder acceder a un *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 un
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>");
%>
```

5.3. Utilizar los *cookies*

Para realizar un ejemplo práctico, se va a seguir con el 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("cookieUsu")) {
        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%>
<p>
<%
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>
```

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 guardarlo 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>
<%
}
%>
```

Se señala erróneamente que los *cookies* están envueltos con la intromisión de la privacidad de los usuarios por parte de determinados websites e incluso con la seguridad. Estos son pequeños archivos de texto que poco daño pueden causar, y que como es lógico no pueden contener ningún código ejecutable. Desde el punto de vista es cierto que determinados sites pueden saber qué clase de palabras se consultan en un buscador o cuál 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.

Preguntas de Certificación

1 ¿Cuál de las siguientes interfaz o clases son usadas para recibir una sesión asociado con un usuario? (Seleccione uno.)

- a GenericServlet
- b ServletConfig
- c ServletContext
- d HttpServlet
- e HttpServletRequest
- f HttpServletResponse

2. ¿Cuál de los siguientes bloques de códigos es insertado que permitirá contar correctamente la cantidad de pedidos que ha hecho un usuario? (Seleccione uno.)

- a HttpSession session = request.getSession();
int count = session.getAttribute("count");
session.setAttribute("count", count++);
- b HttpSession session = request.getSession();
int count = (int) session.getAttribute("count");
session.setAttribute("count", count++);
- c HttpSession session = request.getSession();
int count = ((Integer) session.getAttribute("count")).intValue();
session.setAttribute("count", count++);
- d HttpSession session = request.getSession();
int count = ((Integer) session.getAttribute("count")).intValue();
session.setAttribute("count", new Integer(count++));

3. ¿Cuál de los siguientes métodos se invoca cuando un atributo de una sesión implementa HttpSessionBindingListener al destruir la sesión? (Seleccione uno)

- a sessionDestroyed
- b valueUnbound
- c attributeRemoved
- d sessionInvalidated

4 ¿Cuál de los siguientes métodos es invocado dentro de un atributo de una sesión que implementa apropiadamente las interfaz al destruir la sesión? (Seleccione uno.)

- a sessionDestroyed of HttpSessionListener
- b attributeRemoved of HttpSessionAttributeListener
- c valueUnbound of HttpSessionBindingListener
- d sessionWillPassivate of HttpSessionActivationListener

Resumen

📖 El tiempo que durará la sesión de inactividad en el *Tomcat* se registra en el *web.xml* en minutos.

```
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
```

📖 Método para destruir una sesión programáticamente

```
session.invalidate();
```

📖 Si desea saber más acerca de estos temas, puede consultar la siguiente página.

🖱 <http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpSession.html>

Aquí hallará descripción de la la clase *HttpSession*.



SERVLET Y JSP

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos implementarán una aplicación web en *Java* que realiza operaciones *SELECT*, *INSERT*, *UPDATE*, *DELETE* a la base de datos que contenga *Servlets*, *JSP*, *JavaBeans* y procese los objetos *request* y *response*.

TEMARIO

- Patrones de Diseño de *Software*

ACTIVIDADES PROPUESTAS

- Los alumnos realizarán un carrito de compras en web.

1. Patrones de Diseño de *Software*

El diseño es un **modelo** del sistema, realizado con una serie de principios y técnicas, que permite describir el sistema **con el suficiente detalle como para ser implementado**. Sin embargo, los principios y reglas no son suficientes. En el contexto de diseño, podemos observar que los buenos ingenieros tienen **esquemas y estructuras** de solución que usan numerosas veces en función del contexto del problema. Este es el sentido cabal de la expresión "tener una mente bien amueblada", y no el significado de tener una notable inteligencia. Estos esquemas y estructuras son conceptos reusables y nos permiten no reinventar la rueda. Un buen ingeniero reutiliza un esquema de solución ante problemas similares.

1.1 Historia

El concepto de "patrón de diseño" que tenemos en Ingeniería del *Software* se ha tomado prestado de la arquitectura. En 1977, se publica el libro "A Pattern Language: Towns/Building/Construction", de Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King y Shlomo Angel, Oxford University Press. Contiene numerosos patrones con una notación específica de Alexander.

Alexander comenta que "cada patrón describe **un problema que ocurre una y otra vez** en nuestro entorno, para describir después **el núcleo de la solución** a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo siquiera dos veces de la misma forma". El patrón es un esquema de solución que se aplica a un tipo de problema. Esta aplicación del patrón no es mecánica, sino que requiere de **adaptación y matices**. Por ello, dice Alexander que los numerosos usos de un patrón no se repiten dos veces de la misma forma.

La idea de patrones de diseño estaba "en el aire", la prueba es que numerosos diseñadores se dirigieron a aplicar las ideas de Alexander a su contexto. El catálogo más famoso de patrones se encuentra en "**Design Patterns: Elements of Reusable Object-Oriented Software**", de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, 1995, Addison-Wesley, también conocido como el libro GOF (Gang-Of-Four).

Siguiendo el libro de GOF, los patrones se clasifican según el propósito para el que han sido definidos:

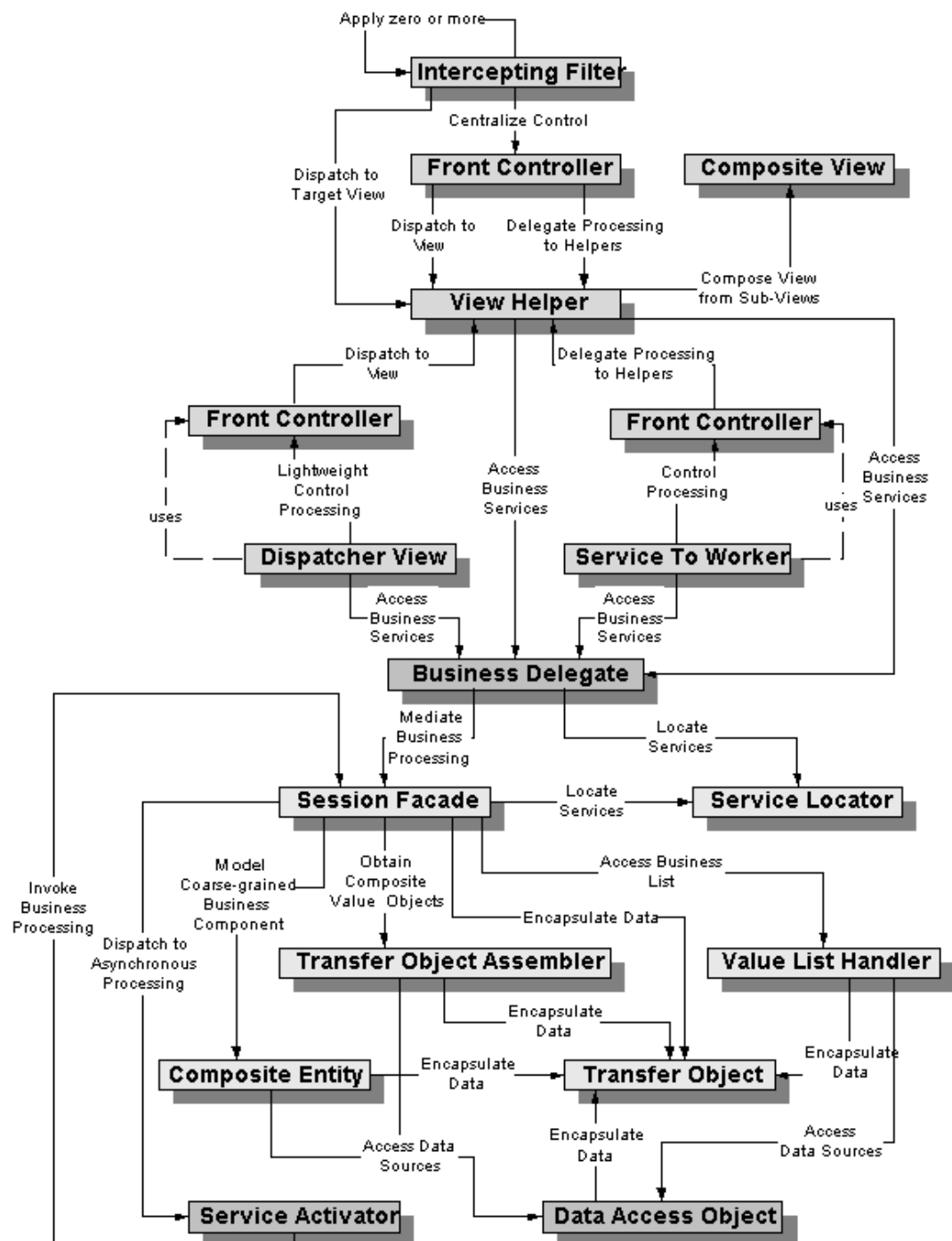
- Creacionales: solucionan problemas de creación de *instancias*. Nos ayudan a encapsular y abstraer dicha creación.
- Estructurales: solucionan problemas de composición (agregación) de clases y objetos.
- De Comportamiento: soluciones respecto a la interacción y responsabilidades entre clases y objetos, así como los algoritmos que encapsulan.

Según el ámbito, se clasifican en patrones de clase y de objeto:

		Propósito		
		Creación	Estructural	Comportamiento
Ámbito	Clase	Factory Method	Adanpter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adanpter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

	Creación	Estructural	De Conducta
Clase	Método de Fabricación	Adaptador (clases)	Interprete Plantilla
Objeto	Fábrica, Constructor, Prototipo, Singleton	Adaptador (objetos), Puente, Composición, Decorador, Fachada, Flyweight	Cadena de Responsabilidad, Comando (orden), Iterador, Intermediario, Observador, Estado, Estrategia, Visitante, Memoria

1.2 Catálogo de patrones J2EE



Capa de Presentación

<i>Decorating Filter / Intercepting Filter</i>	Un objeto que está entre el cliente y los componentes web. Este procesa las peticiones y las respuestas.
<i>Front Controller/ Front Component</i>	Un objeto que acepta todos los requerimientos de un cliente y los direcciona a manejadores apropiados. El patrón <i>Front Controller</i> podría dividir la funcionalidad en 2 diferentes objetos: el <i>Front Controller</i> y el <i>Dispatcher</i> . En ese caso, El <i>Front Controller</i> acepta todos los requerimientos de un cliente y realiza la autenticación, y el <i>Dispatcher</i> direcciona los requerimientos a manejadores apropiada.
<i>View Helper</i>	Un objeto <i>helper</i> que encapsula la lógica de acceso a datos en beneficio de los componentes de la presentación. Por ejemplo, los <i>JavaBeans</i> pueden ser usados como patrón <i>View Helper</i> para las páginas JSP.
<i>Composite view</i>	Un objeto vista que está compuesto de otros objetos vista. Por ejemplo, una página JSP que incluye otras páginas JSP y HTML usando la directiva <i>include</i> o el <i>action include</i> es un patrón <i>Composite View</i> .
<i>Service To Worker</i>	Es como el patrón de diseño MVC con el Controlador actuando como <i>Front Controller</i> pero con una cosa importante: aquí el <i>Dispatcher</i> (el cual es parte del <i>Front Controller</i>) usa <i>View Helpers</i> a gran escala y ayuda en el manejo de la vista.
<i>Dispatcher View</i>	Es como el patrón de diseño MVC con el controlador actuando como <i>Front Controller</i> pero con un asunto importante: aquí el <i>Dispatcher</i> (el cual es parte del <i>Front Controller</i>) no usa <i>View Helpers</i> y realiza muy poco trabajo en el manejo de la vista. El manejo de la vista es manejado por los mismos componentes de la Vista.

Capa de Negocios

<i>Business Delegate</i>	Un objeto que reside en la capa de presentación y en beneficio de los otros componentes de la capa de presentación llama a métodos remotos en los objetos de la capa de negocios.
<i>Value Object/ Data Transfer Object/</i>	Un objeto serializable para la transferencia de

<i>Replicate Object</i>	datos sobre la red.
<i>Session Façade/ Session Entity Façade/ Distributed Façade</i>	El uso de un <i>bean</i> de sesión como una fachada (facade) para encapsular la complejidad de las interacciones entre los objetos de negocio y participantes en un flujo de trabajo. El <i>Session Façade</i> maneja los objetos de negocio y proporciona un servicio de acceso uniforme a los clientes.
<i>Aggregate Entity</i>	Un <i>bean</i> entidad que es construido o es agregado a otros <i>beans</i> de entidad
<i>Value Object Assembler</i>	Un objeto que reside en la capa de negocios y crea <i>Value Objects</i> cuando es requerido
<i>Value List Handler/ Page-by-Page Iterator/ Paged List</i>	Es un objeto que maneja la ejecución de consultas SQL, caché y procesamiento del resultado. Usualmente implementado como <i>beans</i> de sesión
<i>Service Locator</i>	Consiste en utilizar un objeto Service Locator para abstraer toda la utilización JNDI y para ocultar las complejidades de la creación del contexto inicial, de búsqueda de objetos home EJB y recreación de objetos EJB. Varios clientes pueden reutilizar el objeto <i>Service Locator</i> para reducir la complejidad del código, proporcionando un punto de control.

Capa de Integración

<i>Data Access Object Service Activator</i>	Consiste en utilizar un objeto de acceso a datos para abstraer y encapsular todos los accesos a la fuente de datos. El DAO maneja la conexión con la fuente de datos para obtener y almacenar datos.
<i>Service Activator</i>	Se utiliza para recibir peticiones y mensajes asíncronos de los clientes. Cuando se recibe un mensaje, el <i>Service Activator</i> localiza e invoca a los métodos de los componentes de negocio necesarios para cumplir la petición de forma asíncrona.

1.3 Concepto de patrón de diseño

"Una arquitectura orientada a objetos bien estructurada está llena de patrones. La calidad de un sistema orientado a objetos se mide por la atención que los diseñadores han prestado a las colaboraciones entre sus objetos. Los patrones conducen a arquitecturas más pequeñas, más simples y más comprensibles". (Grady Booch)

Los patrones de diseño son **descripciones de clases** cuyos **objetos colaboran entre sí**. Cada patrón es adecuado para ser adaptado a un cierto tipo de problema. Para describir un caso se debe especificar lo siguiente:

- Nombre
- Propósito o finalidad
- Sinónimos (otros nombres por los que puede ser conocido)
- Problema al que es aplicable
- Estructura (diagrama de clases)
- Participantes (responsabilidad de cada clase)
- Colaboraciones (diagrama de interacciones)
- Implementación (consejos, notas y ejemplos)
- Otros patrones con los que está relacionado

Ventajas de los patrones:

La clave para la reutilización es anticiparse a los nuevos requisitos y cambios, de modo que los sistemas evolucionen de forma adecuada. Cada patrón permite que algunos aspectos de la estructura del sistema puedan cambiar independientemente de otros aspectos. Facilitan la reusabilidad, extensibilidad y mantenimiento.

Un patrón es un **esquema o microarquitectura** que supone una solución a problemas (dominios de aplicación) semejantes (aunque los dominios de problema pueden ser muy diferentes e ir desde una aplicación CAD a un cuadro de mando empresarial). Interesa constatar una vez más la vieja distinción entre dominio del problema (donde aparecen las clases propias del dominio, como cuenta, empleado, coche o beneficiario) y el dominio de la solución o aplicación (donde además aparecen clases como ventana, menú, contenedor o listener). Los patrones son patrones del dominio de la solución.

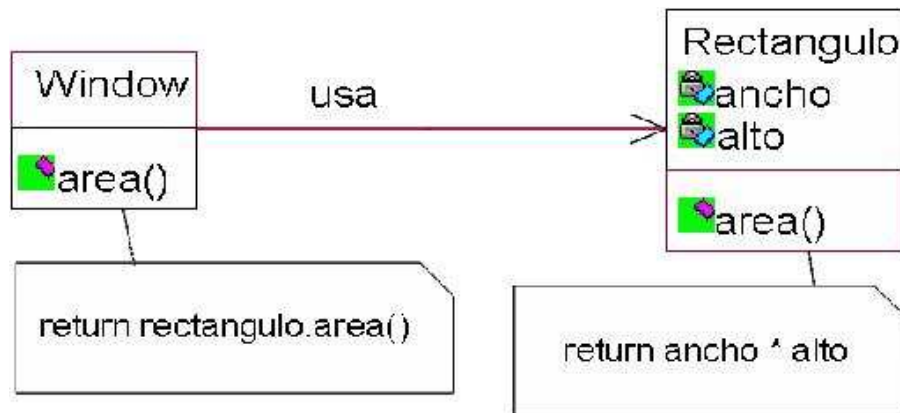
También, conviene distinguir entre un patrón y una arquitectura global del sistema. Por decirlo en breve, es la misma distancia que hay entre el diseño de un componente (o módulo) y el análisis del sistema. Es la diferencia que hay entre el aspecto micro y el macro; por ello, en ocasiones, se denomina a los patrones como "microarquitecturas".

En resumen, un patrón es el denominador común, una estructura común que tienen aplicaciones semejantes. Esto también ocurre en otros órdenes de la vida. Por ejemplo, en nuestra vida cotidiana, aplicamos a menudo el esquema saludo-presentación-mensaje-despedida en ocasiones diversas.

1.4 Patrón *Delegate* "delegado"

La herencia es útil para modelar relaciones de tipo es-un o es-una, ya que estos tipos de relaciones son de naturaleza estática. Sin embargo, relaciones de tipo es-un-rol-ejecutado-por son mal modeladas con herencia.

Un objeto receptor delega operaciones en su delegado. Presente en muchos patrones: State, Strategy, Visitor,...



Un ejemplo: supongamos que un objeto debe ordenar una estructura de datos. Puede delegar en otro objeto el método de comparación. En *Java*, tenemos un caso: la clase *Collections* tiene el método estático *sort()*. Desde este método se delega en un comparador para establecer el orden:

- *Collections* tiene el método *sort()* con un algoritmo de ordenación.
- *Comparador* tiene el método *compare()* que implementa el orden de comparación.

```

import java.util.Comparator;

public class comparador implements Comparator {
    public int compare( Object o1, Object o2 ) {
        if ( ((ente)o1).obt_id() < ((ente)o2).obt_id() )
            return -1;

        if ( ((ente)o1).obt_id() > ((ente)o2).obt_id() )
            return 1;

        return 0;
    }
}
  
```

1.5 Patrón "Modelo-Vista-Controlador"

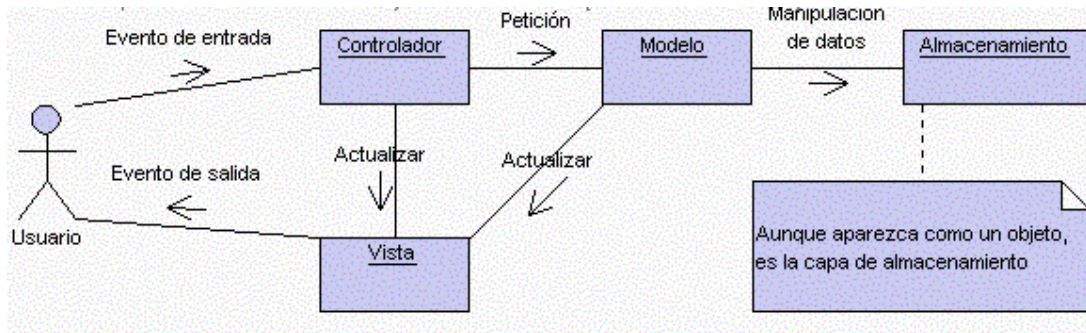
Para el diseño de aplicaciones con sofisticadas interfaz se utiliza el patrón de diseño Modelo-Vista-Controlador. La lógica de una interfaz de usuario cambia con más frecuencia que los almacenes de datos y la lógica de negocio. Si realizamos un diseño ofuscado, es decir, un pastiche que mezcle los componentes de interfaz y de negocio,

entonces la consecuencia será que, cuando necesitemos cambiar la interfaz, tendremos que modificar trabajosamente los componentes de negocio. Mayor trabajo y más riesgo de error.

Se trata de realizar un diseño que desacople la vista del modelo, con la finalidad de mejorar la reusabilidad. De esta forma, las modificaciones en las vistas impactan en menor medida en la lógica de negocio o de datos.

Elementos del patrón:

- Modelo: datos y reglas de negocio
- Vista: muestra la información del modelo al usuario
- Controlador: gestiona las entradas del usuario



Un modelo puede tener diversas vistas, cada una con su correspondiente controlador. Un ejemplo clásico es el de la información de una base de datos, que se puede presentar de diversas formas: diagrama de tarta, de barras, tabular, etc. Veamos cada componente:

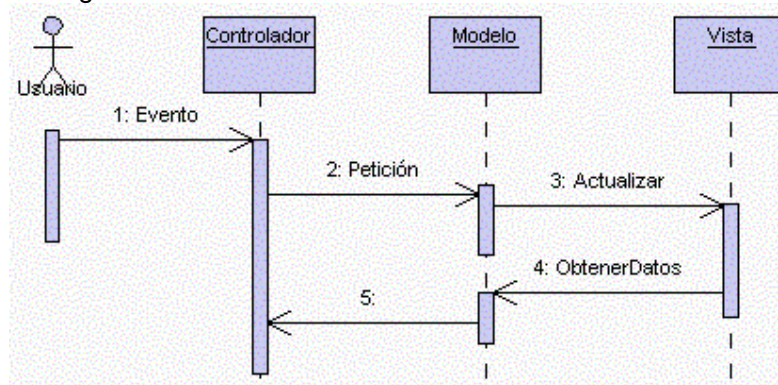
1. El **modelo** es el responsable de lo siguiente:
 - Acceder a la capa de almacenamiento de datos. Lo ideal es que el modelo sea independiente del sistema de almacenamiento.
 - Define las reglas de negocio (la funcionalidad del sistema). Un ejemplo de regla puede ser la siguiente: "Si la mercancía pedida no está en el almacén, consultar el tiempo de entrega estándar del proveedor".
 - Lleva un registro de las vistas y controladores del sistema.
 - Si estamos ante un modelo activo, notificará a las vistas los cambios que en los datos pueda producir un agente externo (por ejemplo, un archivo *bacth* que actualiza los datos, un temporizador que desencadena una inserción, etc).
2. El **controlador** es responsable de lo siguiente:
 - Recibe los eventos de entrada (un clic, un cambio en un campo de texto, etc.).
 - Contiene reglas de gestión de eventos, del tipo "SI Evento Z, entonces Acción W". Estas acciones pueden suponer peticiones al modelo o a las vistas. Una de estas peticiones a las vistas puede ser una llamada al método "Actualizar()". Una petición al modelo puede ser "Obtener_tiempo_de_entrega(nueva_orden_de_venta)".

3. Las **vistas** son responsables de lo siguiente:

- Recibir datos del modelo y los muestra al usuario.
- Tienen un registro de su controlador asociado (normalmente porque además lo *instancia*).
- Pueden dar el servicio de "Actualización()", para que sea invocado por el controlador o por el modelo (cuando es un modelo activo que informa de los cambios en los datos producidos por otros agentes).

Un ejemplo de MVC con un modelo pasivo (aquel que no notifica cambios en los datos) es la navegación web, que responde a las entradas del usuario, pero no detecta los cambios en datos del servidor.

El diagrama de secuencia:



Pasos:

1. El usuario introduce el evento.
2. El Controlador recibe el evento y lo traduce en una petición al Modelo (aunque también puede llamar directamente a la vista).
3. El modelo (si es necesario) llama a la vista para su actualización.
4. Para cumplir con la actualización, la Vista puede solicitar datos al Modelo.
5. El Controlador recibe el control.

Sin embargo, en su implementación, existe una pequeña dificultad: la mayor parte de las herramientas de desarrollo incorporan en las clases de la vista gran parte o todo el procesamiento de eventos, con lo que el controlador queda semiculto dentro de la vista. A pesar de ello, podemos acercarnos bastante al patrón. En el siguiente ejemplo en Java, el objeto vista es un Applet AWT. El controlador (controlador.java) puede gestionar el clic en un botón, de tal forma que recoge datos por medio del Modelo (model.cargar_texto(...)) y los manda a la Vista (el applet) para su actualización (vista.mostrar_texto()):

```

/*****
 *
 * Responde al click en botón "abrir"
 * La respuesta al evento es hacer que se abra en la vista
 * el archivo correspondiente a la referencia seleccionada en el
 * combo box
 *****/
/

```

```
void b_abrir_actionPerformed(ActionEvent e) {
    String texto_archivo = model.cargar_texto( indice_ref );
    // Obtener texto de archivo
    /*** Si la carga de archivo es ok, lo muestro. Si no,
    aviso de error ****/

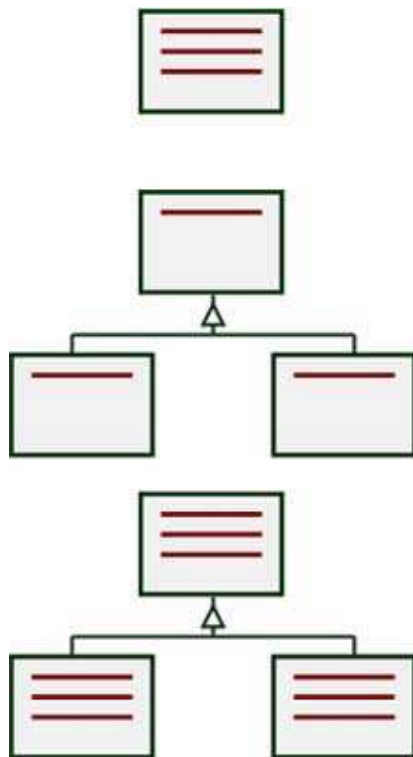
    if (texto_archivo != null) {
        vista.mostrar_texto(texto_archivo);          //
Mostrar texto
        vista.mostrar_aviso("Carga de " + path + "
completada.");
    }
    else
        vista.mostrar_aviso("Error en la carga de " +
path);
}
```

1.6 Patrón Factory "*Factoría*"

En realidad, son una familia de patrones:

- Factoría simple: una clase que crea objetos de otras clases. No delega en otras subclases y sus métodos pueden ser estáticos.
- Factory Method: Se define como una interfaz para crear objetos, como en el Abstract Factory, pero se delega a las subclases implementar la creación en concreto
- Abstract Factory: Nos da una interfaz para crear objetos de alguna familia, sin especificar la clase en concreto.

Los dos últimos están incluidos en el catálogo del GoF.



Simple Factory

Clase con la responsabilidad de crear objetos de otras clases. No delega en subclasses y sus métodos pueden ser estáticos. Puede evolucionar a un Factory Method o Abstract Factory

Factory Method

Define una interfaz para crear un objeto, pero deja que sean las subclasses quienes decidan qué clase instanciar. Permite que una clase delegue sus subclasses la creación de objetos

Abstract Factory

Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas

Estos patrones entran en la categoría de patrones de creación [GoF95], la cual comparten con otros patrones tales como el Singleton, Builder y Prototype [GoF95]. Tienen la responsabilidad de crear *instancias* de objetos de otras clases. Tienen, además, la responsabilidad y el conocimiento necesario para **encapsular la forma en que se crean determinados tipos de objetos** en una aplicación. Existen diferentes patrones de tipo Factory.

Factoria simple

Como **en todas las factorías**, tenemos las clases *instanciadas* (JuegoDelDado y JuegoDeMoneda) que se relacionan con una **clase madre** (extends) o con una **interfaz lógica** (implements). En nuestro caso, usamos interfaz.

A continuación, puede ver un sencillo ejemplo en el que cada juego implementa el método lanzar(): el juego del dado muestra un número aleatorio del 1 al 6 y el de la moneda 'Cara' o 'Cruz':

```
public interface Juego {
    void lanzar();
}

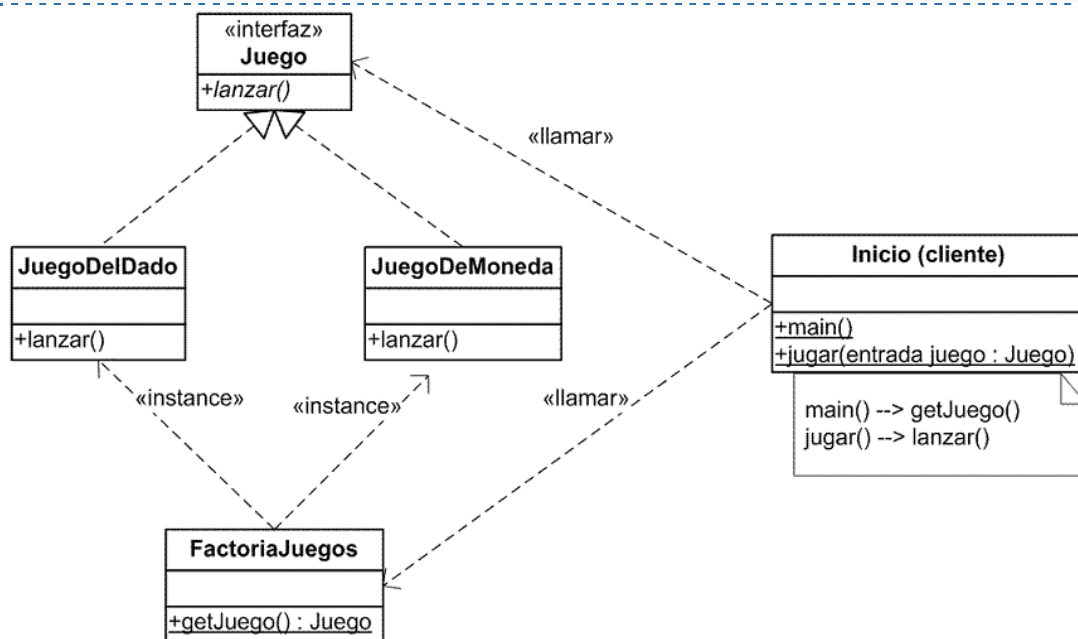
import java.util.Random;
public class JuegoDelDado implements Juego {
    public void lanzar() {
        Random ran = new Random();
        int resul = ran.nextInt(6) + 1;
        System.out.println( resul );
    }
}
```

```

    }
}

import java.util.Random;
public class JuegoDeMoneda implements Juego {
    public void lanzar() {
        Random ran = new Random();
        if ( ran.nextBoolean() )
            System.out.println( "Cara" );
        else
            System.out.println( "Cruz" );
    }
}

```



La clase FactoriaJuegos es única. No delega en una subclase la creación de *instancias* (a diferencia de Factory Method). Esta factoria es muy sencilla: en función del argumento, crea un juego u otro:

```

public class FactoriaJuegos {
    public static Juego getJuego( String nombreJuego ) {
        if ( nombreJuego.equals("JuegoDelDado") )
            return new JuegoDelDado();
        else {
            if ( nombreJuego.equals("JuegoDeMoneda") )
                return new JuegoDeMoneda();
            else
                return null;
        }
    }
}

```

```
}
```

Lo esencial de la clase Factoría es que oculta a la clase cliente (Inicio.java) la complejidad de crear un objeto. Encapsula la creación de la *instancia*.

La clase cliente (Inicio.java) llama al método estático getJuego() de la factoría, para que le devuelva el juego señalado en el argumento. Introduce todos los juegos en un vector y, a continuación, le dice a cada juego que juegue. El método jugar() es un ejemplo de **patrón 'estrategia'**: el método contiene un comportamiento genérico (en nuestro ejemplo, realiza dos lanzamientos para cada juego). El comportamiento específico se define en función del objeto que se pasa como argumento. La parte que varía es el argumento, esta es la estrategia.

```
public class Inicio {
    public static void main(String[] args) {
        //// Crea un vector de juegos
        Vector vec = new Vector();
        vec.add( FactoriaJuegos.getJuego( "JuegoDelDado" ) );
        vec.add( FactoriaJuegos.getJuego( "JuegoDeMoneda" ) );
        vec.add( FactoriaJuegos.getJuego( "JuegoDelDado" ) );

        //// A cada juego del vector le dice que juegue
        for ( int i = 0; i < vec.size(); i++ ) {
            Juego j = (Juego) vec.get(i);
            if ( j != null )
                jugar( j );
            else
                System.out.println("Juego no encontrado");
        }
    }
}

/*****
*****
    * Lanza dos veces
*****
*****/
    public static void jugar( Juego juego ) {
        juego.lanzar();
        juego.lanzar();
    }
}
```

Al recorrer el vector de juegos, vemos un ejemplo típico de **polimorfismo**: a cada referencia del tipo Juego (usamos el mismo interfaz) le decimos que juegue, pero cada juego implementa su forma específica de jugar (más en concreto de lanzar). Es la idea de polimorfismo: **una interfaz y múltiples implementaciones**.

Aunque la forma más **abstracta y profesional** es usar en la factoría **newInstance()** en función de los valores de un archivo **.properties**:

```
import java.io.*;

public class FactoriaJuegos {

    // true=Carga de Properties desde archivo
    private static boolean propiedadesCargadas = false;

    // Propiedades
    private static java.util.Properties prop = new
    java.util.Properties();

    /*****
    * Crea y devuelve el juego
    *****/

    public static Juego getJuego( String nombreJuego ) {
        try {
            //// La clase se consigue leyendo del
            archivo properties
            Class clase = Class.forName( getClass(
            nombreJuego ) );
            //// Creo una instancia
            return (Juego) clase.newInstance();
        }
        catch (ClassNotFoundException e) {           // No
            existe la clase
                e.printStackTrace();
                return null;
            }
        catch (Exception e) {
            // No puedo instanciar la clase
            e.printStackTrace();
            return null;
        }
    }

    /*****
    * Lee un archivo properties donde se indica la clase que debe
    ser instanciada
    *****/

    private static String getClass( String nombrePropiedad )
    {
        try {
            //// Carga de propiedades desde archivo
            if ( !propiedadesCargadas ) {
```

```

        FileInputStream archivo = new
FileInputStream(
    "src/factoriaJuegosNewInstanceProperties/propiedades.properties"
);
        prop.load( archivo );           // Cargo
propiedades
        propiedadesCargadas = true;
    }

    ///// Lectura de propiedad
    String nombreClase = prop.getProperty(
nombrePropiedad, "");
        if ( nombreClase.length() > 0 )
            return nombreClase;
        return null;
    }
    catch ( FileNotFoundException e) { // No se puede
encontrar archivo
        e.printStackTrace();
        return null;
    }
    catch ( IOException e) {                // Falla
load()
        e.printStackTrace();
        return null;
    }
    catch (Exception e) {
        e.printStackTrace();
        return null;    }    }}

```

Resumen

📖 El catálogo más famoso de patrones se encuentra en “**Design Patterns: Elements of Reusable Object-Oriented Software**”, de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, 1995, Addison-Wesley, también conocido como el libro GOF (Gang-Of-Four).

📖 Patrón Factory "Factoría"

En realidad, son una familia de patrones:

- *Factoría* simple: una clase que crea objetos de otras clases. No delega en otras subclases y sus métodos pueden ser estáticos.
- *Factory Method*: se define una interfaz para crear objetos, como en el *Abstract Factory*, pero se delega a las subclases implementar la creación en concreto.
- *Abstract Factory*: da una interfaz para crear objetos de alguna familia, sin especificar la clase en concreto.

📖 Si desea saber más acerca de estos temas, puede consulta la siguiente página.

🖱 <http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html>

UNIDAD DE
APRENDIZAJE

1

SEMANA

6

SERVLET Y JSP

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos implementarán una aplicación web en *Java* que realiza operaciones *SELECT*, *INSERT*, *UPDATE*, *DELETE* a la base de datos que contenga *Servlets*, *JSP*, *JavaBeans* y procese los objetos *request* y *response*.

TEMARIO

- Patrón *Data Access Object*
- Carrito de Compras

ACTIVIDADES PROPUESTAS

- Los alumnos realizarán programas utilizando el patrón DAO en la capa del Modelo de Datos.

1. DAO (*Data Access Object*)

1.1 Contexto

El acceso a los datos varía dependiendo de la fuente de los datos. El acceso al almacenamiento persistente, como una base de datos, varía en gran medida dependiendo del tipo de almacenamiento (bases de datos relacionales, bases de datos orientadas a objetos, archivos de texto, etc.) y de la implementación del vendedor.

1.2 Problema

Muchas aplicaciones de la plataforma JEE en el mundo real necesitan utilizar datos persistentes en algún momento. Para muchas de ellas, este almacenamiento persistente se implementa utilizando diferentes mecanismos, y hay marcadas diferencias en los APIs utilizados para acceder a esos mecanismos de almacenamiento diferentes o datos residentes en sistemas diferentes.

Típicamente, las aplicaciones utilizan componentes distribuidos compartidos persistentes como los *entity bean* para representar la data persistente. Una aplicación es considerada que empleará un *bean managed persistence* para sus *entity beans* cuando estas *entity beans* explícitamente accedan al almacén persistente, es decir, cuando el *entity bean* incluya código para acceder directamente al almacén persistente. Una aplicación con requerimientos simples no utilizaría *entity beans* y, en vez de ellos, utilizaría *session beans* o *Servlets* para acceder directamente al almacén persistente para recuperar y modificar datos.

Las aplicaciones pueden usar la API-JDBC para acceder a los datos que residen en un RDBMS. Esta API permite el acceso estándar y la manipulación de datos en un almacén persistente como una base de datos relacional. JDBC permite a las aplicaciones JEE utilizar sentencias SQL estándar. Sin embargo, las sentencias SQL podrían variar dependiendo del DBMS.

1.3 Solución

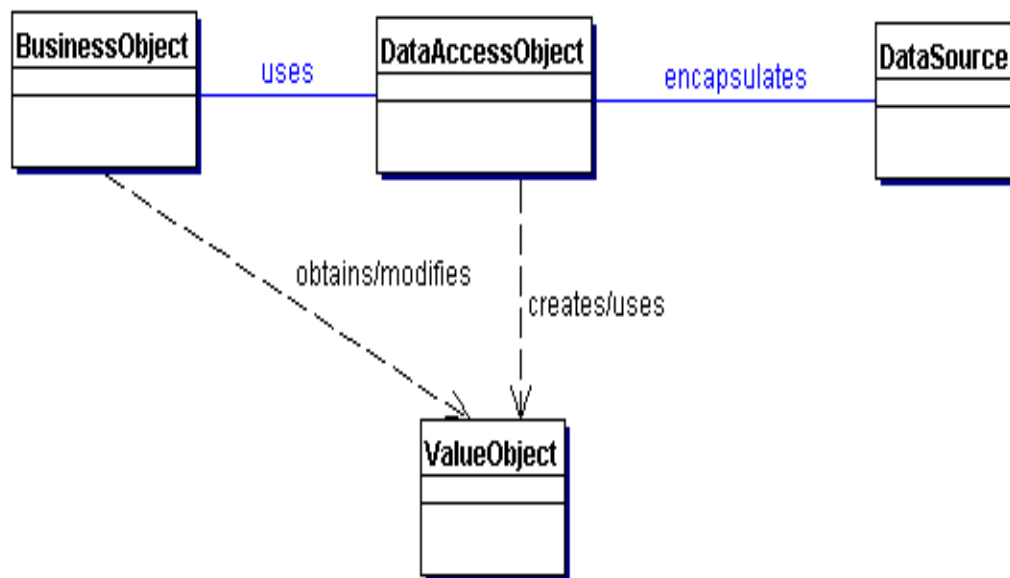
Utilizar un Data Access Object (DAO) para abstraer y encapsular todos los accesos a la fuente de datos. El DAO maneja la conexión con la fuente de datos para obtener y almacenar datos.

El DAO implementa el mecanismo de acceso requerido para trabajar con la fuente de datos.

El Objeto de Acceso a Datos (DAO) es el objeto principal de este patrón. La fuente de datos podría ser un almacén persistente como un RDBMS, un servicio externo como B2B, un servicio de negocios accesado vía CORBA.

Los componentes del negocio que cuentan con los objetos DAO utilizan una interfaz simple expuesta por el DAO para sus clientes. El DAO completamente oculta la implementación de la fuente de datos y lo aparta de los clientes, debido a que la interfaz expuesta por el DAO no cambia cuando la implementación de la fuente de datos cambia. Este patrón permite al DAO adaptarse a diferentes esquemas de almacenamiento sin afectar a sus clientes o componentes de negocio. DAO esencialmente actúa como un adaptador entre el componente de negocio y la fuente de datos (data source).

1.4 Diagrama de Clases



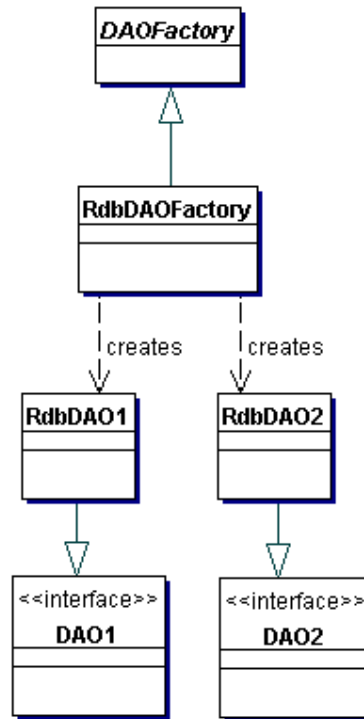
2 ESTRATEGIAS DE IMPLEMENTACIÓN DEL PATRÓN DAO

2.1 Factory

El patrón DAO se puede flexibilizar adoptando los patrones Factory y Abstract Factory.

Cuando el almacenamiento asociado a la aplicación no está sujeto a cambios de una implementación a otra, esta estrategia se puede implementar utilizando el patrón Factory para producir el número de

DAOs que necesita la aplicación. En la siguiente figura, podemos ver el diagrama de clases para este caso:



2.2 Abstract Factory

Cuando el almacenamiento asociado a la aplicación sí está sujeto a cambios de una implementación a otra, esta estrategia se podría implementar usando el patrón Abstract Factory. Este patrón a su vez puede construir y utilizar la implementación Factory.

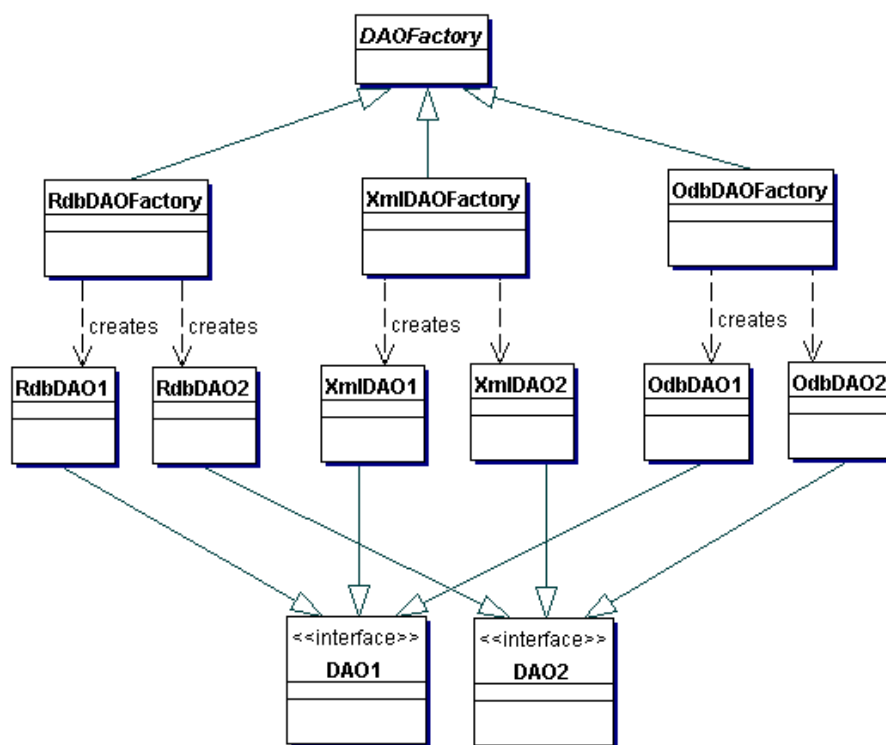
En este caso, esta estrategia proporciona un objeto factoría abstracta de DAOs (Abstract Factory) que puede construir varios tipos de factorías concretas de DAOs: cada factoría soporta un tipo diferente de implementación del almacenamiento persistente. Una vez que obtenemos la factoría concreta de DAOs para una implementación específica, la utilizamos para producir los DAOs soportados e implementados en esa implementación.

En la siguiente figura, podemos ver el diagrama de clases para esta estrategia. En él vemos una fábrica base de DAOs, que es una clase abstracta que extienden e implementan las diferentes factorías concretas de DAOs para soportar el acceso específico a la implementación del almacenamiento.

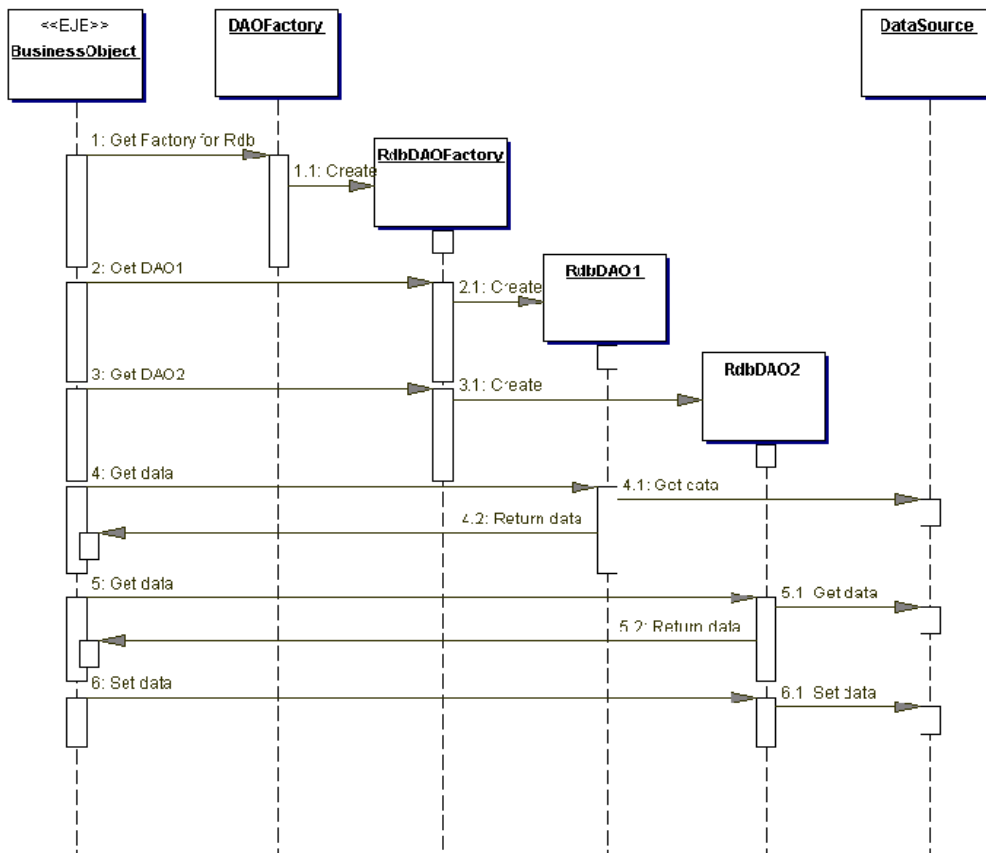
El cliente puede obtener una implementación de la factoría concreta del DAO como una RdbDAOFactory y utilizarla para obtener los DAOs concretos que funcionan en la implementación del almacenamiento.

Por ejemplo, el cliente puede obtener una RdbDAOFactory y utilizarlas para obtener DAOs específicos como RdbCustomerDAO, RdbAccountDAO, etc.

Los DAOs pueden extender e implementar una clase base genérica (mostradas como DAO1 y DAO2) que describa específicamente los requerimientos del DAO para el objeto de negocio que soporta. Cada DAO concreto es responsable de conectar con la fuente de datos y de obtener y manipular los datos para el objeto de negocio que soporta.



En la siguiente figura, podemos ver el diagrama de secuencia para esta estrategia:



2.3 Contexto resultante

DAO permite la transparencia de los detalles de la implementación de la fuente de datos. El acceso es transparente porque los detalles de la implementación se ocultan dentro del DAO.

La capa de DAO's permite una migración más fácil, hace más fácil que una aplicación pueda migrar a una implementación de base de datos diferente. Los objetos de negocio no conocen la implementación de datos subyacente; la migración implica cambios sólo en la capa DAO.

Reduce la Complejidad del Código de los Objetos de Negocio.

Centraliza todos los accesos a datos en una capa independiente.

3. CARRITO DE COMPRAS

3.1 Definición

El carrito de compras es una funcionalidad típica de toda aplicación web comercial y permite a un usuario adquirir productos y/o servicios de manera fácil e intuitiva.

3.2 Características básicas

El carrito de compras que implementaremos será almacenado en la sesión del usuario a través del servlet ServletCarrito. Toda la administración del carrito se manejará a través del servlet.

Como funcionalidades básicas se implementará la funcionalidad de agregar productos desde el catálogo de la empresa hacia el carrito. El número de Ítems agregados al carrito podrá ser modificado por el usuario.

En la parte inferior del carrito, se encontrarán varios enlaces que permitirán cambiar o borrar un ítem del carrito o vaciar totalmente el carrito.

El enlace "Continuar compra" nos lleva nuevamente al catálogo de productos para que seleccionemos ítems adicionales.

Una vez que estemos de acuerdo con lo que se encuentra dentro del carrito de compras, debemos de seleccionar "Terminar pedido " para confirmar nuestra orden.

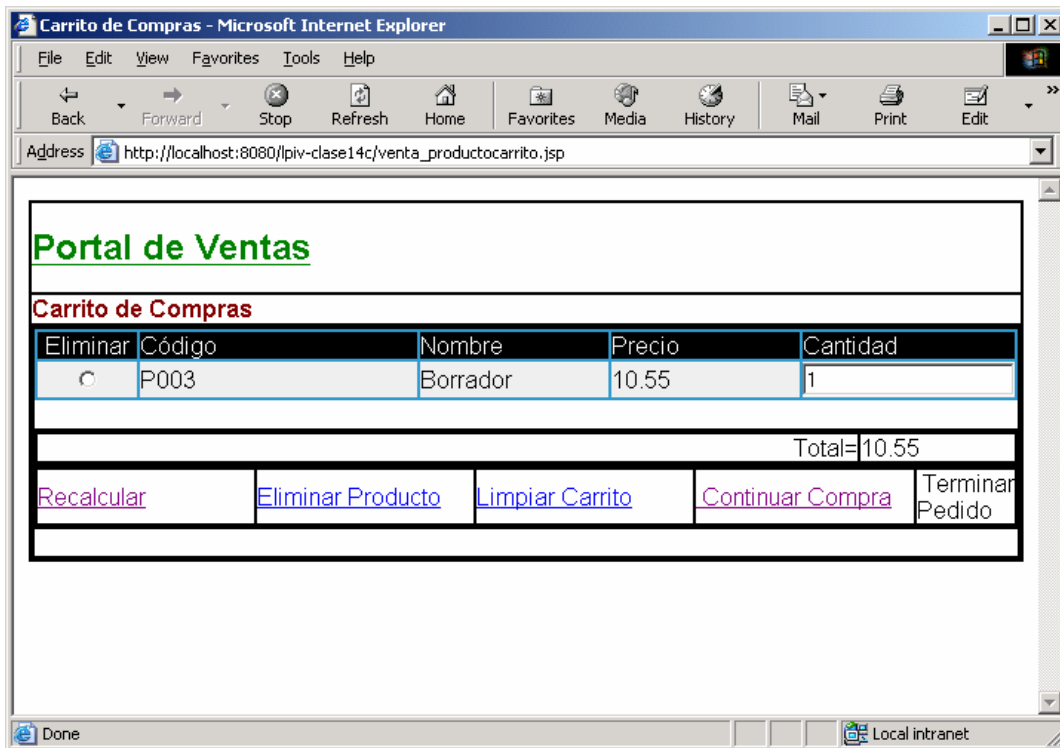
Ejercicio #1:

Cree una aplicación web de nombre **Ipiv-clase14c** e implemente la funcionalidad de carrito de compras utilizando DAO y la clase Lista.java.

Catálogo de Productos

Portal de Ventas			
Lista de productos			
Pag: 1 2			
Código	Nombre	Precio	
P001	Cuaderno	34.55	Comprar
P002	Lapiz	80.55	Comprar
P003	Borrador	10.55	Comprar
P004	Tajador	500.55	Comprar

Carrito de Compras



Script tabla Producto

```
CREATE TABLE [dbo].[PRODUCTO] (
    [CHRPROCODIGO] [char] (4) NOT NULL ,
    [VCHPRONOMBRE] [varchar] (50) NULL ,
    [VCHPRODESCR] [varchar] (500) NULL ,
    [DECPRECIO] [decimal](18, 3) NULL ,
    [INTSTOCK] [int] NULL ,
    [VCHRUTAIMAGEN] [varchar] (50) NULL
) ON [PRIMARY]
GO

ALTER TABLE [dbo].[PRODUCTO] WITH NOCHECK ADD
    CONSTRAINT [PK_PRODUCTO] PRIMARY KEY NONCLUSTERED
        (
            [CHRPROCODIGO]
        ) ON [PRIMARY]
GO
```

venta_productomain.jsp

JSP que visualiza el catálogo de productos y a partir del cual se accede al carrito de compras.

```

<%@page import="edu.cibertec.bean.*" %>
<%@page import="edu.cibertec.util.*" %>
<%@page import="edu.cibertec.admin.dao.*" %>

<%

    DAOFactory daoFactory=DAOFactory.getDAOFactory(DAOFactory.MYSQL);
    ProductoDAO objProductoDAO =daoFactory.getProductoDAO();

    //Para el Nro. de Pagina.
    int intPagina=0;

    //Si la página es nula entonces debe ser 1;
    intPagina=(request.getParameter("pagina")==null?1:Integer.parseInt(request.getPar
    ameter("pagina")));

    Lista objLista=objProductoDAO.obtenerProductos();

    //Número de registro por páginas.
    int iTamanoPagina=4;
    //iTamanoPagina=Integer.parseInt(String.valueOf(application.getAttribute("AppTaman
    oPagina")));
    objLista.setTamPagina(iTamanoPagina);
%>

<html>
<head>
<meta http-equiv="Content-Language" content="es">
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>Carrito de Compras</title>
</head>

<body>

    <table border="1" width="100%" bordercolor='#000000' cellspacing="0"
    cellpadding="0">
        <tr>
            <td width="100%" height="62">
                <p><font size="5" face="GulimChe" color="#008000"><b><u>Portal de
Ventas</u></b></font></td>
            </tr>
            <tr>
                <td width="100%" height="19"><b><font color="#800000">Lista de
                productos</font></b></td>
            </tr>

```

```

<tr>
  <td width="100%" height="19">
    <font size="1" face="Verdana">Pag:
    <%for(int i=1;i<=objLista.getNumPagina();i++){
      if(i==intPagina){
        out.print(i+" ");
      }
    }
    <a
href="venta_productomain.jsp?pagina=<%=i%>"><%=i%></a>
    <%
    }
  }
  </font>
</td>
</tr>
<tr>
  <td width="100%" height="19">
    <table border="1" bordercolor="#3399CC" width="100%" cellpadding="0" cellspacing="0">
      <tr>
        <td width="25%" bgcolor="#000000"><font
color="#FFFFFF">Código</font></td>
        <td width="25%" bgcolor="#000000"><font
color="#FFFFFF">Nombre</font></td>
        <td width="25%" bgcolor="#000000"><font
color="#FFFFFF">Precio</font></td>
        <td width="25%" bgcolor="#000000">&nbsp;</td>
      </tr>

      <%for(int i=objLista.getFirstElementPage(intPagina)-
1;i<=objLista.getLastElementPage(intPagina)-1;i++){%>
        <tr>
          <td
width="25%"><%=((Producto)objLista.getElemento(i)).getCodigo()%></td>
          <td
width="25%"><%=((Producto)objLista.getElemento(i)).getNombre()%></td>
          <td
width="25%"><%=((Producto)objLista.getElemento(i)).getPrecio()%></td>
          <td width="25%"><A
href="Carrito?accion=comprar&codigo=<%=((Producto)objLista.getElemento(i)).getC
odigo()%>&nombre=<%=((Producto)objLista.getElemento(i)).getNombre()%>&precio
=<%=((Producto)objLista.getElemento(i)).getPrecio()%>&cantidad=1">Comprar</A><
/td>
        </tr>
      <%}%>
    </table>
  </td>
</tr>
</table>
</body>
</html>

```

ServletCarrito.java

Implementa toda la funcionalidad base del Carrito de compras.

```
package edu.cibertec.admin.servlet;

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

public class ServletCarrito extends javax.servlet.http.HttpServlet {

    public void elimina(
        javax.servlet.http.HttpServletRequest request,
        javax.servlet.http.HttpServletResponse response)
        throws javax.servlet.ServletException, java.io.IOException {

        // --- Para código de producto.
        String sCodigo;

        // --- Para manejo de la sesión.
        HttpSession se = request.getSession();

        sCodigo = request.getParameter("optCodigo");

        // --- Para eliminar.
        if (sCodigo != null) {
            // --- Declaramos una tabla.
            Hashtable htCarrito;

            // --- Obtenemos la tabla a partir de la sesión.
            htCarrito = (Hashtable) (se.getAttribute("objCarrito"));

            // --- Elimino el elemento.
            htCarrito.remove(sCodigo);

            // --- Guardamos la tabla actualizada en la sesión.
            se.setAttribute("objCarrito", htCarrito);
        }

        // --- Mostramos el contenido del carrito.
        response.sendRedirect("venta_productocarrito.jsp");
    }

    public void limpia(
        javax.servlet.http.HttpServletRequest request,
        javax.servlet.http.HttpServletResponse response)
        throws javax.servlet.ServletException, java.io.IOException {
        // --- Para manejo de la sesión.
        HttpSession se = request.getSession();

        // --- Declaramos una tabla.
        Hashtable htCarrito;
```

```

public void service(
    javax.servlet.http.HttpServletRequest request,
    javax.servlet.http.HttpServletResponse response)
    throws javax.servlet.ServletException, java.io.IOException {

    if (request.getParameter("accion").equals("eliminar")) {
        elimina(request, response);
    } else if (request.getParameter("accion").equals("limpiar")) {
        limpia(request, response);
    } else if (request.getParameter("accion").equals("recalcular")) {
        recalcular(request, response);
    } else {
        update(request, response);
    }

}

public void update(
    javax.servlet.http.HttpServletRequest request,
    javax.servlet.http.HttpServletResponse response)
    throws javax.servlet.ServletException, java.io.IOException {
    // --- Declaramos variables para almacenar ls parámetros.
    String sCodigo = null;
    String sNombre = null;
    String sPrecio = null;
    String sCantidad = null;

    // --- Creamos una matriz para guardar valores.
    String saValores[] = new String[3];

    // --- Recibimos los parámetros.
    sCodigo = request.getParameter("codigo");
    sNombre = request.getParameter("nombre");
    sPrecio = request.getParameter("precio");
    sCantidad = request.getParameter("cantidad");

    // --- Ingresamos valores a la matriz.
    saValores[0] = sNombre;
    saValores[1] = sPrecio;
    saValores[2] = sCantidad;

    // --- Para manejar la sesión.
    HttpSession se = request.getSession();

    // --- Verificamos si existe el carrito
    if (se.getAttribute("objCarrito") == null) {
        // --- Como no existe creamos la tabla.
        Hashtable htCarrito = new Hashtable();

        // --- Guardamos valores en la tabla.
        htCarrito.put(sCodigo, saValores);
    }
}

```



```

        // --- Guardamos la tabla con nuevas filas en la sesión.
        se.setAttribute("objCarrito", htCarrito);
    }

    //Mostramos el contenido del carrito.
    response.sendRedirect("venta_productocarrito.jsp");

}

public void recalcular(
    javax.servlet.http.HttpServletRequest request,
    javax.servlet.http.HttpServletResponse response)
    throws javax.servlet.ServletException, java.io.IOException {

    // --- Para manejo de la sesión.
    HttpSession se = request.getSession();

    // --- Declaramos una tabla.
    Hashtable htCarrito;

    // --- Obtenemos la tabla a partir de la sesión.
    htCarrito = (Hashtable) (se.getAttribute("objCarrito"));

    int valor = 0;
    Enumeration productos = htCarrito.keys();
    while (productos.hasMoreElements()) {
        String sClave = (String) productos.nextElement();
        String saValores[] = (String[]) htCarrito.get(sClave);

        // --- obtenemos la cantidad ingresada en la caja de texto
        String nuevaCantidad = request.getParameter("txt" + sClave);

        boolean flag = false;
        try {
            int x = Integer.valueOf(nuevaCantidad).intValue();
            flag = true;
        } catch (Exception e) {
        }

        // --- si no se ingreso cantidad, se ingresaron letras o se ingreso
        el número cero

        // --- se coloca por defecto el número uno en el campo cantidad
        if (nuevaCantidad.equals("")
            || nuevaCantidad.equals("0")
            || flag == false)
            valor = 1;
        else
            valor = Integer.valueOf(nuevaCantidad).intValue();

        // --- cambiamos el valor del campo cantidad
        saValores[2] = "" + valor;
    }
}

```

```

        // --- Guardamos los nuevos valores en la tabla.
        htCarrito.put(sClave, saValores);
    }
    // --- Guardamos la tabla actualizada en la sesión.
    se.setAttribute("objCarrito", htCarrito);

    // --- Mostramos el contenido del carrito.
    response.sendRedirect("venta_productocarrito.jsp");
}
}

```

venta_productocarrito.jsp

JSP que visualiza el carrito de compras e invoca al ServletCarrito para administrar el carrito.

```

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

<%
//Para las claves.
String sClave;
//Para los datos.
String saValores[];
//Valores Iniciales
int iCantidad=0;double dPrecio=0;double dSubTotal=0;double dTotal=0;int
iContador=0; String sColor=null;
//Obtenemos el carrito lleno
Hashtable htCarrito=(Hashtable)session.getAttribute("objCarrito");
HttpSession se= request.getSession();
if (se.getAttribute("objCarrito") == null)
{
    response.sendRedirect("venta_productomain.jsp");
}

Enumeration enIndices=htCarrito.keys();

%>

<html>

<head>
<meta http-equiv="Content-Language" content="es">
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>Carrito de Compras</title>
<script language="JavaScript">
<!--
    window.name='main'
    function f_eliminar() {
        document.forms[0].accion.value='eliminar';
        document.forms[0].action='Carrito';
        document.forms[0].target='main';
        document.forms[0].submit();
    }
}

```

```

        dSubTotal= iCantidad * dPrecio;
        dTotal = dTotal + dSubTotal;
        iContador++;

        if (iContador% 2 == 0)
            sColor="white";
        else
            sColor="#EFEFEF";

    %>
    <tr bgcolor=<%=sColor%>>
        <td width="16%" align="center"><input type="radio" name="optCodigo"
value="<%=sClave%>"></td>
        <td width="34%"><%=sClave%></td>
        <td width="25%"><%=saValores[0]%></td>
        <td width="25%"><%=saValores[1]%></td>
        <td width="25%"><input type="text" name="txt<%=sClave%>"
value="<%=saValores[2]%>"></td>
    </tr>
    <%=}%>
</table>
</form>
    </td>
</tr>
<tr>
    <td width="100%">
        <table border="1" bordercolor='#000000' width="100%" cellpadding="0" cellspacing="0">
            <tr>
                <td width="84%">
                    <p align="right">Total=</td>
                <td width="16%"><%=dTotal%></td>
            </tr>
        </table>
    </td> </tr>
<tr><td width="100%">
    <table border="1" bordercolor='#000000' width="100%" cellpadding="0" cellspacing="0">
        <tr>
            <td width="25%"><a
href="javascript:f_recalcular();">Recalcular</a></td>
            <td width="25%"><a href="javascript:f_eliminar();">Eliminar
Producto</a></td>
            <td width="25%"><a href="Carrito?accion=limpiar">Limpiar
Carrito</a></td>
            <td width="25%"><a href="venta_productomain.jsp">&nbsp;Continuar
Compra</a></td>
            <td width="25%">&nbsp;Terminar Pedido</td>
        </tr>
    </table>
</td>
</tr><tr> <td width="100%">&nbsp;</td> </tr>
</table>
</body>
</html>

```

```

function f_recalcular() {
    document.forms[0].accion.value='recalcular';
    document.forms[0].action='Carrito';
    document.forms[0].target='main';
    document.forms[0].submit();
}

// -->
</script>
</head>

<body>


    <table border="1" bordercolor='#000000' width="100%" cellpadding="0" cellspacing="0">
    <tr>
        <td width="100%" height="62">
            <p><font size="5" face="GulimChe" color="#008000"><b><u>Portal de
Ventas</u></b></font></td>
        </tr>
        <tr>
            <td width="100%" height="19"><b><font color="#800000">Carrito de
Compras</font></b></td>
        </tr>
        <tr>
            <td width="100%" height="19">
                <table border="1" bordercolor='#000000' width="100%" cellpadding="0" cellspacing="0">
                <tr>
                    <td width="100%">
                        <form>
                            <input type="hidden" name="accion">
                            <table border="1" bordercolor='#3399CC' cellpadding="0" cellspacing="0" width="100%">
                            <tr>
                                <td width="16%" bgcolor="#000000" align="center"><font
color="#FFFFFF">Eliminar</font></td>
                                <td width="34%" bgcolor="#000000"><font
color="#FFFFFF">Código</font></td>
                                <td width="25%" bgcolor="#000000"><font
color="#FFFFFF">Nombre</font></td>
                                <td width="25%" bgcolor="#000000"><font
color="#FFFFFF">Precio</font></td>
                                <td width="25%" bgcolor="#000000"><font
color="#FFFFFF">Cantidad</font></td>
                            </tr>


                            <%
                                while(enIndices.hasMoreElements())
                                {
                                    sClave=(String)enIndices.nextElement();
                                    saValores=(String[])htCarrito.get(sClave);

                                    iCantidad = Integer.parseInt(saValores[2]);
                                    dPrecio = Double.parseDouble(saValores[1]);
                                }
                            <%
                        </td>
                    </td>
                </tr>
            </td>
        </tr>
    </table>
    </td>
    </tr>
    </table>


```

Resumen

 El Patrón DAO está compuesto de dos patrones Factory y Abstract Factory.

 El HashTable permanece guardado en la sesión.

```
Hashtable htCarrito=(Hashtable)session.getAttribute("objCarrito");  
HttpSession se= request.getSession();
```

 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

 <http://java.sun.com/j2se/1.4.2/docs/api/java/util/Hashtable.html>

Aquí hallará la descripción de los métodos del HashTable.

 http://es.wikipedia.org/wiki/Data_Access_Object

Aquí hallará definiciones del patrón DAO.

 http://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o

Aquí hallará definiciones de patrones.

**UNIDAD DE
APRENDIZAJE**

2

SEMANA

9

SEGURIDAD DE APLICACIONES WEB

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos crearán aplicaciones web en Java que utilice para seguridad: *BASIC*, *DIGEST*, *FORM*, and *CLIENT-CERT*, en función de un requerimiento dado.

TEMARIO

- Autenticación, Autorización, Confidencialidad, Integridad de la data
- Autenticación basado en *BASIC*, *DIGEST*, *FORM*, and *CLIENT-CER*

ACTIVIDADES PROPUESTAS

- Los alumnos crearán aplicaciones que permitan utilizar una seguridad adecuada.

1 Mecanismos de seguridad

La seguridad en los *Servlets* se reduce a cuatro conceptos principales:

- **authentication:** El proceso de autenticación es el encargado de determinar que el interesado en acceder a los recursos es realmente quien dice que es. Lo más común es hacer que el usuario o la aplicación envíe un password que lo identifica.
- **authorization:** Es el proceso por el cual se restringe el acceso a ciertos recursos a un conjunto de clientes. En realidad, lo que se define es un conjunto de roles que pueden acceder a ciertos recursos y se asocia cada usuario a uno o más roles.
- **data integrity:** Es el proceso que asegura que los datos que se reciben (tanto el cliente como el servidor) no han sido corrompidos. Generalmente se usan algoritmos de encriptación para lograr esto.
- **confidentiality:** Es el proceso que intenta asegurar que solamente los clientes autorizados reciban la información. La manera de lograr esto puede ser a través del uso de claves públicas/privadas.

2 Tipos de autenticación definidos

La especificación de servlet ofrece cuatro tipos de autenticación:

- **BASIC:** Este es el método más simple, pero el más inseguro. Cuando usamos este tipo de autenticación, se abre una ventana en el browser del cliente para que este ingrese el usuario y el password. El password se envía al servidor codificado en Base64.
- **FORM:** Este método es similar a BASIC, pero utiliza un formulario provisto por el programador. Este formulario tiene que cumplir los siguientes requisitos:
 1. El atributo "action" debe ser "**j_security_check**"
 2. El casillero de ingreso del usuario debe llamarse "**j_username**"
 3. El casillero de ingreso del password debe llamarse "**j_password**"

```
<form method="post" action="j_security_check">  
  Usuario: <input type="text" name="j_username">  
  Password: <input type="text" name="j_password">  
</form>
```

- **CLIENT-CERT:** En este método se usan certificados digitales. El browser debe tener instalado el certificado para que la comunicación se pueda establecer.

- **DIGEST:** La autenticación también es hecha utilizando usuario y password, pero el password es enviado encriptado de una forma mucho más segura (por ejemplo, utilizando HTTPS client authentication).

3 Autorización

El primer paso es definir los roles que se encontrarán definidos dentro de nuestro sistema. Para el caso de *Tomcat*, estos se definen en el archivo ***tomcat-users.xml***. Dentro de este archivo, cada usuario tiene asignado un usuario y *password*, y puede estar asociado a uno o más roles.

```
<tomcat-users>
  <role rolename="Admin"/>
  <role rolename="Member"/>
  <role rolename="Guest"/>
  <user username="jose" password="duke" roles="Admin,">

</tomcat-users>
```

El siguiente paso es relacionar los roles definidos en el archivo ***tomcat-users.xml***, con los roles a utilizar dentro del sistema. Para esto deberemos utilizar los elementos ***<security-role>*** y ***<role-name>*** dentro del archivo ***web.xml***.

```
<security-role>
  <role-name>Admin</role-name>
</security-role>

<security-role>
  <role-name>Member</role-name>
</security-role>

<security-role>
  <role-name>Guest</role-name>
</security-role>
```

Una vez definidos los roles, se procede a establecer “***declarativamente***” el acceso a un conjunto de recursos en combinación con los métodos, a los cuales tendrán accesos determinados roles. El elemento principal es el ***<security-constraint>***.

```
<security-constraint>

  <web-resource-collection>
    <web-resource-name>ActualizarDatos</web-resource-name>
    <url-pattern>/cl/jug/actualizar/*</url-pattern>
```

```

    <url-pattern>/cl/jug/listar/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>

  <auth-constraint>
    <role-name>Admin</role-name>
    <role-name>Member</role-name>
  </auth-constraint>

</security-constraint>

```

- El elemento **<web-resource-name>** es obligatorio y utilizado principalmente por los IDEs.
- El elemento **<url-pattern>** define el recurso al cual se limitará el acceso. Al menos se debe definir un **<url-pattern>**.
- El elemento **<http-method>** describe el método HTTP que será restringido para acceder al recurso definido en los **<url-pattern>**. Si no se define ningún **<http-method>**, entonces se restringirá el acceso a todos los métodos HTTP.
- El elemento **<auth-constraint>**, es opcional y permite definir cuáles roles pueden llamar a los métodos HTTP restringidos por los elementos **<http-method>**.

Para permitir el acceso a todos los roles, se debe definir el elemento **<auth-constraint>** de la siguiente forma:

```

<auth-constraint>
<role-name>*</role-name>
</auth-constraint>
<auth-constraint></auth-constraint>

```

En el caso de que se quiera bloquear el acceso a todos los roles, se deberá definir el **<auth-constraint>** de la siguiente forma:

```

<auth-constraint/>

```

Es importante tener en cuenta que se puede tener más de un elemento **<web-resource-collection>** dentro de un **<security-constraint>**. En este caso, el elemento **<auth-constraint>** se aplica a todos los **<web-resource-collection>** definidos dentro de un **<security-constraint>**.

4 Autenticación

Anteriormente mencionamos que la especificación de servlet ofrece cuatro tipos de autenticación: **BASIC**, **FORM**, **CLIENT-CERT** y **DIGEST**.

Cada uno de estos tipos se puede definir en el **web.xml** de la siguiente forma:

```
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>

<login-config>
<auth-method>FORM</auth-method>
<form-login-config>
  <form-login-page>/loginUsuario.html</form-login-page>
  <form-error-page>/loginError.html</form-error-page>
</form-login-config>
</login-config>

<login-config>
  <auth-method>CLIENT-CERT</auth-method>
</login-config>

<login-config>
  <auth-method>DIGEST</auth-method>
</login-config>
```

Un punto importante es que de los cuatro tipos de autenticación, el tipo **DIGEST** es el único que es opcional para los contenedores JEE.

5 Modos de transporte

La especificación también define tres modos para indicar la seguridad del transporte de los datos a través de la web.

- **NONE**: Ningún tipo de cifrado, simplemente HTTP.
- **CONFIDENTIAL**: Define la confidencialidad de los datos. Es decir, que nadie pueda ver ni modificar los datos que se envían.
- **INTEGRAL**: Define la integridad de los datos; es decir que, aunque alguien pueda ver lo que se envía, se garantice que los datos no se modifiquen por el camino.

```
<security-constraint>

  <web-resource-collection>
    <web-resource-name>ActualizarDatos</web-resource-name>
    <url-pattern>/cl/jug/actualizar/*</url-pattern>
    <url-pattern>/cl/jug/listar/*</url-pattern>
```

```

        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>

    <auth-constraint>
        <role-name>Admin</role-name>
        <role-name>Member</role-name>
    </auth-constraint>

    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-
guarantee>
    </user-data-constraint>

</security-constraint>

```

6 El método isUserInRole()

La clase “**HttpServletRequest**” tiene tres métodos que están asociados al manejo de la seguridad de manera “**programática**”. Uno de estos es el método “**isUserInRole**”.

A través del método “**isUserInRole()**”, en vez de manejar el acceso a nivel de método HTTP, ahora podremos autorizar el acceso a una parte del código de nuestro método basados en los roles del usuario.

Antes de que se llame al método “**isUserInRole()**”, el usuario necesita estar autenticado. En el caso de que llame al método sobre un usuario que no ha autenticado, el contenedor retornará “**false**”.

En el caso de que esté autenticado, el contenedor toma el argumento del método y compara este con los roles definidos para el usuario en el “**request**”.

Si el usuario es *ubicado* a este rol, el contenedor retornará “**true**”. Por ejemplo, dentro del código en el *servlet*:

```

if( request.isUserInRole("DukeChile")) {
    // Permiso para actualizar la informacion
}
else {
    // permiso solo para consultar la informacion
}

```

```

<servlet>
    <security-role-ref>
        <role-name> DukeChile </role-name>

```

```

        <role-link>Admin</role-link>
    </security-role-ref>
    ...
</servlet>

<security-role>
    <role-name>Admin</role-name>
</security-role>

```

El elemento **<security-role-ref>** permite relacionar un **<role-name>** definido “*programáticamente*” (en el servlet) con uno que existe en el **web.xml**.

Preguntas de Certificación

1. ¿Cuál de las siguientes sentencias sigue los conceptos de integridad? (Seleccione uno.)

- a It guarantees that information is accessible only to certain users.
- b It guarantees that the information is kept in encrypted form on the server.
- c It guarantees that unintended parties cannot read the information during transmission between the client and the server.
- d It guarantees that the information is not altered during transmission between the client and the server.

2. ¿Cuál de los siguientes procesos se utiliza cuando un usuario ha accedido a un recurso en particular? (Seleccione uno.)

- a Authorization
- b Authentication
- c Confidentiality
- d Secrecy

3. ¿Cuál de los siguientes procesos se realiza después de haber hecho la autorización? (Seleccione uno.)


- a Data validation
- b User authentication
- c Data encryption
- d Data compression

4. ¿Cuáles de las siguientes acciones deberán ejecutarse para prevenir que su sitio web sea atacado? (Seleccione tres.)

- a Block network traffic at all the ports except the HTTP port.
- b Audit the usage pattern of your server.
- c Audit the Servlet/JSP code.
- d Use HTTPS instead of HTTP.

Resumen

 Existen cuatro tipos de autenticación: **BASIC**, **FORM**, **CLIENT-CERT** y **DIGEST**.

 Si desea saber más acerca de estos temas, puede consultar la siguiente página.

 <http://courses.coreservlets.com/Course-Materials/scwcd.html>

En esta página, hallará descripción sobre seguridad en *Java* web.

**UNIDAD DE
APRENDIZAJE**

3

SEMANA

10

STANDARD ACTIONS

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos crearán aplicaciones web en Java que utilice en los JSP la tecnología *Standard Actions*, que utilice `jsp:useBean`, `jsp:setProperty`, `jsp:getProperty`, `jsp:atribute` en la implementación de cada JSP.

TEMARIO

- `jsp:useBean`, `jsp:setProperty`, `jsp:getProperty`, `jsp:atribute`
- `jsp:include`, `jsp:forward`

ACTIVIDADES PROPUESTAS

- Los alumnos crearán aplicaciones que permitan utilizar los *tag* estándares.

1 La acción <jsp:include>

Esta acción incluye en una página la salida generada por otra perteneciente a la misma aplicación web. La petición se redirige a la página incluida, y la respuesta que genera se incluye en la generada por la principal. Su sintaxis es la siguiente:

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

El atributo *flush* especifica si el flujo de salida de la página principal debería ser enviado al cliente antes de enviar el de la página incluida. En JSP 1.2 este atributo es optativo, y su valor por defecto es *false*. En JSP 1.1 es obligatorio y siempre debía valer *true* (el forzar el vaciado de buffer era problemático, porque una vez que ha sucedido esto no se pueden hacer redirecciones ni ir a páginas de error, ya que ya se han terminado de escribir las cabeceras).

Esta acción presenta la ventaja sobre la directiva del mismo nombre de qué cambios en la página incluida no obligan a recompilar la "principal". No obstante, la página incluida solo tiene acceso al *JspWriter* de la "principal" y no puede generar cabeceras (por ejemplo, no puede crear *cookies*).

Por defecto, la petición que se le pasa a la página incluida es la original, pero se le pueden agregar parámetros adicionales, mediante la etiqueta *jsp:param*. Por ejemplo:

```
<jsp:include page="cabecera.jsp">  
  <jsp:param name="color" value="YELLOW" />  
</jsp:include>
```

La acción <jsp:plugin>

Esta acción sirve para incluir, de manera portable e independiente del navegador, *applets* que utilicen alguna librería de *Java 2* (*Swing*, colecciones, *Java 2D*), ya que las máquinas virtuales *Java* distribuidas con algunos navegadores relativamente antiguos (Explorer 5.x, Netscape 4.x) son de una versión anterior a *Java 2*.

La acción <jsp:forward>

Esta acción se utiliza para redirigir la petición hacia otra página JSP que esté en la misma aplicación web que la actual. Un ejemplo de su sintaxis básica es la siguiente:

```
<jsp:forward page="principal.jsp"/>
```

La salida generada hasta el momento por la página actual se descarta (se borra el buffer). En caso de que no se utilizara buffer de salida, se produciría una excepción.

Al igual que en el caso de `<jsp:include>`, se pueden añadir parámetros a la petición original para que los reciba la nueva página JSP:

```
<jsp:forward page="principal.jsp">
  <jsp:param name="privilegios" value="root" />
</jsp:forward>
```

2 Acción `jsp:include`

Esta acción nos permite insertar archivos en una página que está siendo generada. La sintaxis se parece a esto:

```
<jsp:include page="relative URL" flush="true" />
```

Al contrario que la directiva *include*, que inserta el archivo en el momento de la conversión de la página JSP a un *Servlet*, esta acción inserta el archivo en el momento en que la página es solicitada. Esto se paga un poco en la eficiencia, e imposibilita a la página incluida de contener código JSP general (no puede seleccionar cabeceras HTTP, por ejemplo), pero se obtiene una significativa flexibilidad. Por ejemplo, aquí tenemos una página JSP que inserta cuatro puntos diferentes dentro de una página web "What's New?". Cada vez que cambian las líneas de cabeceras, los autores sólo tienen que actualizar los cuatro archivos, pero pueden dejar como estaba la página JSP principal.

WhatsNew.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>What's New</TITLE>
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD>

<BODY BGCOLOR="#FDF5E6" TEXT="#000000" LINK="#0000EE"
      VLINK="#551A8B" ALINK="#FF0000">

<CENTER>
<table BORDER=5 BGCOLOR="#EF8429">
  <TR><TH CLASS="TITLE">
    What's New at JspNews.com</table>
</CENTER>
```

```
<P>
```

```
<OL>
  <LI><jsp:include page="news/Item1.html" flush="true"/>
  <LI><jsp:include page="news/Item2.html" flush="true"/>
  <LI><jsp:include page="news/Item3.html" flush="true"/>
  <LI><jsp:include page="news/Item4.html" flush="true"/>
</OL>
</BODY>
</HTML>
```

Acción jsp:useBean

Esta acción nos permite cargar y utilizar un *JavaBean* en la página JSP. Esta es una capacidad muy útil porque nos permite utilizar la reusabilidad de las clases Java sin sacrificar la conveniencia de añadir JSP sobre *Servlets* solitarios. La sintaxis más simple para especificar que se debería usar un *Bean* es la siguiente:

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

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

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

Observe que la clase especificada por el *bean* debe estar en el path normal del servidor, no en la parte reservada que obtiene la recarga automática cuando se modifican. Por ejemplo, en el *Java web Server*, él y todas las clases que usa deben ir en el directorio *classes* o estar en un archivo JAR en el directorio *lib*, no en el directorio *Servlets*.

Aquí tenemos un ejemplo muy sencillo que carga un *bean* y selecciona y obtiene un sencillo parámetro String.

BeanTest.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Reusing JavaBeans in JSP</TITLE>
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD>

<BODY>

<CENTER>
<table BORDER=5>
  <TR><TH CLASS="TITLE">
    Reusing JavaBeans in JSP</table>
</CENTER>
<P>

<jsp:useBean id="test" class="hall.SimpleBean" />
<jsp:setProperty name="test"
                  property="message"
                  value="Hello WWW" />

<H1>Message: <I>
<jsp:getProperty name="test" property="message" />
</I></H1>

</BODY>
</HTML>
```

SimpleBean.java

Aquí está el código fuente empleado para el *bean* usado en la página *BeanTest*. También, se puede emplear así:

```
package hall;

public class SimpleBean {
  private String message = "No message specified";

  public String getMessage() {
    return(message);
  }
}
```

```

    public void setMessage(String message) {
        this.message = message;
    }
}

```

Aquí tenemos un resultado típico:

Más detalles de jsp:useBean

La forma más sencilla de usar un *bean* es usar

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

para cargar el *bean*, luego usar `jsp:setProperty` y `jsp:getProperty` para modificar y recuperar propiedades del *bean*. Sin embargo, tenemos dos opciones. Primero, podemos usar un formato de contenedor, llamado

```

<jsp:useBean ...>
Body
</jsp:useBean>

```

para indicar que la porción **Body** sólo se debería ejecutar cuando el *bean* es ejemplarizado por primera vez, no cuando un *bean* existente se encuentre y se utilice. Como se explica abajo, los *bean* pueden ser compartidos, por eso no todas las sentencias `jsp:useBean` resultan en la ejemplarización de un *bean*. Segundo, además de *id* y *class*, hay otros tres atributos que podemos usar: *scope*, *type*, y *beanName*.

Atributo	Uso
<i>id</i>	Da un nombre a la variable que referenciará el <i>bean</i> . Se usará un objeto <i>bean</i> anterior en lugar de ejemplarizar uno nuevo si se puede encontrar uno con el mismo <i>id</i> y <i>scope</i> .
<i>class</i>	Designa el nombre completo del paquete del <i>bean</i> .
<i>scope</i>	Indica el contexto en el que el <i>bean</i> debería estar disponible. Hay cuatro posibles valores: <i>page</i> , <i>request</i> , <i>session</i> , y <i>application</i> . El valor por defecto, <i>page</i> , indica que el <i>bean</i> estará sólo disponible para la página actual (almacenado en el <code>PageContext</code> de la página actual). Un valor de <i>request</i> indica que el <i>bean</i> sólo está disponible para la petición actual del cliente (almacenado en el objeto <code>ServletRequest</code>). Un valor de <i>session</i> indica que el objeto está disponible para todas las páginas durante el tiempo de vida de la <code>HttpSession</code> actual. Finalmente, un valor de <i>application</i> indica que está disponible para todas las páginas que compartan el mismo <code>ServletContext</code> . La razón de la importancia del ámbito

es que una entrada `jsp:useBean` sólo resultará en la ejemplarización de un nuevo objeto si no había objetos anteriores con el mismo id y scope. De otra forma, se usarán los objetos existentes, y cualquier elemento `jsp:setParameter` u otras entradas entre las etiquetas de inicio `jsp:useBean` y la etiqueta de final, serán ignoradas.

type Especifica el tipo de la variable a la que se referirá el objeto. Este debe corresponder con el nombre de la clase o ser una superclase o una interfaz que implemente la clase. Recuerde que el **nombre** de la variable se designa mediante el atributo id.

beanName Da el nombre del *bean*, como lo suministraríamos en el método `instantiate` de *beans*. Está permitido suministrar un *type* y un *beanname*, y omitir el atributo *class*.

3 Acción `jsp:setProperty`

Usamos `jsp:setProperty` para obtener valores de propiedades de los *beans* que se han referenciado anteriormente. Podemos hacer esto en dos contextos. Primero, podemos usar antes `jsp:setProperty`, pero fuera de un elemento `jsp:useBean`, de esta forma:

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName"
                 property="someProperty" ... />
```

En este caso, el `jsp:setProperty` se ejecuta sin importar si se ha ejemplarizado un nuevo *bean* o se ha encontrado uno ya existente. Un segundo contexto en el que `jsp:setProperty` puede aparecer dentro del cuerpo de un elemento `jsp:useBean`, de esta forma:

```
<jsp:useBean id="myName" ... >
...
  <jsp:setProperty name="myName"
                  property="someProperty" ... />
</jsp:useBean>
```

Aquí, el `jsp:setProperty` sólo se ejecuta si se ha ejemplarizado un nuevo objeto, no si se encontró uno ya existente.

Aquí tenemos los cuatro atributos posibles de `jsp:setProperty`:

Atributo	Uso
<i>name</i>	Este atributo requerido designa el <i>bean</i> cuya propiedad va a ser seleccionada. El elemento <code>jsp:useBean</code> debe aparecer antes del elemento <code>jsp:setProperty</code> .
<i>property</i>	Este atributo requerido indica la propiedad que queremos seleccionar. Sin

embargo, hay un caso especial: un valor de "*" significa que todos los parámetros de la petición cuyos nombres correspondan con nombres de propiedades del Bean serán pasados a los métodos de selección apropiados.

value Este atributo opcional especifica el valor para la propiedad. Los valores string son convertidos automáticamente a números, boolean, Boolean, byte, Byte, char, y Character mediante el método estándar `valueOf` en la fuente o la clase envolvente. Por ejemplo, un valor de "true" para una propiedad boolean o Boolean será convertido mediante `Boolean.valueOf`, y un valor de "42" para una propiedad int o Integer será convertido con `Integer.valueOf`. No podemos usar *value* y *param* juntos, pero sí está permitido no usar ninguna.

Este parámetro opcional designa el parámetro de la petición del que se debería derivar la propiedad. Si la petición actual no tiene dicho parámetro, no se hace nada: el sistema no pasa null al método seleccionador de la propiedad. Así, podemos dejar que el *bean* suministre los valores por defecto, sobrescribiéndolos sólo cuando el parámetro dice que lo haga. Por ejemplo, el siguiente código dice "selecciona el valor de la propiedad `numberOfItems` a cualquier valor que tenga el parámetro `numItems` de la petición, si existe dicho parámetro; si no existe, no se hace nada"

param `<jsp:setProperty name="orderBean"`
 `property="numberOfItems"`
 `param="numItems" />`

Si omitimos tanto *value* como *param*, es lo mismo que si suministramos un nombre de parámetro que corresponde con el nombre de una propiedad. Podremos tomar esta idea de automaticidad usando el parámetro de la petición cuyo nombre corresponde con la propiedad suministrada un nombre de propiedad de "*" y omitir tanto *value* como *param*. En este caso, el servidor itera sobre las propiedades disponibles y los parámetros de la petición, correspondiendo aquellas con nombres idénticos.

Aquí tenemos un ejemplo que usa un *bean* para crear una tabla de números primos. Si hay un parámetro llamado `numDigits` en los datos de la petición, se pasa dentro del *bean* a la propiedad `numDigits`. Al igual que en `numPrimes`.

JspPrimes.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Reusing JavaBeans in JSP</TITLE>
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD>

<BODY>

<CENTER>
<table BORDER=5>
  <TR><TH CLASS="TITLE">
```



```

        Reusing JavaBeans in JSP</table>
</CENTER>
<P>

<jsp:useBean id="primetable" class="hall.NumberedPrimes" />
<jsp:setProperty name="primetable" property="numDigits" />
<jsp:setProperty name="primetable" property="numPrimes" />

Some <jsp:getProperty name="primetable" property="numDigits" />
digit primes:
<jsp:getProperty name="primetable" property="numberedList" />

</BODY>
</HTML>

```

Acción jsp:getProperty

Este elemento recupera el valor de una propiedad del *bean*, lo convierte a un string e inserta el valor en la salida. Los dos atributos requeridos son *name*, el nombre de un *bean* referenciado anteriormente mediante `jsp:useBean`, y *property*, la propiedad cuyo valor debería ser insertado. Aquí tenemos un ejemplo:

```

<jsp:useBean id="itemBean" ... />
...
<UL>
    <LI>Number of items:
        <jsp:getProperty name="itemBean" property="numItems" />
    <LI>Cost of each:
        <jsp:getProperty name="itemBean" property="unitCost" />
</UL>

```

Acción jsp:forward

Esta acción nos permite reenviar la petición a otra página. Tiene un sólo atributo, *page*, que debería consistir en una URL relativa. Este podría ser un valor estático o podría ser calculado en el momento de la petición, como en estos dos ejemplos:

```

<jsp:forward page="/utils/errorReporter.jsp" />
<jsp:forward page="<%= someJavaExpression %>" />

```

Acción jsp:plugin

Esta acción nos permite insertar un elemento *OBJECT* o *EMBED* específico del navegador para especificar que el navegador debería ejecutar un *applet* usando el Plug-in Java.

Preguntas de Certificación

1. ¿Cuál de las siguientes sentencias es válida para usar la acción <jsp:useBean>? (Seleccione una opción.)
 - a. <jsp:useBean id="address" class="AddressBean" />
 - b. <jsp:useBean name="address" class="AddressBean"/>
 - c. <jsp:useBean bean="address" class="AddressBean" />
 - d. <jsp:useBean beanName="address" class="AddressBean" />

2. ¿Cuál de las siguientes sentencias es el camino válido para obtener el atributo de los beans? (Seleccione una opción.)
 - a. <jsp:useBean action="get" id="address" property="city" />
 - b. <jsp:getProperty id="address" property="city" />
 - c. <jsp:getProperty name="address" property="city" />
 - d. <jsp:getProperty bean="address" property="*" />

3. ¿Cuáles de los siguientes son usos válidos de la acción <jsp:useBean>? (Seleccione dos opciones.)
 - a. <jsp:useBean id="address" class="AddressBean" name="address" />
 - b. <jsp:useBean id="address" class="AddressBean" type="AddressBean" />
 - c. <jsp:useBean id="address" beanName="AddressBean" class="AddressBean" />
 - d. <jsp:useBean id="address" beanName="AddressBean" type="AddressBean" />

4. ¿Cuál de los siguientes *gets* or *sets* del bean es válido en el contenedor *ServletContext*? (Seleccione una opción.)
 - a. <jsp:useBean id="address" class="AddressBean" />
 - b. <jsp:useBean id="address" class="AddressBean" scope="application" />
 - c. <jsp:useBean id="address" class="AddressBean" scope="servlet" />
 - d. <jsp:useBean id="address" class="AddressBean" scope="session" />
 - e. Ninguna de las anteriores

5. Considere el siguiente código:


```
<html><body>
<jsp:useBean id="address" class="AddressBean" scope="session" />
state = <jsp:getProperty name="address" property="state" />
</body></html>
```

¿Cuáles de las siguientes líneas es válida para remplazar la primera línea de código *Java*? (Seleccione tres líneas.)

 - a. <% state = address.getState(); %>
 - b. <% out.write("state = "); out.print(address.getState()); %>
 - c. <% out.write("state = "); out.print(address.getstate()); %>
 - d. <% out.print("state = " + address.getState()); %>
 - e. state = <%= address.getState() %>
 - f. state = <%! address.getState(); %>

Resumen

 Aquí está el código fuente usado para el *Bean* usado en la página *BeanTest*.

```
package hall;


public class SimpleBean {
    private String message = "No message specified";

    public String getMessage() {
        return(message);
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

La forma más sencilla de usar un Bean es usar lo siguiente:

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

 Si desea saber más acerca de estos temas, puede consultar la siguiente página.

 <http://www.java2s.com/Code/Java/JSP/JSPStandardActionssetProperty.htm>

Aquí hallará teoría sobre JSP *Standard Actions*.

UNIDAD DE
APRENDIZAJE

4

SEMANA

11-12

CUSTOM TAG

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos crearán componentes mediante etiquetas personalizadas en la capa vista, que permitan mostrar de manera reutilizable la presentación de la información.

TEMARIO

- *Custom Tag*

ACTIVIDADES PROPUESTAS

- Los alumnos crearán aplicaciones que permitan utilizar los tag personalizados.

1 CustomTag

La tecnología JSP nos permite introducir código *Java* en páginas HTML. Este documento supone que está familiarizado con la creación de páginas JSP y que tiene instalado *Apache-Tomcat* (en realidad, valdría cualquier servidor de JSP que cumpla la especificación JSP 1.1, pero los ejemplos se han probado en *Tomcat*).

Las JSP están compuestas por varios **tags** o, como las denominaremos en adelante, **etiquetas**. Ejemplos de etiquetas JSP son ***setProperty***, ***include***, ***forward***, ***usebean***. Todas ellas encapsulan comportamientos de una manera sencilla. Nos detendremos en la etiqueta `<jsp:useBean>`. Esta etiqueta crea una *instancia* de una clase definida como un *JavaBean*. El uso de la etiqueta `<jsp:useBean>` en conjunción con los *JavaBeans* hace que podamos separar la lógica de presentación de la de proceso y consigue unos archivos JSP más claros, y lo que es más importante, el que escribe la JSP no tiene por qué tener unos conocimientos avanzados de programación para crear páginas JSP de funcionalidades avanzadas. Pues bien, las *JSP Custom Tags* son un paso más allá. Con ellas se pueden añadir nuevas etiquetas a las ya existentes en JSP (eso sí a partir de la especificación JSP 1.1) que encapsulen comportamientos más o menos complicados. Antes de seguir, vamos a revisar algunos conceptos.

Hay dos tipos principales de etiquetas: etiquetas que no tienen cuerpo y etiquetas que sí lo tienen. Ejemplos son la etiqueta `` que no tiene cuerpo y no tiene etiqueta de final; y `<title>manolo</title>` que tiene cuerpo y etiqueta de final. Estos son también los dos tipos de etiquetas JSP que nos podemos crear a medida.

1.1 La primera etiqueta

Las *JSP Custom Tags* no son más que clases *Java* que implementan o heredan de determinadas clases. Dependiendo de si la etiqueta tendrá cuerpo o no, se debe implementar una interfaz u otra, o como veremos más adelante, se debe heredar de una u otra clase.

Vamos a definir nuestra primera etiqueta personalizada, que como se ha hecho desde el principio de los tiempos será un Hola mundo, muy original :). Nuestra etiqueta va a ser una etiqueta sin cuerpo de la forma `<ejemplo:hola/>`. Atención a la barra invertida del final. Recordemos que JSP sigue una sintaxis XML y las etiquetas que no tienen cierre deben llevar esta barra. Como es una etiqueta sin cuerpo, se debe implementar la interfaz Tag:

```
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;

    public int doEndTag() throws JspException;
    public int doStartTag() throws JspException;
    public Tag getParent();
    public void release();
}
```

```
public void setPageContext(PageContext pc);
public Tag setParent(Tag t);
```

setPageContext: Este método lo invoca el motor de JSP para definir el contexto.

setParent: Lo invoca el motor de JSP para pasarle un manejador de etiqueta a la etiqueta padre.

getParent: Devuelve una *instancia* de la etiqueta padre.

doStartTag: Se procesa la etiqueta de apertura.

doEndTag: Se ejecuta despues de doStartTag.

release: Lo invoca el motor de JSP para hacer limpieza.

La etiqueta deberá implementar todos los métodos y ejecutará la complicada y novedosa tarea de mostrarnos un Hola mundo. Crearemos ahora lo que se llama un Tag Handler. Primero el código y luego comentamos:

```
// HolaMundoTag.java
import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HolaMundoTag implements Tag {
    private PageContext pc;
    private Tag parent;

    public HolaMundoTag() {
        super();
    }

    public int doStartTag() throws JspException {
        try {
            pc.getOut().print("Hola mundo");
        } catch (IOException e){
            throw new JspException("Error al enviar al cliente"
+
                                                                e.getMessage());
        }
        return SKIP_BODY;
    }

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

    public void release(){
```

```

    }

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

    public void setParent(Tag parent){
        this.parent = parent;
    }

    public Tag getParent(){
        return parent;
    }

```

HolaMundoTag.java

Vemos que la funcionalidad de la etiqueta es mínima, la cantidad de métodos a implementar es bastante grande y la mayoría de ellos triviales. La especificación JSP 1.1 nos provee de una clase que implementa una *Custom Tag* con la funcionalidad mínima y de la que podemos heredar para sobrescribir los métodos que necesitemos. Esta clase es `TagSupport`, el código anterior se puede reescribir de la siguiente manera:

```

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

public class HolaMundoTag extends TagSupport{
    public int doStartTag() throws JspException {
        try{
            pageContext.getOut().print("Hola Mundo");
        } catch (IOException e) {
            throw new JspException ("Error: IOException" +
                                    e.getMessage());
        }
        return SKIP_BODY;
    }

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

```

HolaMundoTag.java

Se necesita importar `java.io.*` para escribir en el cliente. Los otros dos import son los típicos para una JSP *Custom Tag*. El método `doStartTag` se invoca cuando se encuentra la etiqueta de apertura, el código tiene que ir incluido en un try-catch porque el método `print()` puede generar una `IOException`, al final se devuelve el valor

SKIP_BODY que le indica al motor de JSP que la etiqueta no tiene cuerpo (en realidad que no queremos evaluar el cuerpo). El método doEndtag se invoca cuando se encuentra la etiqueta de cierre. En este ejemplo, se devuelve EVAL_PAGE para que siga evaluando el resto de la página. Si no queremos evaluar el resto de la página, el valor por devolver será SKIP_PAGE.

Para compilar debemos incluir en el classpath los paquetes de JSP, suponiendo que *Tomcat* está instalado en c:\tomcat:

```
javac -classpath c:\tomcat\lib\common\servlet.jar HolaMundoTag.java
```

Ya tenemos la etiqueta compilada. Ahora tenemos que decirle a *Tomcat* cómo y cuándo utilizarla. Para esto tenemos que crear un archivo que describa a esta etiqueta, en concreto que describa una librería de etiquetas. Este archivo se llama Tag Library Descriptor y sigue una sintaxis xml:

```
<!-- Ejemplos.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/JEE/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>ejemplos</shortname>
    <uri></uri>
    <info>Etiquetas de ejemplo</info>

    <tag>
        <name>holamundo</name>
        <tagclass>HolaMundoTag</tagclass>
        <bodycontent>empty</bodycontent>
        <info>Saludo</info>
    </tag>
</taglib>
```

Ejemplos.tld

El documento empieza con la etiqueta que declara que es un documento xml y la declaración DOCTYPE. Luego, el resto estará encerrado entre <taglib> y </taglib>. Las etiquetas <tlibversion> y <jspversion> definen las versiones de la librería de etiquetas y de la especificación JSP, respectivamente. Vemos que cada etiqueta va encerrada entre <tag> y </tag> y lleva el nombre de la etiqueta (<name>), la clase que implementa la etiqueta (<tagclass>), si tiene cuerpo (<bodycontent>, en este caso empty para indicar que no lleva) y la etiqueta <info> que almacena una descripción de la etiqueta.

Ya tenemos la clase que implementa la etiqueta (que por cierto, para probarla, la guardaremos en c:\tomcat\webapps\ejemplos\WEB-INF\classes\HolaMundoTag.class, suponiendo que el *Tomcat* está en c:\tomcat), el *tag library descriptor* (que

guardaremos en `c:\tomcat\webapps\ejemplos\Ejemplos.tld`) y necesitamos una página JSP para probar nuestra primera etiqueta personalizada; por ejemplo, esta:

```
<!-- HolaMundo.jsp -->
<%@ taglib uri="Ejemplos.tld" prefix="ejemplos" %>
<HTML>
<HEAD>
    <TITLE>Tag Hola Mundo</TITLE>
</HEAD>
<ejemplos:holamundo/>
</BODY>
</HTML>
```

HolaMundo.jsp

Vemos que es una JSP normal a excepción de nuestra nueva etiqueta. Obsérvese que no tiene etiqueta de cierre porque no tiene cuerpo. En la directiva `taglib` definimos dónde se encuentra la librería de etiquetas y el nombre por el que nos referiremos a esta librería durante todo el documento.

Si situamos el archivo `HolaMundo.jsp` en `c:\tomcat\webapps\ejemplos\HolaMundo.jsp`, arrancamos el *Tomcat* y visitando la dirección `http://localhost:8080/ejemplos/HolaMundo.jsp` (sustituyendo el *host* y el puerto de acuerdo con nuestra configuración) veremos nuestra primera etiqueta en acción.



1.2 Etiquetas con parámetros

La etiqueta anterior no tenía ningún parámetro. Hay situaciones en las que nos interesa llamar a una etiqueta con parámetros, lo mismo que haríamos con una función. Ahora vamos a hacer una etiqueta que sume los dos números que le pasemos como parámetros; por ejemplo, para sumar $2+3$ la etiqueta será `<ejemplos:suma num1=2 num2=3/>`.

Por cada parámetro que queremos que soporte la etiqueta, debemos añadir al código de la etiqueta anterior un método `set` del mismo modo que los métodos `set` de los *javabeans*, `set` + el nombre del parámetro. Para esta etiqueta, el código es el siguiente:

```
// SumaTag.java
import java.io.*;
```

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

public class SumaTag 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("Suma: " + (num1+num2));
        } catch (IOException e) {
            throw new JspException ("Error: IOException" +
e.getMessage());
        }
        return SKIP_BODY;
    }

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

SumaTag.java

Tenemos que añadir la nueva etiqueta, a la librería de ejemplos, el archivo Ejemplos.tld. El esqueleto es el mismo que el anterior, solo que ahora tenemos que añadir una serie de etiquetas para cada atributo. Por cada atributo, le diremos su nombre, con la etiqueta <name>, si es necesaria o no, con la etiqueta <required> y si tiene que ser averiguada en tiempo de ejecución con la etiqueta <rtexprvalue>. En nuestro caso, tenemos que añadir esto a Ejemplos.tld entre <taglib> y </taglib>:

```
<tag>
    <name>suma</name>
    <tagclass>SumaTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Saludo</info>
    <attribute>
        <name>num1</name>
        <required>true</required>
        <rtexprvalue>>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>num2</name>
        <required>true</required>
        <rtexprvalue>>false</rtexprvalue>
    </attribute>
```

```
</tag>
```

Ejemplos.tld

El código de la jsp es igual de complicado que el de la etiqueta anterior:

```
<%@ taglib uri="Ejemplos.tld" prefix="ejemplos" %>
<HTML>
<HEAD>
  <TITLE>Tag Suma</TITLE>
</HEAD>
<ejemplos:suma num1="2" num2="3" />
</BODY>
</HTML>
```

Ejemplos.tld

Si colocamos los archivos de la misma forma que antes (c:\tomcat\webapps\ejemplos\Ejemplos.tld, c:\tomcat\webapps\ejemplos\Suma.jsp, c:\tomcat\webapps\ejemplos\classes\SumaTag.class) y visitamos <http://localhost:8080/ejemplos/Suma.jsp>, veremos cómo funciona nuestra segunda etiqueta personalizada.



1.3 Etiquetas con cuerpo

Las etiquetas con cuerpo tienen la misma estructura que etiquetas de html del estilo de `<h1>titulo</h1>`. Son etiquetas con etiqueta de apertura y de cierre con algo dentro, que llamamos cuerpo. Para implementar una etiqueta personalizada con cuerpo, debemos heredar de la clase `BodyTagSupport` (o implementar la interfaz `BodyTag`) y sobrescribir los métodos necesarios. Vamos a hacer una etiqueta que nos diga el tamaño de la cadena que se le pasa en el cuerpo.

El código sobrescribe el método `doAfterBody()` que se ejecuta después de comprobar el contenido del cuerpo de la etiqueta:

```
// LongitudTag.java
```

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

public class LongitudTag extends BodyTagSupport{
    public int doAfterBody() throws JspException {
        try {
            BodyContent bc = getBodyContent();
            String cuerpo = bc.getString();
            JspWriter out = bc.getEnclosingWriter();
            if (cuerpo != null){
                out.print("la longitud de " + cuerpo + "es: " +
                        cuerpo.length());
            }
        } catch (IOException e){
            throw new JspException("Error: IOEXception" +
e.getMessage());
        }
        return SKIP_BODY;
    }
}
```

LongitudTag.java

Primero, tomamos el contenido del cuerpo y lo almacenamos en bc. Lo tratamos como cadena y lo guardamos en cuerpo y obtenemos un canal de salida para escribir en el cliente con getEnclosingWriter(). Compilamos LogitudTag.

Ahora tenemos que incluir la nueva etiqueta en nuestra librería:

```
<tag>
    <name>longitud</name>
    <tagclass>LongitudTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>calcula longitud de una cadena</info>
</tag>
```

Ejemplos.tld

La nueva etiqueta <bodycontent> le indica al motor de jsp cuál va a ser el contenido del cuerpo. Si el contenido es texto, html u otras etiquetas, se indica JSP. Otros tipos de contenidos son tagdependent. Esto no afecta a la interpretación de la etiqueta, solamente es informativo.

La jsp de prueba es tanto o más complicada que las anteriores:

```
<%@ taglib uri="Ejemplos.tld" prefix="ejemplos" %>
```

```

<HTML>
<HEAD>
  <TITLE>Tag Longitud</TITLE>
</HEAD>
<ejemplos:longitud>
  Hola, soy una cadena
</ejemplos:longitud>
</BODY>
</HTML>

```

Longitud.jsp

Como siempre, se colocan los archivos en su sitio, se ejecuta el *Tomcat* y se visita <http://localhost:8080/ejemplos/Longitud.jsp>



Hay un caso concreto de etiquetas con cuerpo, cuando se necesita que se evalúe el cuerpo más de una vez. Esto se consigue devolviendo `EVAL_BODY_TAG` en lugar de `SKIP_BODY` en el método `doAfterBody()`.

Como ejemplo, vamos a programar una etiqueta personalizada que realice un bucle, que escriba un número determinado de veces el cuerpo de la etiqueta.

El código de la clase de la etiqueta bucle es el siguiente:

```

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

public class BucleTag extends BodyTagSupport{
    int veces = 0;
    BodyContent bc;

    public void setVeces(int veces) {
        this.veces = veces;
    }
    public void setBodyContent(BodyContent bc){
        this.bc = bc;
    }
    public int doAfterBody() throws JspException {

```

```

        if (veces-->0){
            try {
                JspWriter out = bc.getEnclosingWriter();
                out.println(bc.getString());
                bc.clearBody();
            } catch (IOException e) {
                system.out.println("Error en Tag Bucle" +
e.getMessage());
            }
            return EVAL_BODY_TAG;
        } else {
            return SKIP_BODY;
        }
    }
}

```

BucleTag.java

Debemos añadir la descripción de esta nueva etiqueta a la librería:

```

<tag>
    <name>bucle</name>
    <tagclass>BucleTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>realiza un bucle</info>
    <attribute>
        <name>veces</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>

```

Ejemplos.tld

Como es la última etiqueta de esta introducción, la JSP va a ser la más complicada XD.

```

<!-- Bucle.jsp-->
<%@ taglib uri="Ejemplos.tld" prefix="ejemplos" %>
<HTML>
<HEAD>
    <TITLE>Tag Bucle</TITLE>
</HEAD>
<ejemplos:bucle veces="5">
Dentro del bucle<br>
</ejemplos:bucle>
</BODY>
</HTML>

```



Bucle.jsp

Después de colocar los archivos en sus lugares habituales, lanzamos el *Tomcat* y visitamos <http://localhost:8080/ejemplos/Bucle.jsp>.

Resumen

 La directiva taglib permite el registro de Tag Library Descriptor (TLD)

```
<%@ taglib uri="aliasDelTLD" prefix="ejemplos" %>
```

 Métodos de la interfaz Tag

setPageContext: Este método lo invoca el motor de JSP para definir el contexto.

setParent: Lo invoca el motor de JSP para pasarle un manejador de etiqueta a la etiqueta padre.

getParent: Devuelve una *instancia* de la etiqueta padre.

doStartTag: Se procesa la etiqueta de apertura.

doEndTag: Se ejecuta después de doStartTag.

release: Lo invoca el motor de JSP para hacer limpieza.

 Si desea saber más acerca de estos temas, puede consultar la siguiente página.

 http://java.sun.com/JEE/tutorial/1_3-fcs/doc/JSPTags.html

Aquí hallará la descripción sobre *CustomTag*.

**UNIDAD DE
APRENDIZAJE****5****SEMANA****13-14**

JAVA STANDARD TAG LIBRARY

LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos crearán aplicaciones utilizando en los JSP las tecnologías de Lenguaje de Expresiones, JavaServer Pages Standard Tag Library y displayTag en la paginación.

TEMARIO

- JSTL
- Objetos Implícitos

ACTIVIDADES PROPUESTAS

- Los alumnos crearán aplicaciones que permitan utilizar los tag estándares.

1 Java Standard Tag Library (JSTL)

JSTL responde a la demanda de los desarrolladores de un conjunto de acciones JSP personalizadas para manejar las tareas que necesitan casi todas las páginas JSP, incluyendo procesamiento condicional, internacionalización, acceso a bases de datos y procesamiento XML. Esto acelerará el desarrollo de JSPs más o menos eliminando la necesidad de elementos de scripting, y los inevitables y difíciles de encontrar errores de sintaxis; y liberando el tiempo anteriormente gastado en desarrollar y aprender trillones de acciones personalizadas específicas del proyecto para estas tareas comunes.

1.1 Introducción a las Librerías JSTL

JSTL 1.0 especifica un conjunto de librerías de etiquetas basadas en el API JSP 1.2. Hay cuatro librerías de etiquetas independientes, cada una contiene acciones personalizadas dirigidas a un área funcional específica. Esta tabla lista cada librería con su prefijo de etiqueta recomendado y la URI por defecto:

Descripción	Prefijo	URI por Defecto
Core	c	http://java.sun.com/jstl/core
XML Processing	x	http://java.sun.com/jstl/xml
I18N & Formatting	fmt	http://java.sun.com/jstl/fmt
Database Access	sql	http://java.sun.com/jstl/sql

La *Librería Core* contiene acciones para las tareas rutinarias, como incluir o excluir una parte de una página dependiendo de una condición en tiempo de ejecución, hacer un bucle sobre una colección de ítems, manipular URLs para seguimiento de sesión, y la correcta interpretación del recurso objetivo. Asimismo, tiene acciones para importar contenido de otros recursos y redireccionar la respuesta a una URL diferente.

La *Librería XML* contiene acciones para procesamiento XML, incluido validar un documento XML y transformarlo usando XSLT. También, proporciona acciones para extraer parte de un documento XML validado, hacer bucles sobre un conjunto de nodos y procesamiento condicional basado en valores de nodos.

La Internacionalización (i18n) y el formateo general están soportados por las acciones de la *Librería I18N & Formatting*. Podemos leer y modificar información almacenada en una base de datos con las acciones proporcionadas por la *Librería Database Access*.

Más adelante, podemos esperar que todos los contenedores web incluyan una implementación de las librerías JSTL, así no tendremos que instalar ningún código adicional. Hasta que esto suceda, podemos descargar e instalar la implementación de referencia (RI) de JSTL. Se ha desarrollado dentro del proyecto **Apache Taglibs** como una librería llamada *Standard*.

Instalar RI es fácil, sólo hay que copiar los archivos JAR del directorio lib de la distribución al directorio WEB-INF/lib de nuestra aplicación. Obsérvese que JSTL 1.0 requiere un contenedor JSP 1.2; por ello, debemos asegurarnos de tener un contenedor compatible con JSP 1.2 antes de probar esto.

Para usar una librería JSTL, tanto si la implementación está incluida con el contenedor o con el RI, debemos declarar la librería usando una directiva taglib, como lo haríamos con una librería de etiquetas personalizadas normal:

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

Obsérvese que siempre deberíamos usar la URI por defecto, incluso aunque la especificación JSP nos permita sobreescribirla. Un contenedor puede generar código optimizado para la acción JSTL en la clase correspondiente a una página JSP. Esto puede resultar en un mejor rendimiento que cuando el código generado llama a los manejadores de etiquetas a través del API estándar. Sin embargo, sólo cuando se usa la URI por defecto, es cuando el contenedor puede utilizar una implementación optimizada.

1.2 El Lenguaje de Expresión JSTL

Además de las librerías de etiquetas, JSTL 1.0 define un llamado *Lenguaje de Expresiones* (EL). **EL** es un lenguaje para acceder a datos de varias fuentes en tiempo de ejecución. Su sintaxis es considerablemente más amigable que la de *Java*, que es el único lenguaje soportado directamente por la especificación JSP 1.2. Todas las acciones JSTL reconocen expresiones **EL** en sus valores de atributos, y se podrían desarrollar acciones personalizadas para que hicieran lo mismo. Se espera que **EL** sea incorporado dentro de la próxima versión de la especificación JSP para mejorar su uso para acceder a datos sobre el lenguaje Java. Si es así, podremos usar expresiones **EL** en un valor de atributo de una acción, o incluso en una plantilla de texto.

EL toma prestada de *JavaScript* la sintaxis para acceder a estructuras de datos tanto como *propiedades* de un objeto (con el operador `.`) como con *elementos con nombres de un array* (con el operador `["nombre"]`). Las propiedades de los componentes *JavaBeans* y las entradas `java.util.Map`, que usan la clave como nombre de propiedad, pueden ser accedidas de esta forma. Aquí tenemos algunos ejemplos:

```
${myObj.myProperty}$
${myObj["myProperty"]}
${myObj[varWithTheName]}$
```

Como se ve aquí, una expresión **EL** siempre debe estar encerrada entre los caracteres `${` y `}`. Las dos primeras expresiones acceden a una propiedad llamada `myProperty` en un objeto representado por una variable llamada `myObj`. La tercera expresión accede a una propiedad con un nombre contenido en una variable. Esta sintaxis se puede utilizar con cualquier expresión que evalúe el nombre de la propiedad.

El operador de acceso a array también se usa para datos representados como una colección de *elementos indexados*, como un *array Java* o una `java.util.List`:

```
${myList[2]}$
```

```
${myList[aVar + 1]}$
```

Además de los operadores de propiedad y elemento array, y los operadores aritméticos, relacionales y lógicos, hay un operador especial para comprobar si un objeto está "vacío" o no puede ser usado en una expresión **EL**. La siguiente tabla lista todos los operadores:

Operador	Descripción
.	Accede a una propiedad.
[]	Accede a un elemento de un array/lista.
()	Agrupar una subexpresión.
+	Suma
-	Resta o negación de un número
/ o div	División
% o mod	Módulo (resto)
== o eq	Comprueba Igualdad.
!= o ne	Comprueba desigualdad.
< o lt	Comprueba menor que.
> o gt	Comprueba mayor que.
<= o le	Comprueba menor o igual que.
>= o ge	Comprueba mayor o igual que.
&& o and	Comprueba AND lógico.
o or	Comprueba OR lógico.
! o not	Complemento binario booleano
empty	Comprueba un valor vacío (null, string vacío o una colección vacía)

Lo que no encontraremos en **EL** son sentencias como asignaciones, if/else o while. Para este tipo de funcionalidades, en JSP se usan los elementos **Action**. **EL** no está pensado para utilizarse como un lenguaje de programación de propósito general, sólo un lenguaje de acceso a datos.

Por supuesto, los literales y las variables también son parte del lenguaje. **EL** proporciona los siguientes literales, similares a los que proporcionan *JavaScript*, *Java*, y otros lenguajes similares:

Tipo de Literal	Descripción
String	Encerrado con comillas simples o dobles. Una comilla del mismo tipo dentro del string puede ser escapada con una barra invertida: (\' en un string encerrado con comillas simples; \" en un string encerrado con comillas dobles). El carácter de barra invertida debe ser escapado como \\ en ambos casos.

Integer	Un signo opcional (+ o -) seguido por dígitos entre 0 y 9.
Coma Floating	Lo mismo que un literal entero, excepto que usa un punto como separador de la parte fraccional y que se puede especificar un exponente con e o E, seguido por un literal entero.
Boolean	true o false
Null	null

Cualquier objeto en uno de los ámbitos de JSP (página, solicitud, sesión o aplicación) se puede utilizar como una variable en una expresión **EL**. Por ejemplo, si tenemos un *bean* con una propiedad `firstName` en el ámbito de la solicitud bajo el nombre `customer`, esta expresión **EL** representa el valor de la propiedad `firstName` del *bean*.

```
${customer.firstName}
```

Sin embargo, **EL** también hace que la información de la solicitud y la información general del contenedor esté disponible como un conjunto de variables implícitas:

Variable	Descripción
<code>param</code>	Una <i>collection</i> de todos de los parámetros de la solicitud como un sólo valor string para cada parámetro
<code>paramValues</code>	Una <i>collection</i> de todos los valores de los parámetros de la solicitud como un array de valores string por cada parámetro
<code>header</code>	Una <i>collection</i> de todas las cabeceras de solicitud como un sólo valor string por cada cabecera
<code>headerValues</code>	Una <i>collection</i> de todos los valores de cabecera de la solicitud como un array de valores string por cada cabecera
<code>cookie</code>	Una <i>collection</i> con todas las cookies de la solicitud en un sólo ejemplar de <code>javax.servlet.http.Cookie</code> por cada cookie
<code>initParams</code>	Una <i>collection</i> de todos los parámetros de inicialización de la aplicación en un sólo valor string por cada parámetro
<code>pageContext</code>	Un ejemplar de la clase <code>javax.servlet.jspPageContext</code>
<code>pageScope</code>	Una <i>collection</i> de todos los objetos en el ámbito de la página
<code>requestScope</code>	Una <i>collection</i> de todos los objetos en el ámbito de la solicitud
<code>sessionScope</code>	Una <i>collection</i> de todos los objetos en el ámbito de la sesión
<code>applicationScope</code>	Una <i>collection</i> de todos los objetos en el ámbito de la aplicación

Las cinco primeras variables implícitas de la tabla nos ofrecen acceso a los valores de parámetros, cabeceras y cookies de la solicitud actual. Aquí hay un ejemplo de cómo acceder a un parámetro de solicitud llamado `listType` y a la cabecera `User-Agent`:

```
${param.listType}
${header['User-Agent']}
```

Observe cómo se debe usar la sintaxis de array para la cabecera, porque el nombre incluye un guion; con la sintaxis de propiedad, sería interpretado como la expresión `vairable header.User` menos el valor de una variable llamada `Agent`.

La variable `initParameter` proporciona acceso a los parámetros de inicialización que se definen en el archivo `web.xml` de la aplicación. La variable `pageContext` tiene varias propiedades que proporcionan acceso al objeto `servlet` que representa la solicitud, la respuesta, la sesión y la aplicación, etc.

Las cuatro últimas variables son colecciones que contienen todos los objetos de cada ámbito específico. Podemos usarlas para limitar la búsqueda de un objeto en sólo un ámbito en lugar de buscar en todos ellos, lo que está por defecto si no se especifica ningún ámbito. En otras palabras, si hay un objeto llamado `customer` en el ámbito de sesión, estas dos primeras expresiones encuentran el mismo objeto, pero la tercera vuelve vacía:

```
${customer}
${sessionScope.customer}
${requestScope.customer}
```

Todas las acciones JSTL aceptan expresiones **EL** como valores de atributo, para todos los atributos excepto para `var` y `scope`, porque estos valores de atributo podrían usarse para chequear el tipo en el momento de la traducción en una futura versión. Hay un atributo de una acción JSTL adicional que no toma un valor de expresión **EL**, pero sólo se usa en la librería XML, por eso lo ignoraremos por ahora. Se pueden usar una o más expresiones **EL** en el mismo valor de atributo, y el texto fijo y las expresiones **EL** se pueden mezclar en el mismo valor de atributo:

```
First name: <lt;c:out value="${customer.firstName}" />>
<lt;c:out value="First name: ${customer.firstName}" />>
```

Antes de saltar a ver ejemplos de utilización de las acciones **Core**, se verá lo que dijimos anteriormente: todas las acciones JSTL de la *librería EL* aceptan expresiones EL. Realmente hay un conjunto paralelo de librerías JSTL, llamado *conjunto de librería RT* que sólo acepta expresiones *Java* del antiguo estilo:

```
First name: <c_rt:out value="<%= customer.getFirstName() %>" />
```

1.3 Procesamiento Condicional y *Bucles*

Veamos algunos ejemplos de cómo podemos usar el JSTL condicional y las acciones de iteración: `<c:if>`; `<c:choose>`, `<c:when>`, y un triple `<c:otherwise>`; y `<c:forEach>`. Por el camino, también usaremos acciones de salida básica y de selección de variables: `<c:out>` y `<c:set>`.

`<c:if>` nos permite incluir, o procesar, condicionalmente una parte de una página, dependiendo de la información durante la ejecución. Este ejemplo incluye un saludo personal si el usuario es un visitante repetitivo, según lo indica la presencia de una *cookie* con el nombre del usuario:

```
<c:if test="${!empty cookie.userName}">
    Welcome back <c:out value="${cookie.userName.value}" />
</c:if>
```

El valor del atributo `test` es una expresión **EL** que chequea si la *cookie* existe. El operador `empty` combinado con el operador "not" (!) significa que evalúa a `true` si el *cookie* no existe, haciendo que el cuerpo del elemento sea procesado. Dentro del cuerpo, la acción `<c:out>` añade el valor de la *cookie* a la respuesta. Así de sencillo.

Pasar a través de una colección de datos es casi tan sencillo. Este fragmento itera sobre una colección de filas de una base de datos con información del tiempo de diferentes ciudades:

```
<c:forEach items="${forecasts.rows}" var="${city}">
    City: <c:out value="${city.name}" />
    Tomorrow's high: <c:out value="${city.high}" />
    Tomorrow's low: <c:out value="${city.low}" />
</c:forEach>
```

La expresión **EL** para el valor `items` obtiene el valor de la propiedad `rows` desde un objeto representado por la variable `forecasts`. Como aprenderemos más adelante, las acciones de bases de datos de JSTL representan un resultado de consulta como un ejemplar de una clase llamada `javax.servlet.jsp.jstl.sql.Result`. Esta clase se puede utilizar como un *bean* con varias propiedades. La propiedad `rows` contiene un array de ejemplares `java.util.SortedMap`, donde cada uno representa una fila con valores de columnas. La acción `<c:forEach>` procesa su cuerpo una vez por cada elemento de la colección especificado por el atributo `items`. Además de con arrays, la acción funciona con cualquier otro tipo de dato que represente una colección, como ejemplares de las clases `java.util.Collection` o `java.util.Map`.

Si se especifica el atributo `var`, el elemento actual de la colección se hace disponible para las acciones del cuerpo como una variable con el nombre especificado. Aquí se llamaba `city` y, como la colección es un array de maps, esta variable contiene un nuevo map con valores de columnas cada vez que se procesa el cuerpo. Los valores de las columnas se añaden a la respuesta por el mismo tipo de acciones `<c:out>` que hemos visto en ejemplos anteriores.

Para ilustrar el uso del resto de las acciones condicionales, extendamos el ejemplo de iteración para procesar sólo un conjunto fijo de filas por cada página solicitada, añadamos enlaces "Previous" y "Next" a la misma página. El usuario puede entonces

pasar sobre los resultados de la base de datos, mirando unas pocas filas cada vez, asumiendo que el objeto Result se ha grabado en el ámbito de la sesión. Aquí está cómo procesar sólo algunas filas:

```
<c:set var="noOfRows" value="10" />

<c:forEach items="${forecasts.rows}" var="${city}"
  begin="${param.first}" end="${param.first + noOfRows - 1}">
  City: <c:out value="${city.name}" />
  Tomorrow's high: <c:out value="${city.high}" />
  Tomorrow's low: <c:out value="${city.low}" />
</c:forEach>
```

La acción `<c:set>` selecciona una variable con el valor especificado por el atributo `value`; que puede ser un valor estático, como en este ejemplo, o una expresión **EL**. También, podemos especificar el ámbito de la variable con el atributo `scope` (`page`, `request`, `session` o `application`). En este ejemplo, hemos seleccionado una variable llamada `noOfRows` a 10 en el ámbito de la página (por defecto). Este es el número de filas que mostraremos en cada solicitud.

El `<c:forEach>`, en este ejemplo, toma los mismos valores para los atributos `items` y `var` como antes, pero hemos añadido dos nuevos atributos:

- El atributo *begin* toma el índice (base 0) del primer elemento de la colección a procesar. Aquí se selecciona al valor de un parámetro de solicitud llamado `first`. Para la primera solicitud, este parámetro no está disponible, por eso la expresión se evalúa a 0; en otras palabras, la primera fila.
- El atributo *end* especifica el índice del último elemento de la colección a procesar. Aquí lo hemos seleccionado al valor del parámetro `first` más `noOfRows` menos uno. Para la primera solicitud, cuando no existe el parámetro de la solicitud, este resultado es 9, por eso la acción itera sobre los índices del 0 al 9.

Luego, añadimos los enlaces "Previous" y "Next":

```
<c:choose>
  <c:when test="${param.first > 0}">
    <a href="foreach.jsp?first=<c:out value="${param.first -
noOfRows}" />">
      Previous Page</a>
  </c:when>
  <c:otherwise>
    Previous Page
  </c:otherwise>
</c:choose>
<c:choose>
  <c:when test="${param.first + noOfRows <
forecasts.rowsCount}">
    <a href="foreach.jsp?first=<c:out value="${param.first +
noOfRows}" />">
```

```

                Next Page</a>
        </c:when>
        <c:otherwise>
            Next Page
        </c:otherwise>
    </c:choose>

```

El `<c:choose>` agrupa una o más acciones `<c:when>`, cada una especificando una condición booleana diferente. La acción `<c:choose>` chequea cada condición en orden y sólo permite la primera acción `<c:choose>` con una condición que se evalúe a true para procesar su cuerpo. El cuerpo `<c:choose>` también podría contener un `<c:otherwise>`. Su cuerpo sólo se procesa si ninguna de las condiciones de los `<c:when>` es true.

En este ejemplo, la primera acción `<c:when>` comprueba si el parámetro `first` es mayor que cero, es decir, si la página muestra un subconjunto de filas distinto del primero. Si esto es cierto, el cuerpo de la acción `<c:when>` añade un enlace a la misma página con un parámetro `first` seleccionado al índice del subconjunto anterior. Si no es true, se procesa el cuerpo de la acción `<c:otherwise>`, añadiendo sólo el texto "Previous Page". El segundo bloque `<c:choose>` proporciona lógica similar para añadir el enlace "Next Page".

1.4 Procesar URL

Los ejemplos anteriores funcionan bien mientras se utilicen las cookies para seguimiento de sesión. No está todo dado; el navegador podría tener desactivadas las cookies, o no soportarlas. Por lo tanto, es una buena idea activar el contenedor para usar la reescritura de URL como backup de los cookies. La reescritura de URL, como se podría conocer, significa poner el ID de la sesión en todas las URL usadas en los enlaces y formularios de la página. Una URL reescrita se parece a algo como esto:

myPage.jsp;jsessionid=ah3bf5e317xmw5

Cuando el usuario pulsa en una línea como esta, el identificador de sesión ID se envía al contenedor como parte de la URL. La librería corazón de JSTL incluye la acción `<c:url>`, que tiene cuidado de la reescritura de URL por nosotros. Aquí está como podemos usarla para mejorar la generación del enlace "Previous Page" del ejemplo anterior:

```

<c:url var="previous" value="foreach.jsp">
    <c:param name="first" value="{param.first - noOfRows}" />
</c:url>
<a href="{c:url value="{previous}"/}">Previous Page</a>

```

`<c:url>` soporta un atributo `var`, usado para especificar una variable para contener la URL codificada, y un atributo `value` para contener la URL a codificar. Se pueden especificar parámetros string para solicitar la URL usando acciones `<c:param>`. Los caracteres especiales en los parámetros especificados por elementos anidados son codificados (si es necesario) y luego añadidos a la URL como parámetros string de la

consulta. El resultado final se pasa a través del proceso de reescritura de URL, añadiendo un ID de sesión si está desactivado el seguimiento de sesión usando cookies. En este ejemplo, se utiliza la URL codificada como un valor de atributo href en un elemento de enlace HTML.

`<c:url>` también realiza otro buen servicio. Las URL relativas en elementos HTML también deben ser relativas a la página que los contiene o al directorio raíz del servidor (si empiezan con una barra inclinada). La primera parte del path de una URL de una página JSP se llama *path de contexto*, y podría variar de instalación a instalación. Por lo tanto, deberíamos evitar codificar el path de contexto en las páginas JSP. Pero algunas veces realmente queremos utilizar un path de URL relativo al servidor en elemento HTML; por ejemplo, cuando necesitamos referirnos a un archivo de imagen localizado en el directorio /images compartido por todas las páginas JSP. Las buenas noticias es que si especificamos una URL con un barra inclinada como el valor `<c:url>`, lo convierte en un path relativo al servidor. Por ejemplo, en una aplicación con el path de contexto /myApp, la acción `<c:url>` convierte el path a /myApp/images/logo.gif:

```
<c:url value="/images/logo.gif" />
```

Hay unas cuantas acciones más relacionadas con las URL en la librería corazón. La acción `<c:import>` es una acción más flexible que la acción estándar `<jsp:include>`. Podemos usarla para incluir contenido desde los recursos dentro de la misma aplicación web, desde otras aplicaciones web en el mismo contenedor, o desde otros servidores, usando protocolos como HTTP y FTP. La acción `<c:redirect>` nos permite redirigir a otro recurso en la misma aplicación web, en otra aplicación web o en un servidor diferente.

1.5 Internacionalización y Formateo

Las grandes *sites* normalmente necesitan complacer a los visitantes de todo el mundo, por lo que servir el contenido sólo en un idioma no es suficiente. Para desarrollar una *site* que proporcione una elección de idiomas, tenemos dos opciones:

- Escribir un conjunto de páginas para cada idioma
- Escribir un conjunto compartido de páginas que ponga el contenido en diferentes idiomas desde fuentes externas

Frecuentemente, terminaremos con una mezcla de estas técnicas, usar páginas separadas para la mayoría de las grandes cantidades de contenido estático y páginas compartidas cuando la cantidad de contenido sea pequeña, pero dinámica (por ejemplo, una página con unas pocas etiquetas fijas mostradas en diferentes idiomas y todos los demás datos que vengan desde una base de datos).

Preparar una aplicación para varios idiomas se llama **internacionalización** (comúnmente abreviado **i18n**); y hacer que el contenido esté disponible para un idioma específico se llama **localización** (o **l10n**). Para hacer esto, necesitamos considerar otras cosas además del idioma: cómo se formatean las fechas y números entre los diferentes países e, incluso, dentro de los países. También, podríamos necesitar adaptar los colores, las imágenes y otro contenido no textual. El término **localidad** se refiere a un conjunto de reglas y contenido aceptable para una región o cultura.

1.6 Formateo de Fechas y Números Sensible a la Localidad

JSTL incluye un conjunto de acciones para simplificar la internacionalización, principalmente cuando páginas compartidas se usan para varios idiomas. Primero, veamos cómo formatear apropiadamente fechas y números. Este ejemplo formatea la fecha actual y un número basándose en las reglas de la localidad por defecto:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>
<html>
<body>
  <h1>Formatting with the default locale</h1>
  <jsp:useBean id="now" class="java.util.Date" />
  Date: <fmt:formatDate value="${now}" dateStyle="full" />
  Number: <fmt:formatNumber value="${now.time}" />
</body>
</html>
```

La primera línea es la directiva taglib para la librería JSTL que contiene las acciones de formateo e internacionalización. El prefijo por defecto, usado aquí, es `fmt`. Para obtener un valor a formatear, entonces crea un objeto `java.util.Date` que representa la fecha y hora actuales, y lo graba como una variable llamada `now`.

Ya estamos preparados para hacer algún formateo sensible a la localidad. La acción JSTL `<fmt:formatDate>` formatea el valor de la variable `now` asignado al valor del atributo `value` usando el tipo de expresión **EL** que aprendimos en la página anterior. El atributo `dateStyle` le dice a la acción cómo debe formatear la fecha.

Podemos usar cualquiera de los valores `default`, `short`, `medium`, `long`, o `full` para el atributo `dateStyle`. El tipo de formateo que representa cada uno de estos tipos depende exactamente de la localidad. Para la localidad *English*, `full` resulta en un string como **Thursday, August 29, 2002**. En este ejemplo, sólo hemos formateado la parte de la fecha, pero también podemos incluir la parte de la hora (o formatear sólo la hora), y definir reglas de formateo personalizadas tanto para la fecha como para la hora en lugar de los estilos dependientes de la localidad:

```
<fmt:formatDate value="${now}" type="both"
pattern="EEEE, dd MMMM yyyy, HH:mm" />
```

El atributo `pattern` toma un patrón de formateo personalizado del mismo tipo que la clase `java.text.SimpleDateFormat`. El patrón usado aquí resulta en **Thursday, 29 August 2002, 17:29** con la localidad *English*. La acción `<fmt:formatNumber>` soporta atributos similares para especificar cómo formatear un número usando estilos dependientes de la localidad para números normales, valores de moneda, o un porcentaje, así como usar patrones personalizados de diferentes tipos.

1.6 Usar JSTL para Seleccionar la Localidad

Volviendo a la cuestión principal: ¿Cómo se determina la localidad para formatear las fechas y los números? Si usamos las acciones JSTL exactamente como en este

ejemplo, sin hacer nada más, el formateo de localidad se determina comparando las localidades especificadas en un cabecera de solicitud llamada *Accept-Language* con las localidades soportadas por el entorno de ejecución *Java*.

La cabecera, si está presente, contiene una o más especificaciones de localidades (en la forma de código de idioma y posiblemente un código de país), con información sobre su prioridad relativa. El usuario utiliza las configuraciones del navegador para definir qué especificaciones de localidades enviar. Las localidades de la solicitud se comparan en orden de prioridad con las ofrecidas por la máquina virtual *Java*, y se selecciona la que mejor coincida.

Si no se encuentra una correspondencia, la acción de formateo busca la llamada **selección de configuración de la localidad pre-definida**. Una selección de configuración es un valor seleccionado por un parámetro de contexto en el archivo *web.xml* de la aplicación o por una acción JSTL o una sentencia *Java* en uno de los ámbitos JSP. Para seleccionar esta localidad en el archivo *web.xml*, incluimos estos elementos:

```
<context-param>
<param-name>
    javax.servlet.jsp.jstl.fmt.fallbackLocale
</param-name>
<param-value>
    de
</param-value>
</context-param>
```

Con esta selección, se usará la localidad *German* (especificada por el código de idioma de como valor de parámetro) si ninguna de las localidades especificadas por la solicitud está soportada por el entorno *Java*.

JSTL también nos permite seleccionar una localidad por defecto para la aplicación que podremos sobrescribir cuando lo necesitemos. Esto nos da control total sobre la localidad a utilizar, en lugar de dejarlo a la configuración del navegador del visitante. La localidad por defecto también es una selección de configuración. Se puede especificar con un parámetro de contexto en el archivo *web.xml* de esta forma:

```
<context-param>
<param-name>
    javax.servlet.jsp.jstl.fmt.locale
</param-name>
<param-value>
    en
</param-value>
</context-param>
```

Esta selección establece *English* como la localidad por defecto, resultando en la anterior página JSP siempre formateará la fecha y los números de acuerdo con las reglas inglesas.

Para sobrescribir la configuración de localidad por defecto, podemos usar la acción `<fmt:setLocale>`. Selecciona la localidad por defecto dentro de un ámbito JSP específico. Aquí tenemos un ejemplo que selecciona la localidad por defecto del ámbito de la página, basándose en una cookie que sigue la pista de la localidad seleccionada por el usuario en una visita anterior:

```
<h1>Formatting with a locale set by setLocale</h1>
<c:if test="${!empty cookie.preferredLocale}">
<fmt:setLocale value="${cookie.preferredLocale.value}" />
</c:if>
<jsp:useBean id="now" class="java.util.Date" />
Date: <fmt:formatDate value="${now}" dateStyle="full" />
Number: <fmt:formatNumber value="${now.time}" />
```

Aquí, primero hemos usado la acción JSTL `<c:if>` para comprobar si con la solicitud se ha recibido una cookie llamada `preferredLocale`. Si es así, la acción `<fmt:setLocale>` sobrescribe la localidad para la página actual y selecciona la variable de configuración de la localidad en el ámbito de la página. Si lo queremos, podemos seleccionar la localidad para otro ámbito usando el atributo `scope`. Finalmente, se formatean la fecha y el número de acuerdo con las reglas de la localidad especificada por la cookie, o la localidad por defecto, si la cookie no está presente.

Además de acciones para formateo, JSTL incluye acciones para interpretar (analizar) fechas y número de una forma sensible a la localidad: `<fmt:parseDate>` y `<fmt:parseNumber>`. Estas acciones soportan casi todos los mismos atributos que sus contrapartidas de formateo, y se pueden utilizar para convertir entradas de fechas y números en sus formas nativas antes de procesarlas.

1.7 Usar un Controlador para Seleccionar la Localidad

Si se ha desarrollado aplicaciones web usando tecnología *Java*, sin duda se habrá oído algo sobre el patrón MVC y mejor, sobre el marco de trabajo *struts* de *Apache*. La idea básica que hay detrás del patrón MVC es que una aplicación es más fácil de mantener y evolucionar si las diferentes partes de la aplicación (Modelo, Vista y Controlador) se implementan como componentes separados. Para *Java*, esto normalmente significa usar *beans* como el Modelo (lógica de negocio), páginas JSP para la Vista (la interfaz de usuario) y un servlet como el Controlador (la parte que controla la comunicación entre la Vista y el Modelo). *Struts* proporciona un *servlet* Controlador genérico que delega el procesamiento específico de tipos específicos de solicitudes a clases llamadas *Action*, y luego usa una página JSP especificada por la *Action* para generar la respuesta.

JSTL está diseñado para jugar bien en una aplicación basada en MVC exponiendo una clase que puede ser utilizada por cualquier componente *Java*, como una clase *Action* de *Struts*, para acceder a las variables de configuración usadas por las acciones JSTL en las páginas JSP que representan la capa vista. La clase se llama `javax.servlet.jsp.jstl.core.Config` y contiene constantes (variables static final *String*) para todas las variables de configuración, y métodos para seleccionar, obtener y eliminar las variables en diferentes ámbitos JSP. Podemos usar el código, como el que se muestra a continuación, en una clase *Action* de *Struts* para seleccionar la localidad por defecto para la sesión, basado en un perfil de datos, cuando un usuario entra en la aplicación:

```

import javax.servlet.jsp.jstl.core.Config;
...
public class LoginAction extends Action {

    public ActionForward perform(ActionMapping mapping,
        ActionForm form, HttpServletRequest request,
        HttpServletResponse response) throws IOException,
        ServletException {

        String userName = request.getParameter("userName");
        String password = request.getParameter("password");
        ...
        User user = authenticate(userName, password);
        if (user != null) {
            /*
             * Valid user. Set the default locale in the session
             * to the user's preferred locale
             */
            HttpSession session = request.getSession();
            Config.set(session, Config.FMT_LOCALE,
user.getLocale());
            ...
        }
    }
}

```

La clase Config proporciona cuatro métodos set(), uno por cada ámbito JSP (especificado por el primer parámetro de método). Aquí hemos usado el que selecciona la variable en el ámbito de sesión. El segundo parámetro es el nombre de la configuración, típicamente seleccionado por la constante correspondiente. El tercer parámetro es el valor de la variable. Para leer y eliminar variable, la clase proporciona métodos get() y remove() similares, más un método find() para localizar una variable en cualquier ámbito. Además de usar estos métodos en un Controlador, podríamos querer usarlos en nuestro propio manejador de etiquetas personalizado para aprovecharnos de los mecanismos de configuración JSTL.

1.8 Generar Texto Localizado

Aunque formatear fechas y número es importante cuando se localiza una aplicación, el contenido de texto es, por supuesto, incluso más importante. JSTL está basado en *Java*, por eso trata con el soporte genérico de *i18n* de la plataforma *Java*. Cuando viene con texto, este soporte está basado en lo que se llama un **paquete de recursos**. En su forma más simple, un paquete de recursos está representado por un archivo de texto que contiene claves y un valor de texto para cada clave. Este ejemplo muestra un archivo con dos claves (*hello* y *goodbye*) y sus valores:

```

hello=Hello
goodbye=Goodbye

```

Las distintas localidades se soportan creando archivos separados para cada una de ellas, con un nombre de archivo que incluye el nombre de la localidad. Por ejemplo, si

el archivo del paquete de recursos anterior representa la localidad *English*, debería almacenarse en un archivo llamado *labels_en.properties*. El paquete de recursos para la localidad Swedish sería almacenado en un archivo llamado *labels_sv.properties*, donde sv es el código de idioma. La parte fija del nombre de archivo, *labels* de este ejemplo, se llama el *nombre base* del paquete de recursos, y se combina con una especificación de localidad (como un código de idioma) para encontrar el paquete de recursos para una localidad específica. Todos los paquetes de recursos incluyen las mismas claves; sólo difiere el texto dependiendo de la localidad. Los archivos deben situarse en un directorio que sea parte del classpath del contenedor, normalmente el directorio *WEB-INF/classes* de la aplicación.

La acción JSTL que añade texto desde un paquete de recursos a una página es `<fmt:message>`. El paquete a utilizar puede especificarse de varias formas. Una es anidar las acciones `<fmt:message>` dentro del cuerpo de una acción `<fmt:bundle>`:

```
<fmt:bundle basename="labels">
Hello: <fmt:message key="hello" />
Goodbye: <fmt:message key="goodbye" />
</fmt:bundle>
```

En este caso, la acción `<fmt:bundle>` localiza el paquete de la localidad que es la correspondencia más cercana entre la selección de configuración de la localidad (o las localidades en el cabecera *Accept-Language* si no hay localidad por defecto) y los paquetes de recursos disponibles para el nombre base especificado. Las acciones anidadas obtienen el texto desde el paquete por la clave asociada.

Al igual que con las acciones de formato, podemos establecer un paquete por defecto, para toda la aplicación con un parámetro de contexto o con la acción `<fmt:setBundle>` o la clase *Config* para un ámbito JSP específico:

```
<fmt:setBundle basename="labels"/>
...
Hello: <fmt:message key="hello" />
Goodbye: <fmt:message key="goodbye" />
```

Después de que se haya definido un paquete por defecto, podemos usar acciones `<fmt:message>` independientes dentro del ámbito donde se estableció el valor por defecto.

1.9 Acceder a una Base de Datos

Un sujeto de controversia es la inclusión en JSTL de acciones para acceder a bases de datos. Algunos gente ve esto como una mala práctica y argumenta que todos los accesos a bases de datos se deberían realizar desde componentes *Java* puros en una aplicación basada en MVC en lugar de desde página JSP. Este precepto es para aplicaciones que son extensas, ya que resulta bien práctico utilizar JSTL para aplicaciones sencillas. Sin el soporte de JSTL, estas aplicaciones normalmente terminan con el código para acceder a la base de datos dentro de scripts, lo que es peor todavía para un mantenimiento y desarrollo correctos. Por lo tanto, veremos cómo acceder a bases de datos desde páginas JSP; sin embargo, esta aproximación no es la mejor para todos los tipos de aplicaciones. Si el equipo de desarrollo incluye programadores *Java*, se debería considerar seriamente encapsular el código de

acceso a la base de datos en clases *Java*, y utilizar las JSP sólo para mostrar los resultados.

Las acciones de bases de datos de JSTL están basadas en el API JDBC de *Java* y usan las abstracciones `javax.sql.DataSource` presentada en JDBC 2.0 para representar una base de datos. Un *DataSource* proporciona conexiones a una base de datos y puede implementar una característica llamada almacén de conexiones. Abrir una conexión física a una base de datos es una operación que consume mucho tiempo. Con el almacenamiento de conexiones, sólo necesitamos hacerlo una vez, y la misma conexión se puede reutilizar una y otra vez, sin los riesgos de problemas asociados con otras aproximaciones de compartición de conexiones.

1.10 Poner un objeto DataSource a Disposición de JSTL

JSTL soporta varias formas para hacer que un objeto *DataSource* esté disponible para las acciones de bases de datos. En un contenedor web con soporte JNDI (*Java Naming and Directory Interface*), se puede definir un *DataSource* por defecto como un recurso JNDI con un parámetro de contexto en el archivo *web.xml*:

```
<context-param>
<param-name>
    javax.servlet.jsp.jstl.sql.dataSource
</param-name>
<param-value>
    jdbc/Production
</param-value>
</context-param>
```

Se deben utilizar las herramientas de configuración JNDI del contenedor WEB para configurar un recurso JNDI con el nombre especificado; por ejemplo, con un nombre de usuario y una *password* de una cuenta de usuario en una base de datos, las conexiones mínimas y máximas en el almacén, etc. La forma de realizar esto varía entre los contenedores y está fuera del ámbito de este artículo.

Una alternativa para los contenedores que no soportan JNDI es que un oyente del ciclo de vida de la aplicación (contexto del servlet) cree y configure un *DataSource* y lo seleccione como el valor por defecto usando la clase *Config* de JSTL:

```
import javax.servlet.*;
import javax.servlet.http.*;
import oracle.jdbc.pool.*;

public class AppListener implements ServletContextListener {

    private OracleConnectionCacheImpl ds =null;

    public void contextInitialized(ServletContextEvent sce){
        ServletContext application =sce.getServletContext();

        try {
```

```

        ds = new OracleConnectionCacheImpl();

ds.setURL("jdbc:oracle:thin:@voyager2:1521:Oracle9i");
        ds.setMaxLimit(20);
        ds.setUser("scott");
        ds.setPassword("tiger");
    }
    catch (Exception e){
        application.log("Failed to create data source:"+
e.getMessage());
    }
    Config.set(application, Config.SQL_DATASOURCE, ds);
}
...
}

```

La clase oyente de este ejemplo crea un `DataSource` con capacidades de almacén de conexiones para una base de datos Oracle9i, y la hace disponible como por defecto para las acciones JSTL usando la clase `Config` para seleccionar la variable de configuración correspondiente.

Una tercera forma, sólo disponible para prototipos o aplicaciones que no van a utilizarse tan duramente como para necesitar el almacén de conexiones, es usar la acción `<sql:setDataSource>`:

```

<sql:setDataSource
url="jdbc:mysql://dbserver/dbname"
driver="org.gjt.mm.mysql.Driver"
user="scott"
password="tiger" />

```

Esta acción crea una sencilla fuente de datos, sin almacenamiento, para la URL JDBC especificada, con el usuario y la password, usando el driver JDBC especificado. Podríamos usar esta acción para empezar, pero recomendamos la utilización de una de las otras alternativas para una *site* del mundo real. Además de privarnos del almacén de conexiones para una fuente de datos creada de esta forma, no es una buena idea incluir información sensible como la URL de la base de datos, el nombre de usuario y la password en una página JSP, ya que sería posible para alguien acceder al código fuente de la página.

1.11 Leer Datos de la Base de Datos

Con una `DataSource` a nuestra disposición, podemos acceder a la base de datos. Aquí podemos leer datos desde una base de datos representada por el `DataSource` por defecto:

```

<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql" %>

<html>
<body>
  <h1>Reading database data</h1>
  <sql:query var="emps" sql="SELECT * FROM Employee" />
  ...
</body>
</html>

```

Primero, necesitamos declarar la librería JSTL que contiene las acciones de bases de datos, usando la directiva `taglib` de la parte superior de este ejemplo. La acción `<sql:query>` ejecuta la sentencia *SQL SELECT* especificada por el atributo `sql` (o como el cuerpo del elemento acción) y graba los resultados en una variable nombrada por el atributo `var`.

El resultado de la consulta a la base de datos se devuelve como un *bean* del tipo `javax.servlet.jsp.jstl.sql.Result` con varias propiedades de solo lectura:

Propiedad	Tipo Java	Descripción
rows	<code>java.util.SortedMap[]</code>	Un <i>array</i> con un mapa insensible a las mayúsculas por cada fila con las claves correspondiendo con los nombres de columna y valores correspondiendo con los valores de columna
rowsByIndex	<code>Object[][]</code>	Un <i>array</i> con un array por fila con valores de columna
columnNames	<code>String[]</code>	Un <i>array</i> con nombres de columnas
rowCount	<code>int</code>	El número de filas en el resultado
limitedByMaxRows	<code>boolean</code>	<i>true</i> si no se han incluido todas las filas debido a que se ha alcanzado el límite máximo de filas especificado

Ya vimos cómo utilizar la acción `<c:forEach>` para mostrar todas o sólo algunas filas, por eso ahora veremos cómo podemos *obtener* sólo algunas filas para mostrarlas todas en esta parte. Los enlaces **Next** y **Previous** le permiten al usuario solicitar un conjunto diferente. Primero, aquí está cómo leer un subconjunto de filas y luego mostrar el subconjunto completo:

```

<c:set var="noOfRows" value="10" />

<sql:query var="emps"
startRow="${param.start}" maxRows="${noOfRows}">
SELECT * FROM Employee
</sql:query>

<ul>
<c:forEach items="${emps.rows}" var="${emp}">

```

```
<li><c:out value="\${emp.name}" />
</c:forEach>
</ul>
```

El atributo `startRow` para la acción `<sql:query>` se envía a una expresión **EL** que lee el valor de un parámetro de solicitud llamado `start`. Luego veremos cómo cambia este valor cuando se pulsa sobre los enlaces **Next** y **Previous**. La primera vez que se accede a la página, el parámetro no existe, por eso la expresión se evalúa a 0. Esto significa que el resultado de la consulta contiene filas empezando con la primera que corresponda (índice 0). El atributo `maxRows` limita el número total de filas del valor de la variable `noOfRows`, lo seleccionamos a 10 en este ejemplo. La acción `<c:forEach>` hace un bucle sobre todas las columnas del resultado y genera una lista de ítems con los valores de columna por cada fila.

También, debemos generar los enlaces **Next** y **Previous** para permitir que el usuario seleccione un nuevo conjunto de filas:

```
<c:choose>
<c:when test="\${param.start > 0}">
  <a href="emplist.jsp?start=<c:out
    value="\${param.start - noOfRows}" />">Previous Page</a>
</c:when>
<c:otherwise>
  Previous Page
</c:otherwise>
</c:choose>
<c:choose>
<c:when test="\${emps.limitedByMaxRows}">
  <a href="emplist.jsp?start=<c:out
    value="\${param.start + noOfRows}" />">Next Page</a>
</c:when>
<c:otherwise>
  Next Page
</c:otherwise>
</c:choose>
```

El primer bloque `<c:choose>` es idéntico al de la página 1; si el parámetro de la solicitud `start` es mayor que cero, la página actual muestra un subconjunto de filas distinto del primero, por eso se añade un enlace **Previous**. El enlace apunta hacia la misma página, e incluye el parámetro `start` con un valor que es el valor actual menos el número de filas mostrado por cada página.

El segundo bloque `<c:choose>` se aprovecha de la propiedad `limitedByMaxRows` del resultado de la consulta. Si esta propiedad es `true`, significa que se ha truncado el resultado al número de filas mostrado por cada página. De aquí, se genera el enlace **Next** con valor del parámetro `start` para el nuevo subconjunto de filas.

1.12 Escribir Datos en la Base de Datos

Además de leer datos desde una base de datos, también podemos usar JSTL para actualizar información. Este ejemplo muestra cómo insertar una nueva fila en una tabla:

```
<c:catch var="error">
<fmt:parseDate var="empDate" value="\${param.empDate}"
  pattern="yyyy-MM-dd" />
</c:catch>
<c:if test="\${error != null}">
<jsp:useBean id="empDate" class="java.util.Date" />
</c:if>

<sql:update>
INSERT INTO Employee (FirstName, LastName, EmpDate)
  VALUES(?, ?, ?)
<sql:param value="\${param.firstName}" />
<sql:param value="\${param.lastName}" />
<sql:dateParam value="\${empDate}" type="date" />
</sql:update>
```

Antes de insertar la fila, este ejemplo ilustra cómo usar las acciones de validación JSTL. La página espera que todos los datos para la nueva fila se envíen como parámetros de solicitud (quizás introducidos en un formulario HTML), incluyendo una fecha de contratación. Antes de que la fecha se pueda insertar en la base de datos, debemos convertirla a su forma nativa *Java*. Esto es lo que hace la acción `<fmt:parseDate>`. El atributo `value` contiene una expresión **EL** que obtiene el valor del parámetro de solicitud `empDate`. La acción trata de interpretarlo como una fecha escrita en el formato especificado por el atributo *pattern* (un año de cuatro dígitos, seguido por dos dígitos del mes y dos dígitos del día, separados por barras inclinadas). Si tiene éxito, almacena la fecha en su forma nativa con el nombre especificado por el atributo `var`.

La acción `<c:catch>` tiene en cuenta los strings inválidos. Si el valor del parámetro no puede ser interpretado como una fecha, el `<fmt:parseDate>` lanza una excepción, que la acción `<c:catch>` captura y graba en la variable especificada. Cuando esto sucede, la condición de comprobación de la acción `<c:if>` se evalúa a *true*, por lo que la fecha de contratación es creada por la acción `<jsp:useBean>` anidada.

Para insertar la fila, usamos la acción `<sql:update>`. Como la acción de consulta, la sentencia SQL se puede especificar como el cuerpo del elemento o mediante un atributo `sql`. La acción `<sql:update>` se puede usar para ejecutar sentencias *INSERT*, *UPDATE*, *DELETE*, así como sentencias para crear o eliminar objetos en la base de datos, como *CREATE TABLE* y *DROP TABLE*. El número de filas afectado por la sentencia puede capturarse de forma opcional en una variable nombrada por el atributo `var`.

En este ejemplo (como en la mayoría de las aplicaciones del mundo real), no se conocen los nombres de las columnas en tiempo de ejecución; vienen de los parámetros de la solicitud. Por lo tanto, la sentencia *INSERT* de SQL incluye una marca de interrogación por cada valor como un contenedor y parámetros internos de la acción que seleccionan el valor dinámicamente. Las columnas `FirstName` y `LastName`

son columnas de texto y las acciones `<sql:param>` seleccionan sus valores al valor del parámetro de solicitud correspondiente.

Sin embargo, la columna `EmpDate` es una columna de fecha, demandando una atención especial. Primero de todo, debemos usar una variable que contenga la fecha en su forma nativa (un objeto `java.util.Date`) en lugar de usar el valor del parámetro de solicitud, usamos la variable creada por las acciones `<fmt:parseDate>` o `<jsp:useBean>`. Segundo, debemos usar la acción `<sql:dateParam>` para seleccionar el valor. En este ejemplo, hemos usado la parte de la fecha, por eso hemos seleccionado el atributo opcional `type` a `date`. Otros valores válidos son `time` y `timestamp` (por defecto), para las columnas que sólo toman la hora, o la fecha y la hora.

Hay una acción JSTL más que no hemos descrito hasta ahora: `<sql:transaction>`. Podemos usarla para agrupar varias acciones `update` (o incluso `query`) donde todas ellas se deben ejecutar como parte de la misma transacción de la base de datos. El ejemplo estándar es transferir una cantidad de dinero de una cuenta a otra; implementada como una sentencia SQL que elimina el dinero de la primera cuenta y otra sentencia que lo añade a la segunda.

Si encapsulamos todos los accesos a una base de datos en clases *Java* en lugar de usar las acciones de JSTL, todavía hay una parte de JSTL que nos puede ser útil. Es una clase llamada `javax.servlet.jsp.jstl.sql.ResultSupport`, con estos dos métodos:

```
public static Result toResult(java.sql.ResultSet rs);
public static Result toResult(java.sql.ResultSet rs, int maxRows);
```

Podemos usar esta clase para convertir un objeto `ResultSet` estándar JDBC en un objeto `Result` JSTL antes de reenviarlo a la página JSP para mostrarlo. Las acciones JSTL pueden acceder fácilmente a los datos de un objeto `Result`, como vimos anteriormente. Otra aproximación, todavía mejor, es pasar el resultado de la consulta a la página JSP como una estructura de datos personalizada, como una *List* de *beans* que contienen los datos de cada fila, pero el objeto `Result` aún es un buen candidato para prototipos y pequeñas aplicaciones.

Función	Descripción
<code>fn:contains(string, substring)</code>	Devuelve <i>true</i> si el <i>string</i> contiene <i>substring</i> .
<code>fn:containsIgnoreCase(string, substring)</code>	Devuelve <i>true</i> si el <i>string</i> contiene a <i>substring</i> , ignorando capitalización.
<code>fn:endsWith(string, suffix)</code>	Devuelve <i>true</i> si el <i>string</i> termina con <i>suffix</i> .
<code>fn:escapeXml(string)</code>	Devuelve el <i>string</i> con todos aquellos caracteres con especial significado en XML y HTML convertidos a sus códigos de escape pertinentes.
<code>fn:indexOf(string, substring)</code>	Devuelve la primera ocurrencia

	de substring en el <i>string</i> .
fn:join(array, separator)	Devuelve un <i>string</i> compuesto por los elementos del <i>array</i> , separados por <i>separator</i> .
fn:length(item)	Devuelve el número de elementos en ítem (si un <i>array</i> o colección), o el número de caracteres (es un <i>string</i>).
fn:replace(string, before, after)	Devuelve un <i>string</i> donde todas las ocurrencias del <i>string before</i> han sido reemplazadas por el <i>string after</i> .
fn:split(string, separator)	Devuelve un array donde los elementos son las partes del <i>string</i> separadas por <i>separator</i> .
fn:startsWith(string, prefix)	Devuelve <i>true</i> si el <i>string</i> comienza por <i>prefix</i> .
fn:substring(string, begin, end)	Devuelve la parte del <i>string</i> que comienza por el índice <i>begin</i> y que acaba en el índice <i>end</i> .
fn:substringAfter(string, substring)	Devuelve la parte del <i>string</i> que sigue a <i>substring</i> .
fn:substringBefore(string, substring)	Devuelve la parte de <i>string</i> que precede a <i>substring</i> .
fn:toLowerCase(<i>string</i>)	Devuelve un <i>string</i> con todos los caracteres de la entrada convertidos a minúsculas.
fn:toUpperCase(string)	Devuelve un <i>string</i> con todos los caracteres de la entrada convertidos a mayúsculas.
fn:trim(string)	Devuelve un <i>string</i> sin espacios en sus laterales.

Preguntas de Certificación

1. ¿Cuál es la etiqueta que invoca al contenedor web y busca al *Tag Library Descriptor*?

- a <taglib-directory></taglib-directory>
- b <taglib-uri></taglib-uri>
- c <taglib-location></taglib-location>
- d <tld-directory></tld-directory>
- e <taglib-name></taglib-name>

2. ¿Cuáles son las dos expresiones que no retornan el campo accept de la cabecera?

- a \${header.accept}
- b \${header[accept]}
- c \${header['accept']}
- d \${header["accept"]}
- e \${header.'accept'}


3. ¿Cuál de los siguientes métodos permite registrar el nombre de una función?

- a <function-signature></function-signature>
- b <function-name></function-name>
- c <method-class></method-class>
- d <method-signature></method-signature>
- e <function-class></function-class>

4. ¿Cuál de los siguientes métodos es correcto para el uso de una función EL?

- a public static expFun(void)
- b expFun(void)
- c private expFun(void)
- d public expFun(void)
- e public native expFun(void)

Resumen

 Los usos de Lenguaje de Expresiones es obtener las propiedades de un objeto directamente a través de estas formas:

```
${myObj.myProperty}$
${myObj["myProperty"]}
${myObj[varWithTheName]}$
```

 Si desea saber más acerca de estos temas, puede consultar la siguiente página.

 <http://java.sun.com/javase/5/docs/tutorial/doc/bnahq.html>

Aquí hallará definiciones sobre los temas planteados.