Tutoriales

Desarrollo de Aplicaciones Web con JSP y XML

Autor: Sun
Traductor: Juan Antonio Palos (Ozito)

Últimos comentarios

Últimos 5 comentarios

Muy bueno (14/10/2005)

Por Carlos

Hola, muy bueno el manual, pero alguien me lo podría mandar en pdf?? Muchas gracias

Buen Manual Pero ... (09/10/2005)

Por Web oncito

Podran enviarme este manual al correo mex_tiranus@yahoo.com.mx

Muy bueno (01/10/2005)

Por mkmendo

Muy bueno muchachos. Pero seria excelente tenerlo en pdf. Para los que estan buscando el manual de JBUILDER X , por aqui lo tengo y en PDF (600 pag listo para imprimir)Espero que aqui tambien hagan los mismo. EN PDF PLEASEEEEEEEEEEE ;(

ah me olvida. Cualquier novedad estoy online MKMENDO@YAHOO.COM

Peticion (07/09/2005)

Por Rogelio

Muy bueno este curso. Me gustaria bajarlo a mi computador ¿quien puede ayudarme?

Contactos (25/07/2005)

Por Kadith

Estoy leyendo el manual de JSP y HTML, pero me gustaria relacionarme con los que estan inscrito en el curso, falta una sección de chat entre los inscrito, interesados Kadith1977@yahoo.com, estamos en contacto.

Desarrollo de Aplicaciones Web con JSP y XML

Autor: Sun
Traductor: Juan Antonio Palos (Ozito)

Parte I, Conocer JSP

Si hemos tenido la oportunidad de construir aplicaciones Web usando tecnologías como CGI y Servlets, estaremos acostumbrados a la idea de escribir un programa que genere la pagina entera (las partes estáticas y dinámicas) usando el mismo programa. Si estamos buscando una solución en la cual podamos separar las dos partes, no tenemos que buscar más. Las JavaServer Pages (JSP) están aquí.

Las páginas JSP permiten que separemos la presentación final de la lógica de negocio (capas media y final). Es un gran "Rapid Application Development" (RAD) de aplicaciones Web. Esta sección explica los conceptos y las ventajas de la tecnología JSP, y luego demostraremos cómo utilizar esta tecnología emocionante, y cómo crear componentes reutilizables para manejar formularios.

La Web Dinámica

El Web se ha desarrollado desde un sistema de información distribuido hypermedia basado en red que ofrecía información estática hasta un mercado para vender y comprar mercancias y servicios. Las aplicaciones cada vez más sofisticadas para permitir este mercado requieren una tecnología para presentar la información dinámica.

Las soluciones de primera generación incluyeron CGI, que es un mecanismo para ejecutar programas externos en un servidor web. El problema con los scripts CGI es la escalabilidad; se crea un nuevo proceso para cada petición.

Las soluciones de segunda generación incluyeron vendedores de servidores Web que proporcionaban plug-ins y a APIs para sus servidores. El problema es que sus soluciones eran específicas a sus productos servidores. Por ejemplo, Microsoft proporcionó las páginas activas del servidor (ASP) que hicieron más fácil crear el contenido dinámico. Sin embargo, su solución sólo trabajaba con Microsoft IIS o Personal Web Server. Por lo tanto, si deseabamos utilizar ASP teníamos que confiarnos a los productos de Microsoft y no estaríamos gozando de la libertad de seleccionar nuestro servidor web y sistema operativo preferidos!

Otra tecnología de segunda generación que es absolutamente popular en las empresa son los Servlets. Los Servlets hacen más fácil escribir aplicaciones del lado del servidor usando la tecnología Java. El problema con los CGI o los Servlets, sin embargo, es que tenemos que

seguir el ciclo de vida de escribir, compilar y desplegar .

Las páginas JSP son una solución de tercera generación que se pueden combinar fácilmente con algunas soluciones de la segunda generación, creando el contenido dinámico, y haciéndo más fácil y más rápido construir las aplicaciones basadas en Web que trabajan con una variedad de otras tecnologías: servidores Web, navegadores Web, servidores de aplicación y otras herramientas de desarrollo.

JavaServer Pages (JSP)

La tecnología JSP es una especificación abierta (y gratis) disponible y desarrollada por Sun Microsystems como un alternativa a Active Server Pages (ASP) de Microsoft, y son un componente dominante de la especificación de Java 2 Enterprise Edition (J2EE). Muchos de los servidores de aplicaciones comercialmente disponibles (como BEA WebLogic, IBM WebSphere, Live JRun, Orion, etcétera) ya utilizan tecnología JSP.

JSP contra ASP

JSP y ASP ofrecen funciones similares. Ambos utilizan etiquetas para permitir código embebido en una página HTML, seguimiento de sesión, y conexión a bases de datos. Algunas de las diferencias triviales son:

- Las páginas ASP están escritas en VBScript y las páginas JSP están escritas en lenguaje Java. Por lo tanto, las páginas JSP son independientes de la plataforma y las páginas ASP no lo son.
- Las páginas JSP usan tecnología **JavaBeans** como arquitectura de componentes y las páginas ASP usan componentes ActiveX.

Más allá de estas diferencias triviales, hay varias diferencias importantes, que podrían ayudarnos a elegir la tecnología para nuestras aplicaciones:

- **Velocidad y Escalabilidad**: Aunque las páginas ASP son cacheadas, siempre son interpretadas, las páginas JSP son compiladas en Servlets Java y cargadas en memoria la primera vez que se las llama, y son ejecutadas para todas las llamadas siguientes. Esto le da a las páginas JSP la ventaja de la velocidad y escalabilidad sobre las páginas ASP.

- **Etiquetas Extensibles**: Las páginas JSP tiene una característica avanzada conocida como etiquetas extensibles. Esto mecanismo permite a los desarrolladores crear etiquetas personalizadas. En otras palabras, las etiquetas extensibles nos permiten extender la síntaxis de las etiquetas de las páginas JSP. No podemos hacer esto en ASP.
- **Libertad de Elección**: A menos que instalemos Chili!Soft ASP, las páginas ASP sólo trabajan con Microsoft IIS y Personal Web Server. El uso de páginas ASP requiere un compromiso con los productos de Microsoft, mientras que las páginas JSP no nos imponen ningún servidor web ni sistema operativo. Las páginas JSP se están convirtiendo en un estándard ampliamente soportado.

Para una comparación más detallada entre las páginas JSP y ASP puedes ver Comparing JSP and ASP.

Entorno de Software

Para ejecutar las páginas JSP, necesitamos un servidor web con un contenedor Web que cumpla con las especificaciones de JSP y de Servlet. El contenedor Web se ejecuta en el servidor Web y maneja la ejecución de todas las páginas JSP y de los servlets que se ejecutan en ese servidor Web. Tomcat 3.2.1 es una completa implementación de referencia para las especificacions Java Servlet 2.2 y JSP 1.1. Dese aquí puedes descargar las versiones binarias de Tomcat.

Para configurar Tomcat:

- Configuramos la variable de entorno `JAVA_HOME` para que apunte al directorio raíz de nuestra instalación de Java 2 Standard Edition (J2SE).
- Configuramos la variable de entorno `TOMCAT_HOME` para que apunte al directorio raíz

de nuestra instalación de Tomcat.

- Para arrancar Tomcat, usamos `TOMCAT_HOME/bin/startup.bat` para Windows o `startup.sh` para UNIX.

  Por defecto, empezará a escuchar en el puerto 8080.

- Grabamos nuestros ficheros `.jsp` en `TOMCAT_HOME/webapps/examples/jsp` y nuestras clases JavaBeans en `TOMCAT_HOME/webapps/examples/web-inf/classes`.

Nota:

Si estamos trabajando bajo Windows, podemos obtener un error **Out of space environment** cuando intentemos arrancar Tomcat. Hay dos maneras de corregir esto: cambiar la configuración inicial de la memoria de la ventana de DOS a un valor mayor de 3200. O editar el fichero **config.sys** y agregar la línea siguiente: `SHELL=c:\PATHTO\command.com /E:4096 /P`.

¿Cómo Funcionan las Páginas JSP?

Una página JSP es básicamente una página Web con HTML tradicional y código Java. La extensión de fichero de una página JSP es ".jsp" en vez de ".html" o ".htm", y eso le dice al servidor que esta página requiere un manejo especial que se conseguirá con una extensión del servidor o un plug-in. Aquí hay un sencillo ejemplo :

Ejemplo 1: date.jsp

```
<HTML>
<HEAD>
<TITLE>JSP Example</TITLE>
</HEAD>
<BODY BGCOLOR="ffffcc">
<CENTER>
<H2>Date and Time</H2>
<%
java.util.Date today = new java.util.Date();
out.println("Today's date is: "+today);
%>
</CENTER>
</BODY>
</HTML>
```

Este ejemplo contiene HTML tradicional y algún código Java. La etiqueta `<%` identifica el inicio de un `scriptlet`, y la etiqueta `%>` identifica el final de un `scriptlet`. Cuando un navegador solicite la página `date.jsp` veremos algo similar a la Figura 1.

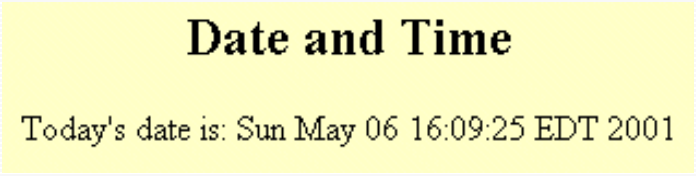


Figura 1: Petición de date.jsp

Detrás de la Escena

Cuando se llame a ésta página `(date.jsp)`, será compilada (por el motor JSP) en un Servlet Java. En este momento el Servlet es manejado por el motor Servelt como cualquier otro Servlet. El motor Servlet carga la clase Servlet (usando un cargador de clases) y lo ejecuta para crear HTML dinámico para enviarlo al navegador, como se ve en la Figura 2. Para este ejemplo, el Servelt crea un objeto `Date` y lo escribe como un `String` en el objeto `out`, que es el stream de salida hacia el navegador.

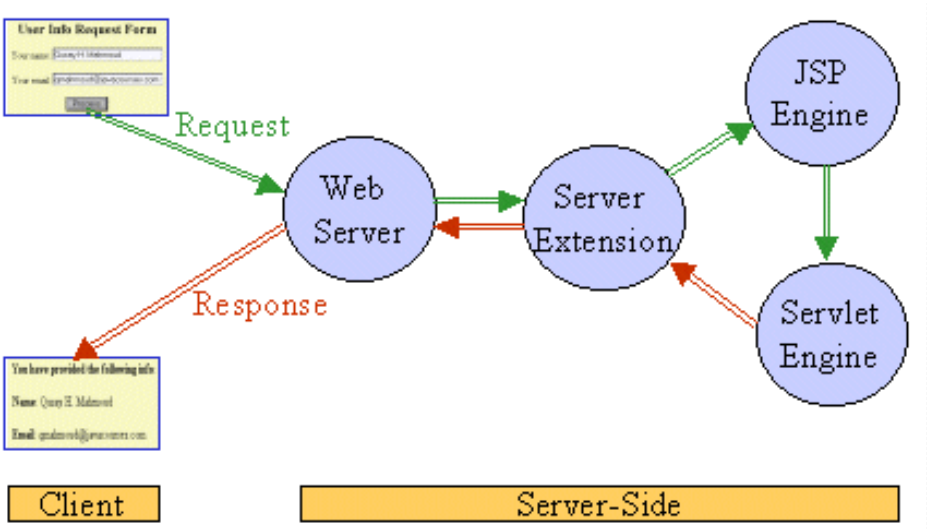


Figura 2: Flujo Petición/Respuesta cuando se llama a una página JSP.

La siguiente vez que se solicite la página, el motor JSP ejecuta el Servlet ya cargado **a menos** que la página JSP haya cambiado, en cuyo caso es automáticamente recompilada en un Servlet y ejecutada.

Elementos de Script

En el ejemplo `date.jsp` se usa todo el nombre de la clase `Date` incluyendo el nombre del paquete, lo que podría llegar a ser tedioso. Si queremos crear un ejemplar de la clase `Date` usando simplemente: `Date today = new Date();` sin tener que especificar el path completo de la clase, usamos la directiva `page` de esta forma:

Ejemplo 2 :date2.jsp

```
<%@page import="java.util.*" %>
<HTML>
<HEAD>
<TITLE>JSP Example</TITLE>
</HEAD>
<BODY BGCOLOR="ffffcc">
<CENTER>
<H2>Date and Time</H2>
<%
java.util.Date today = new java.util.Date();
out.println("Today's date is: "+today);
%>
</CENTER>
</BODY>
</HTML>
```

Todavía hay otra forma de hacer lo mismo usando la etiqueta **<%=** escrbiendo:

Ejemplo 3:date3.jsp

```
<%@page import="java.util.*" %>
<HTML>
<HEAD>
<TITLE>JSP Example</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffcc">
<CENTER>
<H2>Date and Time</H2>
Today's date is: <%= new Date() %>
</CENTER>
</BODY>
</HTML>
```

Como podemos ver, se puede conseguir el mismo resultado usando diferentes etiquetas y técnicas. Hay varios elementos de script JSP. Hay algunas reglas convencionales que nos ayudarán a usar más efectivamente los elementos de Script JSP.

- Usamos `<% ... %>` para manejar declaraciones, expresiones, o cualquier otro tipo de código válido.
- Usamos la directiva `page` como en `<%@page ... %>` para definir el lenguaje de escript. También puede usarse para especificar sentencias `import`. Aquí hay un ejemplo:

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

- Usamos `<%! .... %>` para declarar variables o métodos. Por ejemplo:

```
<%! int x = 10; double y = 2.0; %>
```

- Usamos `<%= ... %>` para definir una expresión y forzar el resultado a un `String`. Por ejemplo: `<%= a+b %>` o `<%= new java.util.Date() %>`.
- Usamos la directiva `include` como en `<%@ include ... %>` para insertar el contenido de otro fichero en el fichero JSP principal. Por ejemplo:

```
<%@include file="copyright.html" %>
```

Puedes encontrar más información sobre las directivas y scriptles de JSP en las páginas del tutorial sobre Servlets y JSP

Manejar Formularios

Una de las partes más comunes en aplicaciones de comercio electrónico es un formulario HTML donde el usuario introduce alguna información como su nombre y dirección. Usando JSP, los datos del formulario (la información que el usuario introduce en él) se almacenan en un objeto `request` que es enviado desde el navegador hasta el contenedor JSP. La petición es procesada y el resultado se envía a través de un objeto `response` de vuelta al navegador. Estos dos objetos está disponibles implícitamente para nosotros.

Para demostrar como manejar formularios HTML usando JSP, aquí tenemos un formulario de ejemplo con dos campos: uno para el nombre y otro para el email. Como podemos ver, el formulario HTML está definido en un fichero fuente JSP. Se utiliza el método `request.getParameter` para recuperar los datos desde el formulario en variables creadas usando etiquetas JSP.

La página `process.jsp` imprime un formulario o la información proporcionada por el usuario dependiendo de los valores de los campo del formulario. Si los valores del formulario son `null` se muestra el formulario, si no es así, se mostrará la información proporcionada por el usuario. Observa que el formulario es creado y manejado por el código del mismo fichero JSP.

Ejemplo 4: process.jsp

```
<HTML>
<HEAD>
<TITLE>Form Example</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffcc">
<% if (request.getParameter("name")==
null && request.getParameter("email")
 == null) { %>
<CENTER>
<H2>User Info Request Form</H2>
<FORM METHOD="GET" ACTION="process.jsp">
<P>
Your name: <input type="text" name="name" size=26>
<P>
Your email: <input type="text" name="email" size=26>
<P>
<input type="submit" value="Process">
</FORM>
```

```
</CENTER>
<% } else { %>
<%! String name, email; %>
<%
name = request.getParameter("name");
email = request.getParameter("email");
%>
<P>
<B>You have provided the following info</B>:
<P>
<B>Name</B>: <%= name %><P>
<B>Email</B>: <%= email %>
<% } %>
</BODY>
</HTML>
```

Si solicitarámos `process.jsp` desde un navegador Web, veríamos algo similar a la figura 3:

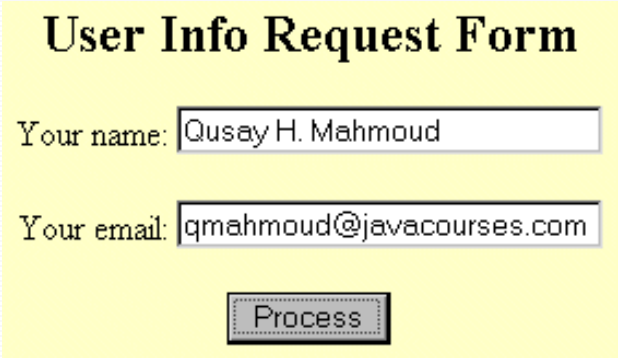


Figura 3: imagen de process.jsp

Introducimos nuestro nombre y email y pulsamos Process para enviar el formulario para su proceso, y veremos algo similar a a la Figura 4:

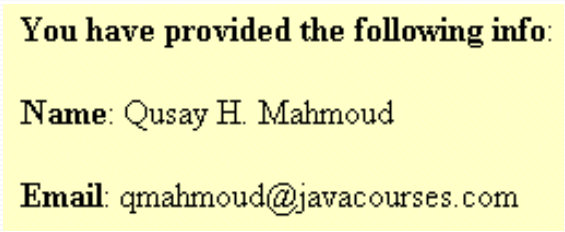


Figura 4 : Formulario procesado

Componentes Reutilizables

El ejemplo del formulario anterior es simple en el sentido de que no hay mucho código implicado. Cuanto más código esté implicado, más importante es no mezclar la lógica del negocio con la presentación final en el mismo fichero. La separación de la lógica de negocio de la presentación permite cambios en cualquier sitio sin afectar al otro. Sin embargo, el código de producción JSP se debe limitar a la presentación final. Así pues, ¿cómo implementamos la parte de la lógica de negocio?

Aquí es donde los JavaBeans entran en juego. Esta tecnología es un modelo de componente portable, independiente de la plataforma que permite a los desarrolladors escribir componentes y reutilizarlos en cualquier lugar. En el contexto de JSP, los JavaBeans contienen la lógica de negocio que devuelve datos a un script en una página JSP, que a su vez formatea los datos devueltos por el componente JavaBean para su visualización en el navegdor. Una página JSP utiliza un componente JavaBean fijando y obteniendo las propiedades que proporciona.

¿Cuáles son los beneficios?

Hay muchos beneficios en la utilización de JavaBeans para mejorar las páginas JSP:

- Componentes Reutilizables: diferentes aplicaciones pueden reutilizar los mismos componentes.

- Separación de la lógica de negocio de la lógica de presentación: podemos modificar la forma de mostrar los datos sin que afecte a la lógica del negocio.
- Protegemos nuestra propiedad intelectual manteniendo secreto nuestro código fuente.

Ejemplo: Usar JavaBeans con JSP

Ahora, veamos como modificar el ejemplo anterior, `process.jsp` para usar JavaBeans. En el formulario anterior había dos campos: `name` y `email`. En JavaBeans, son llamados propiedades. Por eso, primero escribimos un componente JavaBean con métodos `setX getX`, donde X es el nombre de la propiedad. Por ejemplo, si tenemos unos métodos llamados `setName` y `getName` entonces tenemos una propiedad llamada `name`. El ejemplo 5 muestra un componente `FormBean`.

Los buenos componentes deben poder interoperar con otros componentes de diferentes vendedores. Por lo tanto, para conseguir la reutilización del componente, debemos seguir dos reglas importantes (que son impuestas por la arquitectura JavaBeans):

1. Nuestra clase bean debe proporcionar un constructor sin argumentos para que pueda ser creado usando `Beans.instantiate`.
2. Nuestra clase bean debe soportar persistencia implementando el interface `Serializable` o `Externalizable`.

Ejemplo 5: FormBean.java

```
package userinfo;
import java.io.*;

public class FormBean implements Serializable {
  private String name;
  private String email;
  public FormBean() {
   name = null;
   email = null;
  }
  public void setName(String name) {
   this.name = name;
  }
  public String getName() {
   return name;
  }
  public void setEmail(String email) {
   this.email = email;
  }
  public String getEmail() {
   return email;
  }
}
```

Para poder usar el componente `FormBean` en el fichero JSP, necesitamos ejemplarizar el componente. Esto se hace usando la etiqueta `<jsp:useBean>`. La siguiente línea `<jsp:setProperty>` se ejecuta cuando se ha ejemplarizado el bean, y se usa para inicializar sus propiedades. En este caso, ambas propiedades (`name` y `email`) se configuran usando una sóla sentencia. Otra posible forma de configurar las propiedades es hacerlo una a una, pero primero necesitamos recuperar los datos desde el formulario. Aquí tenemos un ejemplo de como configurar la propiedad `name`:

```
<%! String yourname, youremail; %>
<% yourname = request.getParameter("name"); %>
<jsp:setProperty name="formbean" property="name"
value="<%=yourname%>"/>
```

Una vez que se han inicializado las propiedades con los datos recuperados del formulario, se recuperan los valores de las propiedades usando `<jsp:getProperty>` en la parte `else`, como se ve en el Ejemplo 6:

Ejemplo 6: process2.jsp

```
<jsp:useBean id="formbean" class="userinfo.FormBean"/>
<jsp:setProperty name="formbean" property="*"/>
<HTML>
<HEAD>
<TITLE>Form Example</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffcc">
<% if (request.getParameter("name")==null
 && request.getParameter("email") == null) { %>
<CENTER>
<H2>User Info Request Form </H2>
<form method="GET" action="process2.jsp">
<P>
Your name: <input type="text" name="name" size=27>
<p>
Your email: <input type="text" name="email" size=27>
<P>
<input type="submit" value="Process">
</FORM>
</CENTER>
<% } else { %>
<P>
<B>You have provided the following info</B>:
<P>
<B>Name</B>: <jsp:getProperty name="formbean" property="name"/>
<P>
<B>Email</B>: <jsp:getProperty name="formbean" property="email"/
>
<% } %>
</BODY>
</HTML>
```

Conclusión

Los desarrolladores interesados en crear aplicaciones Web de calidad deben familiarizarse con las tecnologías que son aplicables no solamente para el mercado de hoy en día sino también para el de mañana, a saber JSP y XML. La siguiente página discutirá las capacidades que proporciona la tecnología JSP y que son ideales para trabajar con XML; y muestra cómo utilizar con eficacia JSP y XML. JSP y XML hacen una combinación excelente para las aplicaciones Web que comparten información, porque las páginas JSP tienen soporte interno de XML en la forma de librerías de etiquetas JSP personalizadas.

Indice de contenidos

Desarrollo de Aplicaciones Web con JSP y XML

Autor: Sun
Traductor: Juan Antonio Palos (Ozito)

Parte II: JSP con XML en Mente

El "Extensible Markup Language" (XML) se ha convertido en el formato estándar de facto para la representación de datos en Internet. Los datos XML se pueden procesar e interpretar en cualquier plataforma -- desde el dispositivo de mano a una unidad central. Es un compañero perfecto para las aplicaciones Java que necesitan datos portables.

Java es el lenguaje de programación ganador para utilizar con XML. La mayoría de los analizadores de sintaxis de XML se escriben en Java, y proporciona una colección comprensiva de APIs Java pensada específicamente para construir aplicaciones basadas en XML. La tecnología JavaServer Pages (JSP) tiene acceso a todo esto puesto que puede utilizar todo el poder de la plataforma Java para acceder a objetos del lenguaje de programación para analizar y transformar documentos XML. JSP se ha diseñado con XML en mente; podemos escribir una página JSP como un documento XML!

Esta página presenta una breve introducción a XML y luego muestra como:

- Presentar documentos XML
- Generar XML usando JSP
- Analizar documentos XML usando "Simple Access API for XML" (SAX) y "Document Object Model" (DOM)

- Usar SAX y DOM en JSP
- Transformar XML en otros lenguajes de marcas.

Introducción a XML

XML es un metalenguaje usado para definir documentos que contienen datos estructurados. Las características y beneficios del XML se pueden agrupar en estas áreas principales:

- Extensibilidad: como metalenguaje, XML puede usarse para crear sus propios lenguajes de marcas. Hoy en día existen muchos lenguajes de marcas basados en XML, incluyendo "Wireless Markup Language" (WML).
- Estructura precisa: HTML sufre de una pobre estructura que hace dificil procesar eficientemente documentos HTML. Por otro lado, los documentos XML están bien estructurados, cada documento tiene un elemento raíz y todos los demás elementos deben estar anidados dentro de otros elementos.
- Dos tipos de documentos: hay dos tipos principales de documentos XML.
    1. **Documentos Válidos**: un documento XML válido está definido por una "Document Type Definition" (DTD), que es la gramática del documento que define qué tipos de elementos, atributos y entidades podría haber en el documento. El DTD define el orden y también la ocurrencia de elementos.
    2. **Documentos Bien-Formateados**: un documento XML bien formateado no tiene que adherirse a una DTD. Pero debe seguir dos reglas: 1) todo elemento debe tener una etiqueta de apertura y otra de cierre. 2) debe haber un elemento raíz que contenga todos los otros elementos.
- Extensión Poderosa: Como se mencionó anteriormente, XML sólo se usa para definir la síntaxis. En otras palabras, se usa para definir contenido. Para definir la semántica, el estilo o la presentación, necesitamos usar "Extensible Stylesheet Language" (XSL). Observa que un documento podría tener múltiples hojas de estilo que podrían ser usadas por diferentes usuarios.

Nota:

Un documento bien formateado es manejable si el documento es simple, o cuando se procesan estructuras sencillas a través de la red cuando no hay problemas de ancho de banda, ya que no tiene la sobrecarga de un DTD complejo.

XML contra HTML

XML y HTML son lenguajes de marcas, donde las etiquetas se usan para anotar los datos. Las principales diferencias son:

- En HTML, la síntaxis y la semántica del documento está definidas. HTML sólo se puede usar para crear una representación visible para el usuario. XML permite definir síntaxis de documentos.
- Los documentos HTML no están bien formateados. Por ejemplo, no todas las etiquetas tiene su correspondiente etiqueta de cierre. Los

documentos XML están bien formateados.

- Los nombres de las etiquetas son sensibles a las mayúsculas en XML, pero no en HTML.

Miremos un documento XML. El ejemplo siguiente es un documento de ejemplo XML, desde un servidor ficticio de cotizaciones, representa una lista de stocks. Como podemos ver, hay un elemento raíz (`portfolio`), y cada elemento tiene una etiqueta de inicio y una etiqueta de cierre..

Ejemplo 1: strocks.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<portfolio>
  <stock>
    <symbol>SUNW</symbol>
    <name>Sun Microsystems</name>
    <price>17.1</price>
  </stock>
  <stock>
    <symbol>AOL</symbol>
    <name>America Online</name>
    <price>51.05</price>
  </stock>
  <stock>
    <symbol>IBM</symbol>
    <name>International Business Machines</name>
    <price>116.10</price>
  </stock>
  <stock>
    <symbol>MOT</symbol>
    <name>MOTOROLA</name>
    <price>15.20</price>
  </stock>
</portfolio>
```

La primera línea indica el número de versión de XML, que es 1; y le permite al procesador conocer qué esquema de códificación se va a utilizar "UTF-8".

Aunque es un documento sencillo contiene información útil que puede ser procesada fácilmente porque está bien estructurada. Por ejemplo, el servidor de stocks podría querer ordenar el `portfolio`, en un orden específico, basándose en el precio de los stocks.

Presentar Documentos XML

Los documentos XML contienen datos portables. Esto significa que el ejemplo 1 puede procesarse como salida para diferentes navegadores (navegadores de escritorio para PC, micronavegadores para dispositivos de mano). En otras palabras, un documento XML puede ser transformado en HTML o WML o cualquier otro lenguaje de marcas.

Si cargamos el documento **stocks.xml** en un navegador que soporte XML (como IE de Microsoft), veríamos algo similar a la Figura 1:

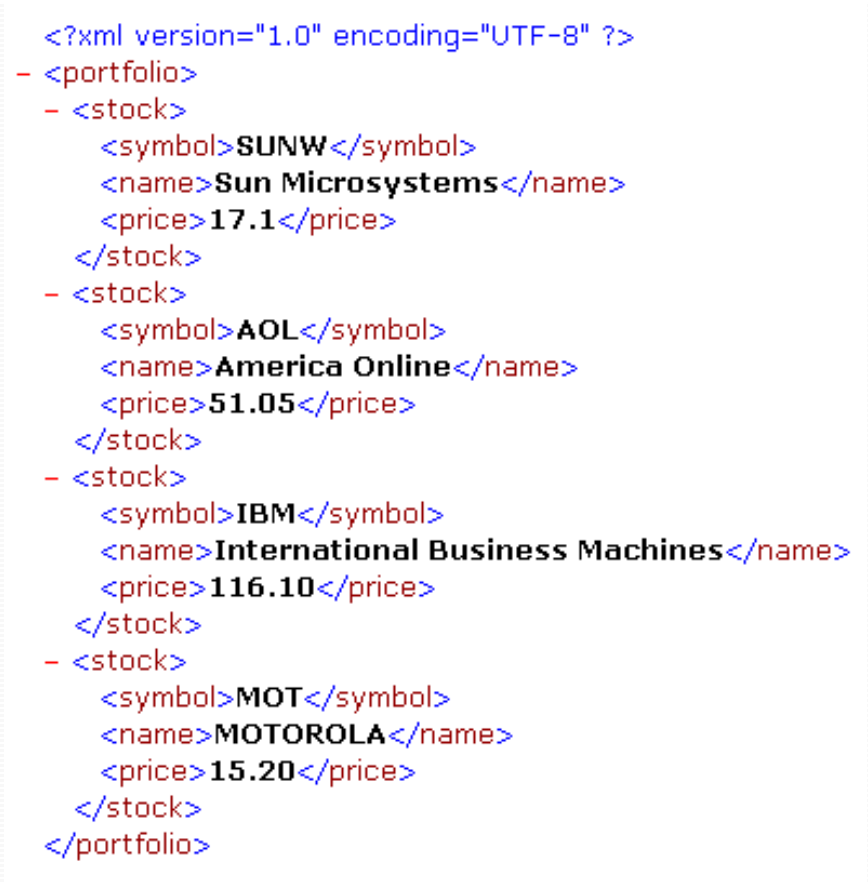


Figura 1: stocks.xml en un navegador

Básicamente, el navegador ha analizado el documento y lo ha mostrado de una forma estructurada. Pero, esto no es realmente útil desde el punto de vista de un usuario. Los usuarios desean ver los datos mostrados como información útil de una forma que sea fácil de navegar.

Una forma agradable para mostrar documentos XML es aplicar una transformación sobre el documento XML, para extraer los datos o para crear un nuevo formato (como transformar datos XML a HTML). Esta transformación se puede hacer usando un lenguaje de transformación como "Extensible Stylesheet Language Transformation" (XSLT), que forma parte de XSL. XSL permite que escribamos el vocabulario XML para especificar la semántica del formato. Es decir hay dos partes de XSL, que son parte de la actividad de estilo del World Wide Web Consortium (W3C).

- Lenguaje de Transformación (XSLT)
- Lenguaje de Formateo (objetos de formateo XSL)

La hoja de estilos XSL del Ejemplo 2 realiza la transformación rquerida para el elemento `portfolio`. Genera marcas HTML y extrae datos de los elementos del documento **stocks.xml**.

Ejemplo 2: stocks.xsl

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0" xmlns:xsl=
"http://www.w3.org/TR/WD-xsl">

<xsl:template match="/">
  <html>
  <head>
  <title>Stocks</title>
  </head>
  <body bgcolor="#ffffcc" text="#0000ff">
    <xsl:apply-templates/>
  </body>
  </html>
</xsl:template>

<xsl:template match="portfolio">

   <table border="2" width="50%">
     <tr>
      <th>Stock Symbol</th>
      <th>Company Name</th>
      <th>Price</th>
     </tr>
     <xsl:for-each select="stock">
       <tr>
         <td>
           <i><xsl:value-of select=
           "symbol"/></i>
         </td>
       <td>
          <xsl:value-of select="name"/>
        </td>
       <td>
          <xsl:value-of select="price"/>
       </td>

        </tr>
      </xsl:for-each>
    </table>
  </xsl:template>

</xsl:stylesheet>
```

Parece un poco complejo , ¿no? Sola al principio, una vez que conozcamos la síntaxis XSL es bastante sencillo. Aquí lo tenemos todo sobre la síntaxis de arriba:

- **xsl:stylesheet**: elemento raíz
- **xsl:template**: cómo transformar los nodos seleccionados

- **match**: atributo para seleccionar un nodo
- **"/"**: nodo raíz del documento XML de entrada
- **xsl:apply-templates**: aplica las plantillas a los hijos del nodo seleccionado
- **xsl:value-of**: nodo seleccionados (extrae datos de los nodos seleccionados)

Ahora la cuestión es: ¿cómo usar esta hoja de estilos XSL con el documento **stocks.xml**? La respuesta es sencilla, necesitamos modificar la primera línea del documento **stocks.xml** del Ejemplo 1 para usar la hoja de estilos **stocks.xsl** para su representación. La primera línea del ejemplo 1 ahora debería ser:

```
<?xml:stylesheet type="text/xsl" href="stocks.xsl" version="1.0" encoding="UTF-8"?>
```

Esto dice básicamente que cuando cargamos **stocks.xml** en un navegador (será analizado como un árbol de nodos), se debe utilizar la hoja de estilos correspondiente para extraer datos de los nodos del árbol. Cuando cargamos el fichero **stocks.xml** modificado en un navegador que soporta XML y XSL (como IE de Microsoft), veremos algo similar a la figura 2.:

Stock Symbol	Company Name	Price
SUNW	Sun Microsystems	17.1
AOL	America Online	51.05
IBM	International Business Machines	116.10
MOT	MOTOROLA	15.20

Figura 2: Usando una hoja de estilo (stocks.xsl)

Generar XML desde JSP

Se puede usar la tecnología JSP para generar documentos XML. Una página JSP podría generar fácilmente una respuesta conteniendo el documento **stocks.xml** del Ejemplo 1, como se ve en el ejemplo 3. El principal requerimiento para generar XML es que la página JSP seleccione el tipo de contenido de la página de forma apropiada. Como podemos ver a partir del ejemplo 3, el tipo de contenido se selecciona a `text/xml`. Se puede usar la misma técnica para generar otros lenguajes de marcas (como WML).

Ejemplo 3: genXML.jsp

```
<%@ page contentType="text/xml" %>
<?xml version="1.0" encoding="UTF-8"?>
<portfolio>
```

```
  <stock>
    <symbol>SUNW</symbol>
    <name>Sun Microsystems</name>
    <price>17.1</price>
  </stock>
  <stock>
    <symbol>AOL</symbol>
    <name>America Online</name>
    <price>51.05</price>
  </stock>
  <stock>
    <symbol>IBM</symbol>
    <name>International Business
    Machines</name>
    <price>116.10</price>
  </stock>
  <stock>
    <symbol>MOT</symbol>
    <name>MOTOROLA</name>
    <price>15.20</price>
  </stock>
</portfolio>
```

Si solicitamos la página **genXML.jsp** desde un servidor Web (como Tomcat), veríamos algo similar a la figura 1.

Generar XML desde JSP y JavaBeans

Los datos para XML también pueden recuperarse desde un componente JavaBean. Por ejemplo, el siguiente componente JavaBean, `PortfolioBean`, define un bean con datos de stocks:

Ejemplo 4: PortfolioBean.java

```
package stocks;

import java.util.*;

public class PortfolioBean implements
java.io.Serializable {
   private Vector portfolio = new Vector();

   public PortfolioBean() {
      portfolio.addElement(new Stock("SUNW",
       "Sun Microsystems", (float) 17.1));
      portfolio.addElement(new Stock("AOL",
       "America Online", (float) 51.05));
      portfolio.addElement(new Stock("IBM",
       "International Business Machines",
        (float) 116.10));
      portfolio.addElement(new Stock("MOT",
       "MOTOROLA", (float) 15.20));
```

```
   }

   public Iterator getPortfolio() {
      return portfolio.iterator();
   }
}
```

La clase `PortfolioBean` usa la clase `Stock` que se muestra en el Ejemplo 5.

Ejemplo 5: Stock.java

```
package stocks;

public class Stock implements java.io.Serializable {
   private String symbol;
   private String name;
   private float price;

   public Stock(String symbol, String name,
    float price) {
      this.symbol = symbol;
      this.name = name;
      this.price = price;
   }

   public String getSymbol() {
      return symbol;
   }

   public String getName() {
      return name;
   }

   public float getPrice() {
      return price;
   }
}
```

Ahora, podemos escribir una página JSP para generar un documento XML donde los datos son recuperados desde el `PortfolioBean`, como se ve en el Ejemplo 6.

Ejemplo 6: stocks.jsp

```
<%@ page contentType="text/xml" %>
<%@ page import="stocks.*" %>

<jsp:useBean id="portfolio"
class="stocks.PortfolioBean" />
```

```
<%
java.util.Iterator folio =
portfolio.getPortfolio();
Stock stock = null;
%>

<?xml version="1.0" encoding="UTF-8"?>
<portfolio>
  <% while (folio.hasNext()) { %>
    <% stock = (Stock)folio.next(); %>
    <stock>
      <symbol<>%=
      stock.getSymbol() %></symbol>
      <name<>%=
      stock.getName() %></name>
      <price<>%=
      stock.getPrice() %></price>
    </stock>
  <% } %>
</portfolio>
```

Si solicitamos la página **stocks.jsp** desde un navegador Web que soporte XML, obtendríamos algo similar a la Figura 3.

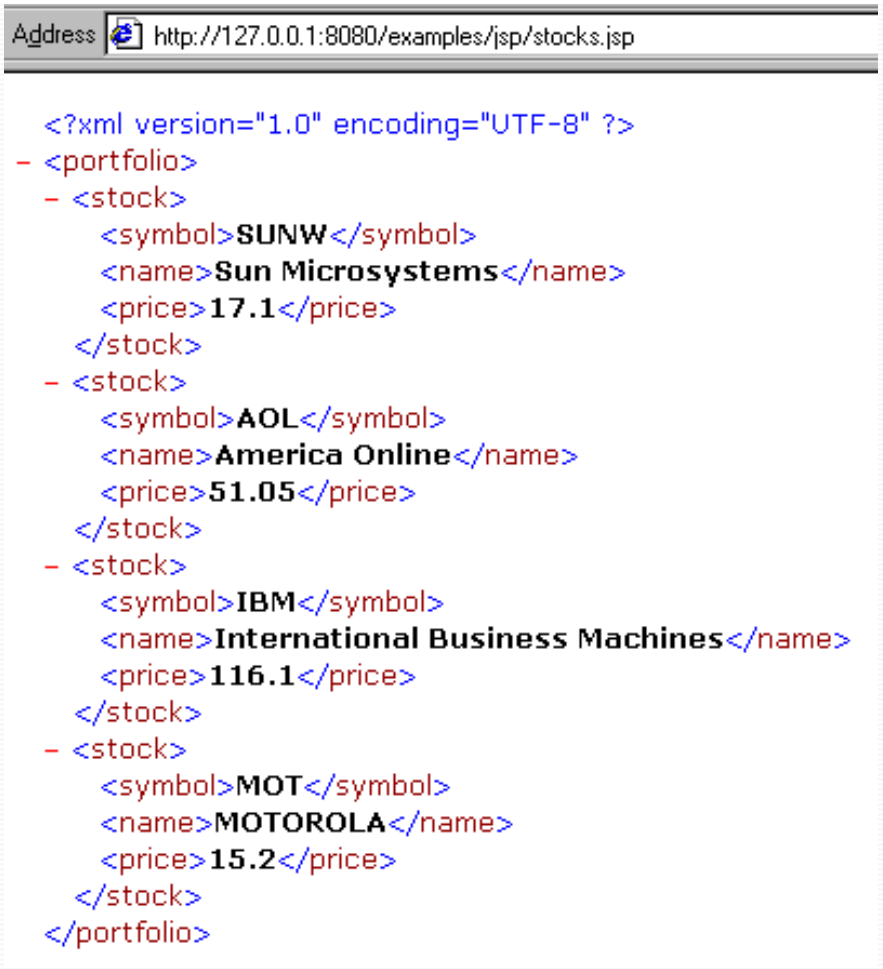


Figure 3: Generar XML desde JSP y JavaBeans

Si reemplazamos la línea:

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

con una línea que especifique una hoja de estilos:

```
<?xml:stylesheet type="text/xsl" href="stocks.xsl" version="1.0" encoding="UTF-8"?>
```

El documento XML será generado y se aplicará la hoja de estilo XSL y veremos algo similar a la figura 2 anterior.

Convertir XML a Objetos del Lado del Servidor

Hemos visto cómo generar XML, así que la pregunta ahora es cómo consumirlo (o usarlo) en aplicaciones. Para poder hacer todo lo que necesitamos para convertir XML en objetos del lado del servidor y extraer las propiedades del objeto. La conversión no es automática; tenemos que analizar manualmente un documento XML, y encapsularlo dentro de un componente JavaBeans. En el futuro sin embargo, la tecnología de unión XML/Java automatizará este proceso pues permitirá compilar un esquema XML en clases Java.

Para analizar se pueden usar dos interfaces:

- Simple API for XML (SAX)
- Document Object Model (DOM)

Antes de introducirnos en estas técnica de análisis, primero describiremos el entorno de software.

El Entorno de Software

El entorno de software que usaremos para analizar es el API para Prorcesamiento de XML (JAXP) versión 1.1 que soporta SAX , DOM level 2, y XSL transformations. JAXP puede ser descargado desde aquí.

Para instalarlo, lo descomprimimos en un directorio de nuestro elección, y actualizamos el `classpath` para incluir el árbol de fichero JAR del JAXP.

- `crimson.jar`: el analizador XML por defecto, que fue derivado del analizador "Java Project X " de Sun
- `xalan.jar`: El motor XSLT por defecto
- `jaxp.jar:` los APIs

Alternativamente, podemos instalar estos ficheros JAR como extensiones de Java 2 simplemente copiándolos en el directorio `JAVA_HOME/jre/lib/ext`, donde **JAVA_HOME** es el directorio donde instalamos el JDK (por ejemplo c:\jdk1.3). Instalar los ficheros JAR como extensiones de Java 2 elimina la necesidad de modificar el `classpath`.

API Simple para XML (SAX)

SAX es un sencillo API para XML. No es un analizador! Simplemente es un interface estándar, implementado por muchos y diferentes analizadores XML, que fue desarrollado por los miembros de la XML-DEV mailing list, actualmente hospedada por OASIS.

La ventaja principal de SAX es que es ligero y rápido. Esto es principalmente porque es un API basado en eventos, lo que significa que reporta eventos de análisis (como el comienzo y el final de los elementos) directamente a la aplicación usando servicios repetidos, según lo mostrado en la Figura 4. Por lo tanto, la aplicación implementa manejadores para ocuparse de los diversos eventos, como el manejo de eventos en un interface gráfico de usuario.

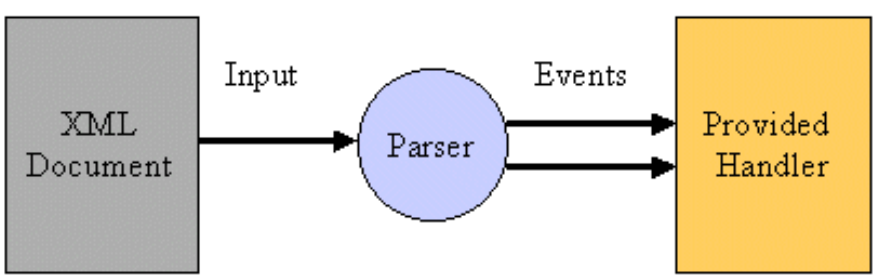


Figura 4: SAX usa retrollamadas para notificar a los manejadores las cosas de interés

Usar SAX implica los siguientes pasos, mostrados en le Figura 7.

- Implementar uno o más manejadores (en este ejemplo se implementa el `ContentHandler`)
- Crear un `XMLReader`
- Crear un `InputSource`
- Llamar a `parse` sobre la fuente de entrada

Observa que `MySAXParserBean` sobreescribe los métodos `startElement`, `endElement`, y `characters`, todos los cuales están definidos por el interface `ContentHandler`. El analizador llama al método `startElement` al principio de cada elemento del documento XML, y llama al método `characters` para reportar cada dato de caracter, y finalmente llama al método `endElement` al final de cada elemento del documento XML.

Ejemplo 7: MyParserBean.java

```
package saxbean;

import java.io.*;
import java.util.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParserFactory;
import
javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;

public class MySAXParserBean extends
DefaultHandler implements java.io.Serializable {
   private String text;
   private Vector vector = new Vector();
   private MyElement current = null;

   public MySAXParserBean() {
   }
```

```
  public Vector parse(String filename) throws
  Exception {
    SAXParserFactory spf =
     SAXParserFactory.newInstance();
    spf.setValidating(false);
    SAXParser saxParser = spf.newSAXParser();
    // create an XML reader
    XMLReader reader = saxParser.getXMLReader();
    FileReader file = new FileReader(filename);
    // set handler
    reader.setContentHandler(this);
    // call parse on an input source
    reader.parse(new InputSource(file));
    return vector;
  }

  // receive notification of the beginning of an
  element
  public void startElement (String uri, String name,
   String qName, Attributes atts) {
     current = new MyElement(
     uri, name, qName, atts);
     vector.addElement(current);
     text = new String();

  }

  // receive notification of the end of an element
  public void endElement (String uri, String name,
  String qName) {
    if(current != null && text != null) {
       current.setValue(text.trim());
    }
    current = null;
  }

  // receive notification of character data
  public void characters (char ch[], int start,
  int length) {
    if(current != null && text != null) {
       String value = new String(
       ch, start, length);
       text += value;
    }
  }
}
```

`MySAXParserBean` está usando la clase `MyElement`, que se define en el ejemplo 8.

Ejemplo 8: MyElement.java

```
package saxbean;

import org.xml.sax.Attributes;

public class MyElement implements
java.io.Serializable {
   String uri;
   String localName;
   String qName;
   String value=null;
   Attributes attributes;

   public MyElement(String uri, String localName,
    String qName, Attributes attributes) {
      this.uri = uri;
      this.localName = localName;
      this.qName = qName;
      this.attributes = attributes;
   }

   public String getUri() {
      return uri;
   }

   public String getLocalName() {
      return localName;
   }

   public String getQname() {
      return qName;
   }

   public Attributes getAttributes() {
      return attributes;
   }

   public String getValue() {
      return value;
   }

   public void setValue(String value) {
      this.value = value;
   }
}
```

Ahora, si queremos, podemos probar el `MySAXParserBean` desde la línea de comando para asegurarnos de que funciona. El ejemplo 9 muestra una sencilla clase de prueba:

Ejemplo 9: MyTestDriver.java

```
import java.io.*;
import java.util.*;

public class MyTestDriver {

   public static void main(String argv[]) {
      String file = new String(argv[0]);
      MySAXParserBean p = new MySAXParserBean();
      String str = null;
      try {
         Collection v = p.parse(file);
         Iterator it = v.iterator();
         while(it.hasNext()) {
            MyElement element =
            (MyElement)it.next();
            String tag = element.getLocalName();

            if(tag.equals("symbol")) {
              System.out.println("Symbol.
              " + element.getValue());
            } else if(tag.equals("name")) {
              System.out.println("Name: "
              +element.getValue());
            } else if (tag.equals("price")) {
              System.out.println("Price: "
              +element.getValue());
            }
         }
      } catch (Exception e) {
      }
   }
}
```

Si ejecutamos `MyTestDriver` proporcionándole el documento XML del ejemplo 1 "**stocks.xml**", veremos algo similar a la Figura 5.

```
C:\jaxp-1.1>java MyTestDriver stocks.xml
Symbol: SUNW
Name: Sun Microsystems
Price: 17.1
Symbol: AOL
Name: America Online
Price: 51.05
Symbol: IBM
Name: International Business Machines
Price: 116.10
Symbol: MOT
Name: MOTOROLA
Price: 15.20
```

Figura 5: Probar el analizador XML desde la línea de comandos.

Ahora, veamos cómo usar el `MySAXParserBean` desde una página JSP. Como podemos ver desde el Ejemplo 10, es sencillo. LLamamos al método `parse` de `MySAXParserBean` sobre el documento XML (**stocks.xml**), y luego iteramos sobre los stocks para extraer los datos y formatearlos en una tabla HTML.

> **Nota**:
>
> El documento XML de entrada al método `parse` pueder ser una URL que referencie a un documento XML (como http://www.host.com/xml/ebiz.xml).

Ejemplo 10: saxstocks.jsp

```
<html>
<head>
<title>sax parser</title>
<%@ page import="java.util.*" %>
<%@ page import="saxbean.*" %>
</head>

<body bgcolor="#ffffcc">

<jsp:useBean id="saxparser"
class="saxbean.MySAXParserBean" />

<%
Collection stocks =
saxparser.parse("c:/stocks/stocks.xml");
Iterator ir = stocks.iterator();
%>

<center>
<h3>My Portfolio</h3>
<table border="2" width="50%">
 <tr>
 <th>Stock Symbol</th>
 <th>Company Name</th>
 <th>Price</th>
 </tr>
 <tr>

<%
while(ir.hasNext()) {
  MyElement element = (MyElement) ir.next();
  String tag = element.getLocalName();
  if(tag.equals("symbol")) { %>
    <td><%= element.getValue() %></td>
  <% } else if (tag.equals("name")) { %>
    <td><%= element.getValue() %></td>
  <% } else if (tag.equals("price")) { %>
```

```
    <td><%= element.getValue() %><
    /td></tr><tr>
  <% } %>
<% } %>

</body>
</html>
```

Si solicitamos **saxstocks.jsp** veremos algo similar a la Figura 6.

My Portfolio

Stock Symbol	Company Name	Price
SUNW	Sun Microsystems	17.1
AOL	America Online	51.05
IBM	International Business Machines	116.10
MOT	MOTOROLA	15.20

Figura 6: Usar MySAXParserBean desdeJSP

Document Object Model (DOM)

DOM es un interface neutral a la plataforma - y al lenguaje- para acceder y actualizar documentos. Sin embargo, al contrario que SAX, DOM accede a un documento XML a través de una estructura arborescente, compuesta por nodos elemento y nodos de texto, según lo mostrado en la figura 7.

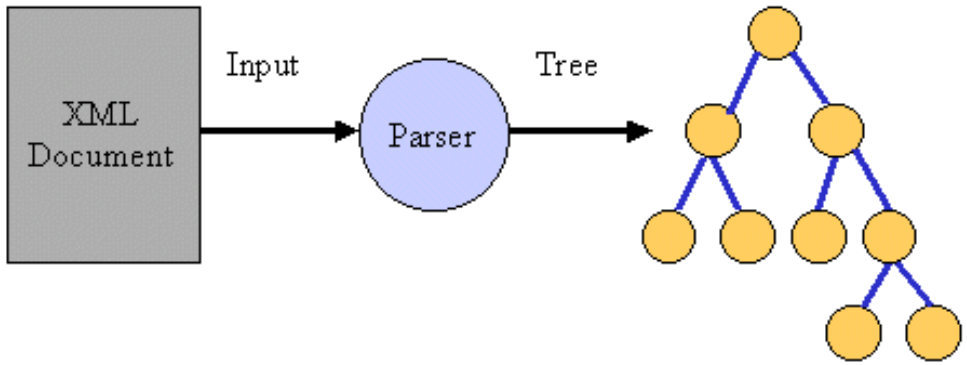


Figura 7: DOM crea un árbol desde un documento XML.

El árbol será construido en memoria y por lo tanto se consumen grandes requisitos de memoria al usar DOM. Sin embargo, la ventaja de DOM es que es más simple de programar que SAX. Como podemos ver desde el ejemplo 11, un documento XML se puede convertir en un árbol con tres líneas de código. Una vez que tengamos un árbol, podemos recorrerlo o atravesarlo hacia adelante y hacia atrás.

Ejemplo 11: MyDOMParserBean.java

```
package dombean;

import javax.xml.parsers.*;
import org.w3c.dom.*;

import java.io.*;

public class MyDOMParserBean
implements java.io.Serializable {
   public MyDOMParserBean() {
   }

   public static Document
   getDocument(String file) throws Exception {

    // Step 1: create a DocumentBuilderFactory
     DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

    // Step 2: create a DocumentBuilder
     DocumentBuilder db = dbf.newDocumentBuilder();

    // Step 3: parse the input file to geta Document object
     Document doc = db.parse(new File(file));
     return doc;
   }
}
```

Esto producirá un árbol, por eso necesitamos atraversarlo, Esto se hace usando el método `traverseTree` en la página JSP, **domstocks.jsp**, en el ejemplo 12.

Ejemplo 12: domstocks.jsp

```
<html>
<head>
<title>dom parser</title>
<%@ page import="javax.xml.parsers.*" %>
<%@ page import="org.w3c.dom.*" %>
<%@ page import="dombean.*" %>
</head>

<body bgcolor="#ffffcc">

<center>
<h3>My Portfolio</h3>
```

```
<table border="2" width="50%">
 <tr>
 <th>Stock Symbol</th>
 <th>Company Name</th>
 <th>Price</th>
 </tr>

<jsp:useBean id="domparser" class="dombean.MyDOMParserBean" />

<%
Document doc = domparser.getDocument("c:/stocks/stocks.xml");
traverseTree(doc, out);
%>

<%! private void traverseTree(Node node,
 JspWriter out) throws Exception {
        if(node == null) {
           return;
        }
        int type = node.getNodeType();

        switch (type) {
           // handle document nodes
           case Node.DOCUMENT_NODE: {
             out.println("<tr>");
             traverseTree(((Document)node).getDocumentElement(), out);
             break;
          }
          // handle element nodes
          case Node.ELEMENT_NODE: {
            String elementName = node.getNodeName();
             if(elementName.equals("stock")) {
               out.println("</tr><tr>");
             }
             NodeList childNodes = node.getChildNodes();
             if(childNodes != null) {
                int length = childNodes.getLength();
                for (int loopIndex = 0; loopIndex <
                length ; loopIndex++)
                {
                   traverseTree
                   (childNodes.item(loopIndex),out);
                 }
              }
              break;
           }
           // handle text nodes
           case Node.TEXT_NODE: {
              String data =
              node.getNodeValue().trim();

              if((data.indexOf("\n")  <0) && (data.length() > 0)) {
```

```
                out.println("<td>"+
                data+"</td>");
              }
            }
          }
    }

%>

</body>
</html>
```

Si solicitamos **domstocks.jsp** desde un navegador, veríamos algo similar a la figura 6 de arriba.

Transformar XML

Como que los navegadores están llegando a estar disponibles en más dispositivos, la habilidad de transformar datos de Internet en múltiples formatos, como XML, HTML, WML, o XHTML, se está convirtiendo cada vez en más importante. El XSLT se puede utilizar para transformar XML en cualquier formato deseado. Un motor de transformación o un procesador tomaría un documento XML como entrada y utilizaría la hoja de estilo de transformación XSL para crear un nuevo formato de documento, según lo mostrado en la figura 8.

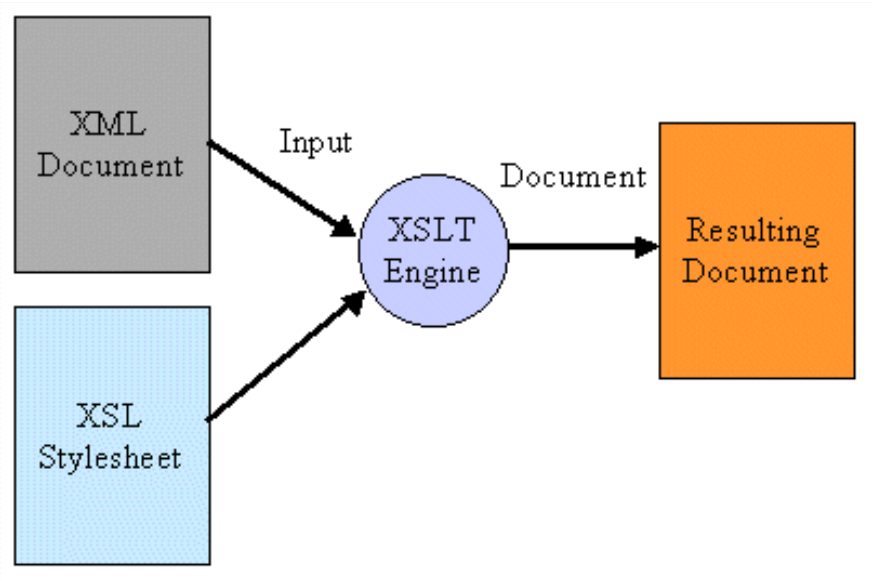


Figura 8: Usar un motor XSLT

El ejemplo siguiente, **xml2html**, muestra cómo transformar el documento XML, **stocks.xml,** en un documento HTML, **stocks.html**. Como podemos ver, una vez que tengamos el documento XML y la hoja de estilo XSL a aplicar, toda la transformación se hace en tres líneas.

Ejemplo 13: xml2html.java

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import java.io.*;

public class xml2html {
        public static void main(String[] args)
    throws TransformerException,
    TransformerConfigurationException,
           FileNotFoundException, IOException
  {
        TransformerFactory tFactory = TransformerFactory.newInstance();
        Transformer transformer = tFactory.newTransformer(new StreamSource("stocks.xsl"));
        transformer.transform(new StreamSource(args[0]), new StreamResult(new FileOutputStream(args[1])));
        System.out.println("** The output is written in "+ args[1]+" **");
  }
}
```

Para ejecutar este ejemplo, usamos el comando:

```
C:> java xml2html stocks.xml stocks.html
```

stocks.xml es el fichero de entrada, y será transformado en **stocks.html** basándose en la hoja de estilos **stocks.xsl**. Aquí estamos utilizando la hoja de estilo anterior, pero hemos agregado un par de nuevas líneas para especificar el método de salida (HTML en este caso) según lo mostrado en el ejemplo 14.

Ejemplo 14: stocks2.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl=
"http://www.w3.org/1999/XSL/Transform" version=
"1.0">

<xsl:output method="html" indent="yes"/>

<xsl:template match="/">
  <html>
  <body>
   <xsl:apply-templates/>
  </body>
  </html>
</xsl:template>

<xsl:template match="portfolio">
   <table border="2" width="50%">
     <xsl:for-each select="stock">
       <tr>
         <td>
           <i><xsl:value-of select=
           "symbol"/></i>
         </td>
```

```
        <td>
           <xsl:value-of select="name"/>
         </td>
        <td>
           <xsl:value-of select="price"/>
        </td>

         </tr>
       </xsl:for-each>
     </table>
  </xsl:template>
</xsl:stylesheet>
```

Se generarán las etiquetas HTML como se muestra en la Figura 9:

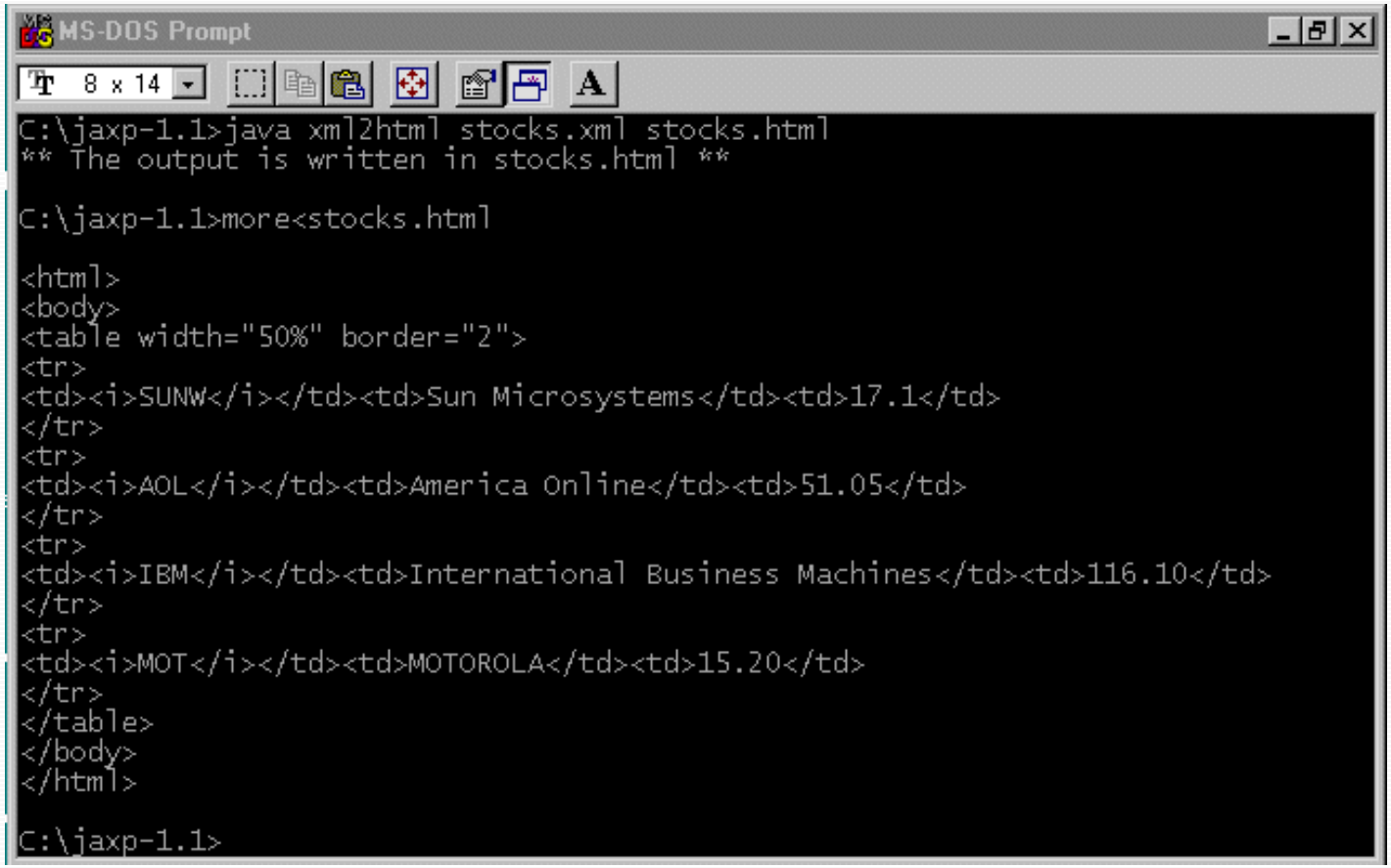


Figura 9: Transformación xml2html

Indice de contenidos

Desarrollo de Aplicaciones Web con JSP y XML

Autor: Sun
Traductor: Juan Antonio Palos (Ozito)

Parte III: Desarrollar Etiquetas JSP Personalizadas

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

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

Está página presenta una breve introducción a las etiquetas personalizadas, presenta:

- Cómo desarrollar y desplegar etiquetas sencillas.
- Cómo desarrollar y desplegar etiquetas avanzadas: etiquetas parametrizadas y con cuerpo.
- Cómo describir etiquetas con el "Tag Library Descriptor" (TLD)

Finalmente, se proporcionarán algunas guías para la programación.

Introducción a las Etiquetas

Si tenemos experiencia con HTML, ya conocemos algo sobre los tipos de etiquetas que se pueden usar. Básicamente hay dos tipo de etiquetas, y ámbos pueden tener atributos (información sobre cómo la etiqueta debería hacer su trabajo):

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

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

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

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

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

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

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

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

Etiquetas JSP Personalizadas

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

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

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

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

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

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

O podría escribirse como una etiqueta con cuerpo:

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

Nota:

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

Beneficios de las Etiquetas Personalizadas

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

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

En breve, podemos usar etiquetas personalizadas para realizar tareas complejas de la misma forma que utilizamos HTML para crear una representación.

Definir una Etiqueta

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

desarrollo de etiquetas con cuerpo se discute más adelante. El ejemplo 1 muestra el código fuente del interface `Tag` que debemos implementar:

Ejemplo 1: `Tag.java`

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

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

Todas las etiquetas deben implementar el interface `Tag` (o uno de sus interfaces) como si definiera todos los métodos que el motor de ejecución JSP llama para ejecutar una etiqueta. La Tabla 1 proporciona una descripción de los métodos del interface `Tag`:

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

Mi Primera Etiqueta

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

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

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

1. Desarrollar el Controlador de Etiqueta

Un **controlador de etiqueta** es un objeto llamado por el motor de ejecución JSP para evaluar la etiqueta personalizada durante la ejecución de una página JSP que referencia la etiqueta. Los métodos del controlador de etiqueta son llamados por la clase de implementación en varios momentos durante la evaluación de la etiqueta. Cada controlador de etiqueta debe implementar un interface especializado. En este ejemplo, la etiqueta implementa el interface `Tag` como se muestra en el ejemplo 2:

Ejemplo 2: `HelloTag.java`

```
package tags;

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

public class HelloTag implements Tag {
   private PageContext pageContext;
   private Tag parent;

   public HelloTag() {
super();
```

```
   }

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

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

   public void release() {
   }

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

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

   public Tag getParent() {
return parent;
   }
}
```

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

Para compilar la clase `HelloTag`, asumiendo que **Tomcat** está instalado en **c:\tomcat**:

- Creamos un nuevo subdirectorio llamando **tags**, que es el nombre del paquete que contiene la clase `HelloTag`. Este debería crearse en: **c:\tomcat\webapps\examples\web-inf\classes**.
- Grabamos **HelloTag.java** en el subdirectorio **tags**.
- Lo compilamos con el comando:

```
c:\tomcat\webapps\examples\web-inf\classes\tags>
javac -classpath c:\tomcat\lib\servlet.jar
HelloTag.java
```

2. Crear el Descriptor de Librería de Etiquetas

El siguiente paso es especificar cómo ejecutará la etiqueta el motor de ejecución JSP . Esto puede hacerse crerando un "Tag Library Descriptor" (TLD), que es un documento XML. El ejemplo 3 muestra un TLD de ejemplo:

Ejemplo 3: `mytaglib.tld`

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

<!-- a tag library descriptor -->

<taglib>
   <tlibversion>1.0</tlibversion>
   <jspversion>1.1</jspversion>
   <shortname>first</shortname>
   <uri></uri>
   <info>A simple tab library for the
   examples</info>
```

```
  <tag>
    <name>hello</name>
    <tagclass>tags.HelloTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Say Hi</info>
  </tag>
</taglib>
```

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

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

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

Grabamos **mytaglib.tld** en el directorio: **c:\tomcat\webapps\examples\web-inf\jsp**.

3. Probar la Etiqueta

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

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

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

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

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

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

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

Ahora, escribimos un JSP y usamos la primera síntaxis. Lo podremos ver en el ejemplo 4:

Ejemplo 4: `Hello.jsp`

```
<%@ taglib uri="/WEB-INF/jsp/mytaglib.tld"
 prefix="first" %>
<HTML>
<HEAD>
<TITLE>Hello Tag</TITLE>
</HEAD>

<BODY bgcolor="#ffffcc">

<B>My first tag prints</B>:

<first:hello/>

</BODY>
</HTML>
```

El `taglib` se usa para decirle al motor de ejecución JSP donde encontrar el descriptor para nuestra etiqueta, y el `prefix` especifica cómo se referirá a las etiquetas de esta librería. Con esto correcto, el motor de ejecución JSP reconocerá cualquier uso de nuestra etiqueta a lo largo del JSP, mientras que la precedamos

con el prefijo `first` como en `<first:hello/>`.

Alternativamente, podemos usar la segunda opción de referencia creando un fichero JAR. O podemos usar la tercera opción simplemente reemplazando la primera línea del ejemplo 4 con la siguiente línea:

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

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

Ahora, si solicitamos **Hello.jsp** desde nuestro navegador, veríamos algo similar la la figura 1:

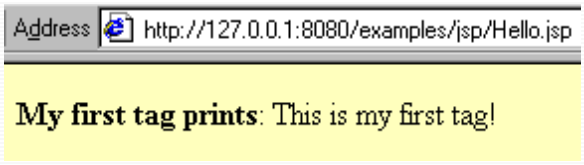


Figura 1: Primera Etiqueta Personalizada

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

Por lo tanto, la forma más fácil de escribir etiquetas sencillas es **extender** la clase `TagSupport` en vez de **implementar** el interface `Tag`. Podemos pensar en la clase abstracta `TagSupport` como en un adaptador. Habiendo dicho esto, la clase `HelloTag` del ejemplo 4 podría implementarse más fácilmente como se ve en el ejemplo 5.

Ejemplo 5: Extender la clase TagSupport

```
package tags;

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

public class HelloTag extends TagSupport {

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

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

Etiquetas Parametrizadas

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

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

Añadir un método `set` y cambiar el mensaje de salida resulta en el Ejemplo 5:

Ejemplo 5: Una etiqueta con atributo

```
package tags;

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

public class HelloTagParam extends TagSupport {
   private String name;


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


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

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

Lo siguiente que necesitamos hacer es añadir una nueva etiqueta a **mytaglib.tld**. La nueva etiqueta se muestra en el ejemplo 6. Este fragmento de código debería añadirse a **mytaglib.tld**justo antes de la línea, `</taglib>`:

Ejemplo 6: revisión de `mytaglib.tld`

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

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

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

Ahora, podemos probar la nueva etiqueta. El código fuente del ejemplo 7 muestra cómo probarla usando un atributo `name` de "JavaDuke".

Ejemplo 7: `HelloTagParam.jsp`

```
<%@ taglib uri="mytags" prefix="first" %>
<HTML>
<HEAD>
<TITLE>Hello Tag with Parameter</TITLE>
</HEAD>

<BODY bgcolor="#ffffcc">
<B>My parameterized tag prints</B>:

<P>

<first:helloparam name="JavaDuke"/>
```

```
</BODY>
</HTML>
```

Si solicitamos **HelloTagParam.jsp** desde un navegador web, veremos una salida similar a la de la Figura 2:

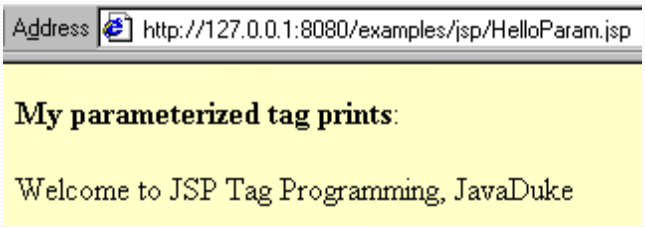


Figura 2: Probando una etiqueta parametrizada.

Librerías de Etiquetas

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

Aquí desarrollamos nuestra librería de etiquetas. Como un ejemplo, desarrollamos una sencila librería matemática que proporciona dos etiquetas, una para sumar dos números y la otra para restar un número de otro. Cada etiqueta está representada por una clase. El código fuente de las dos clases, `Add.java` y `Subtract.java`, se muestra en el ejemplo 8.

Ejemplo 8: `Add.java` y `Subtract.java`

```
package tags;

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

public class Add extends TagSupport {
   private int num1, num2;

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

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


   public int doStartTag() throws JspException {
try {
   pageContext.getOut().print("Welcome
   to First
    Grade Math! ");
   pageContext.getOut().print("The sum of: " +
   num1 + " and " + num2 + " is: " + (
   num1+num2));
} catch (IOException ioe) {
   throw new JspException("Error:
   IOException
    while writing to client");
}
return SKIP_BODY;
   }
}

// Subtract.java

package tags;

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

public class Subtract extends TagSupport {
   private int num1, num2;

   public void setNum1(int num1) {
```

```
this.num1 = num1;
   }

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


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

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

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

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

Ejemplo 9: revisar `mytaglib.tld`

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

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

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

Ahora podemos probar nuestra nueva librería de etiquetas matemáticas usando el probador mostrado en el ejemplo 10.

Ejemplo 10: `math.jsp`

```
<%@ taglib uri="mytags" prefix="math" %>
<HTML>
<HEAD>
<TITLE>Hello Tag with Parameter</TITLE>
</HEAD>

<BODY bgcolor="#ffffcc">
<B>Calling first tag</B>
<P>
<math:add num1="1212" num2="121"/>
<P>
<B>Calling second tag</B>
<P>
<math:sub num1="2123" num2="3"/>

</BODY>
</HTML>
```

Si solicitamos **math.jsp** desde un navegador web, veríamos una salida similar a la de la Figura 3:

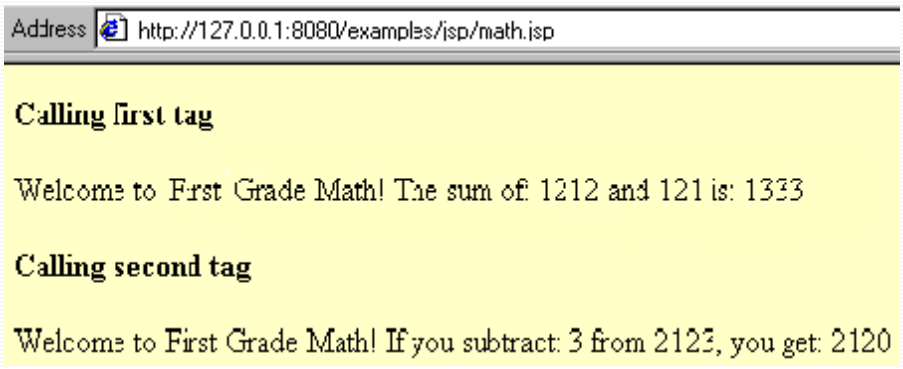


Figura 3: Probando la librería de etiquetas matemáticas.

Etiquetas con Cuerpo

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

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

Una Evaluación

Aquí tenemos un ejemplo de una sóla evaluación donde hemos extendido la clase `BodyTagSupport`. Este ejemplo lee el contenido del cuerpo, lo convierte a minúsculas, y luego lo reescribe de vuelta hacia el cliente. El ejemplo 11 muestra el código fuente. El contenido del cuerpo es recuperado como un String, convertido a minúsculas, y luego escrito de vuelta al cliente.

Ejemplo 11: `ToLowerCaseTag.java`

```
package tags;

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

public class ToLowerCaseTag extends BodyTagSupport {
```

```
   public int doAfterBody() throws JspException {
try {
   BodyContent bc = getBodyContent();
   // get the bodycontent as string
   String body = bc.getString();
   // getJspWriter to output content
   JspWriter out = bc.getEnclosingWriter();
   if(body != null) {
out.print(body.toLowerCase());
   }
} catch(IOException ioe) {
   throw new JspException("Error:
    "+ioe.getMessage());
}
return SKIP_BODY;
   }
}
```

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

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

Observa que cuando escribimos una etiqueta con cuerpo, el valor de la etiqueta `<bodycontent>` debe ser `JSP` o `jspcontent`, como se explicó anteriormente.

En el ejemplo 12, podemos ver un probador para este ejemplo:

Ejemplo 12: `lowercase.jsp`

```
<%@ taglib uri="mytags" prefix="first" %>
<HTML>
<HEAD>
<TITLE>Body Tag</TITLE>
</HEAD>

<BODY bgcolor="#ffffcc">

<first:tolowercase>
Welcome to JSP Custom Tags Programming.
</first:tolowercase>

</BODY>
</HTML>
```

Si solictamos `lowercase.jsp` desde un navegador web, veríamos algo similar a la figura 4:

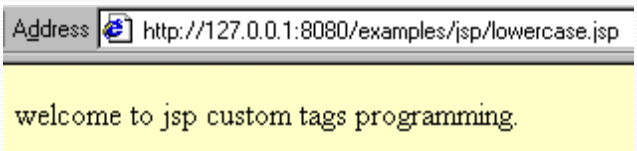


Figura 4: Probar la etiqueta `lowercase`

Multiples Evaluaciones

Ahora veamos un ejemplo de un cuerpo de etiqueta evaluado mútiples veces. El ejemplo acepta un string e imprime el string tantas veces como se indique en el JSP. El código fuente se muestra en el ejemplo 13:

Ejemplo 13: `LoopTag.java`

```
package tags;

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

```
public class LoopTag extends BodyTagSupport {

   int times = 0;

   BodyContent bodyContent;

   public void setTimes(int times) {

this.times = times;
   }

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

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

   public int doAfterBody() throws JspException {
if (times >1) {
  times--;
  return EVAL_BODY_TAG;
} else {
   return SKIP_BODY;
}
   }

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

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

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

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

```
<tag>
  <name>loop</name>
  <tagclass>tags.LoopTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>Tag with body and parameter</info>
  <attribute>
     <name>times</name>
     <required>true</required>
     <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

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

En el Ejemplo 14 podemos ver un JSP probador:

Ejemplo 14: `loops.jsp`

```
<%@ taglib uri="mytags" prefix="first" %>
<HTML>
<HEAD>
<TITLE>Body Tag</TITLE>
</HEAD>

<BODY bgcolor="#ffffcc">

<first:loop times="4">
Welcome to Custom Tags Programming.<BR>
</first:loop>

</BODY>
</HTML>
```

Finalmente, si solicitamos **loops.jsp** desde un navegador, veríamos una salida similar a la de la Figura 5:

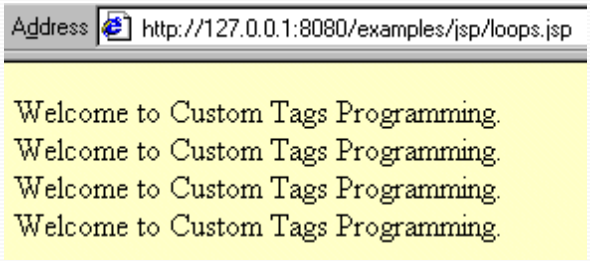


Figura 5: Probando `loops.jsp`

Guías de Programación

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

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

Conclusión

En la página anterior Parte III: JSP con XML en Mente, vimos como analizar documentos XML. Pero incluso el ojo desentrenado habrá observado que hemos embebido mucho código de análisis (o lógica) en JSP. Incluso aunque hemos usado JavaBeans para encapsular mucho código Java, todavía hemos terminado páginas JSP mezclando la lógica de programación con la presentación.

Las etiquetas personalizadas nos ayudan a mejorar la separación de la lógica del programa (análisis e iteracción de la Parte II) de la presentación. Los distintos ejemplos mostrados en esta página muestran como desarrollar y desplegar etiquetas simples y avanzadas. Como ejercicio, podrías querer reescribir los ejemplos SAX y DOM de la Parte II como librerías de etiquetas. También podrías echar un vistazo a lo que tiene que ofrecer el Jakarta Taglibs Project sobre análisis de XML y transformaciones XSL. También proporciona librerías de etiquetas para otras cosas.

Indice de contenidos

Tutoriales

Desarrollo de Aplicaciones Web con JSP y XML

Autor: Sun
Traductor: Juan Antonio Palos (Ozito)

Parte IV: Usar los Servicios de J2EE desde JSP

Java 2 Enterprise Edition (J2EE) es un estándar que define un entorno para el desarrollo y despliegue de aplicaciones empresariales. Reduce el coste y la complejidad del desarrollo de aplicaciones empresariales multi-capa y proporciona un modelo de aplicación distribuida multi-capa. En otras palabras, es enteramente distribuido y por lo tanto, las distintas partes de una aplicacion se pueden ejecutar en diferentes dispositivos.

Las aplicaciones Web desarrolladas usando JavaServer Pages (JSP) podrían requerir alguna interacción con servicios J2EE. Por ejemplo, un sistema de control de inventario basado en Web podría necesitar acceder a servicios de directorio de J2EE para obtener el acceso a una base de datos. O podríamos querer usar JavaBeans Enterprise (EJB) en nuestra aplicación.

Esta página presenta una breve introducción a J2EE, y nos muestra cómo:

- Describir servicios J2EE en un Descriptor de Desarrollo Web (**web.xml**)
- Referenciar servicios J2EE
- Acceder o usar servicios J2EE desde JSPs

Introducción a J2EE

J2EE es una plataforma estándar para desarrollar y desplegar aplicaciones empresariales. La arquitectura de J2EE, que está basada en componentes, hace muy sencillo el desarrollo de este tipo de aplicaciones porque la lógica de negocios está organizada dentro de componentes reutilizables y el servicio subyacente lo proporciona el J2EE en la forma de un contenedor por cada tipo de componente. Pensemos en un contenedor como un interface entre el componente y la funcionalidad de bajo-nivel que soporta el componente. Por lo tanto, antes de poder ejecutar un componente de una aplicación cliente, debe configurarse como un servicio J2EE y desplegarse dentro de su contenedor.

En la siguiene tabla se describen brevemente los servicios estándar incluidos en la versión 1.3 de la J2EE specification:

Servicio Estándar	Descripción
HTTP	Define el servicio HTTP donde el API del lado del cliente está definido por el paquete `java.net`, y el API del lado del servidor está definido por el API Servlet (incluyendo JSP). HTTPS, o el uso de HTTP sobre "Secure Socket Layer" (SSL), está soportado por los mismos APIs del cliente y del servidor que HTTP.
Enterprise JavaBeans (EJBs)	Un modelo de componente para desarrollo empresarial en Java.

Java Transaction API (JTA)	Un interface para componentes de aplicaciones J2EE para manejar transaciones.
RMI-IIOP	Define los APIs que permiten el uso de programación al estilo RMI que es independiente del protocolo subyacente. Las aplicaciones J2EE necesitan usar RMI-IIOP cuando acceden a componentes EJB.
Java IDL	Permite a las aplicaciones J2EE invocar a objetos CORBA usnado el protocolo IIOP.
JDBC	API para acceder a bases de datos.
Java Message Service (JMS)	API para mensajería que soporta punto-a-punto y modelos de publicación por subscripción.
Java Naming and Directory Interface (JNDI)	API para acceder a sistemas de nombres y directorios de recursos J2EE.
JavaMail	Un API estándar para enviar email.
JavaBeans Activation Framework (JAF)	Un API estándar usado por JavaMail.
Java API for XML Parsing (JAXP)	Un API estándar para analizar documentos XML. Incluye soporte para SAX y DOM.
J2EE Connector Architecture	Un API estándar para permitir la conectividad de sistemas legales y aplicaciones empresariales no-Java.
Java Authentication and Authorization Service (JAAS)	Un API estándar para permtiir que los servicios J2EE autentifiquen y fuercen el control de acceso sobre los usuarios.

J2EE promueve el desarrollo de aplicaciones multi-capa en las que el contenedor web almacena componentes web que están dedicados a menejar la lógica de presentación de una aplicación dada, y responden a las peticiones del cliente (como un navegador web). Por otro lado, el contenedor EJB, almacena componentes de aplicación que responden a las peticiones de la capa web como se muestra en la figura 1:

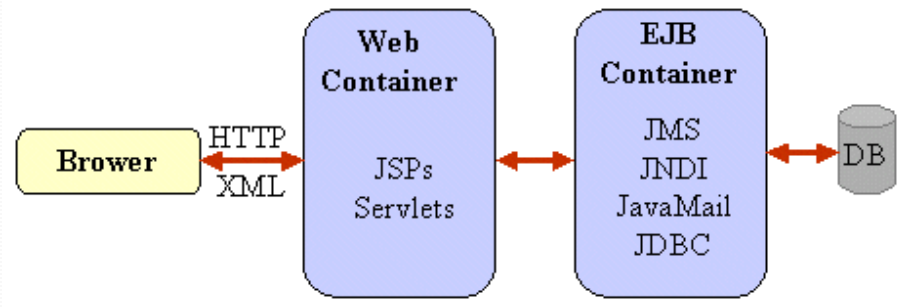


Figura 1: Aplicaciones Multi-capa

Las aplicaciones que usan esta arquitectura son escalables implícitamente. Esta arquitectura separa el acceso a los datos de las interacciones con el usuario final, y se asegura la reutilización del código basado en componentes. En la capa web, J2EE promueve el uso de JSPs para la creacción de contenido dinámico para los clientes web.

Etiquetas Personalizadas y J2EE

J2EE tiene mucho que ofrecer a los desarrolladores de aplicaciones web y a los desarrolladores de etiquetas JSP personalizadas. Como hemos podido ver de la tabla anterior, tiene un rico conjunto de APIs estándars para enviar email, acceder a bases de datos, analizar documentos XML, etc. Nuestras aplicaciones Web pueden beneficiarse de etos APIs. Por ejemplo, podemos escribir una etiqueta JSP personalizada que envíe email que puede ser facilmente utilizada por desarrolladores de contenido web que no están familiarizados con Java. Si no estás familiarizado con las etiquetas JSP personalizadas, sus beneficios y cómo crerarlas, puedes volver a la página anterior Desarrollar Etiquetas JSP Personalizadas.

EJBs

Un EJB es un componente del lado del servidor que encapsula la lógica del negocio de una aplicación. En cualquier aplicación, los beans enterprise implementan los métodos de la lógica del negocio, que pueden ser invocados por clientes remotos para acceder a los servicios importantes proporcionados por la aplicación.

Beneficios

Los EJBs simplifican el desarrollo de grandes aplicaciones empresariales seguras y distribuidas por las siguientes razones:

- **Los desarrolladores pueden concentrarse en solventar la lógica del negocio**: el contenedor EJB proporciona servicos a nivel del sistema como el control de transaciones y las autorizaciones de seguridad. Por lo tanto, los desarrolladores no tienen que preocuparse de estos problemas.
- **Clientes pequeños**: Los desarrolladores no tienen que desarrollar código para las reglas de negocio o

accesos a bases de datos; pueden concentrarse en la presentación del cliente. El resultado final es que los clientes son pequeños, y esto es especialmente importante para clientes que se ejecutan en pequeños dispositivos con recustos limitados.
- **Desarrollo rápido**: Los EJBs son componentes portables, y por lo tanto los ensambladores de aplicaciones pueden construir nuevas aplicaciones desde beans existentes. Las aplicaciones resultantes se pueden ejecutar en cualquier servidor compatible J2EE.

Componentes

Hay dos tipos principales de componentes EJB : **session** y **entity**. Un EJB de sesión se usa para realizar una tarea para un cliente, y un EJB de entidad es específico del dominio y se usa para representar un objeto de entidad del negocio que existe en un almacenamiento persistente. Sin embargo, los beans de entidad y de sesión tienen algunas diferencias que podemos ver en la siguiente tabla:

EJB de Sesión	EJB de Entidad
Transitorio	Persistente
Puede ser usado por un sólo cliente.	Puede ser usadopor muchos clientes.
No tiene identidad	Tiene una identidad (como una clave primaria)

Desarrollar EJBs

Para desarrollar EJBs, todo bean enterprise necesita:

- Un **interface remoto** que exponga los métodos que soporta bean enterprise.
- Un **interface home** que proporciona los métodos del ciclo de vida del bean enterprise.
- Una clase de implementación, incluyendo la lógica de negocio que necesite.

EJBs contra Servlets

A primera vista, los EJBs y los Servlets son tecnologías similares porque ambos son componentes distribuidos del lado del servidor. Sin embargo, hay una diferencia importante entre los dos en el tipo de solución que ofrecen; los EJBs no pueden aceptar peticiones HTTP. En otras palabras, los EJBs no peuden servir peticiones que vienen directamente desde un navegador Web, y los servlets si pueden. Servlets y JSPs se pueden usar para implementar presentación y control web, pero al contrario que los EJBs, no pueden manejar transaciones distribuidas. Los EJBs pueden ser llamados desde cualquier cliente basado en Java.

¿Cúando usar EJBs?

Los EJBs son buenos para las aplicaciones que tienen alguno de estos requerimientos:

- **Escalabilidad**: si tenemos un número creciente de usuarios, los EJBs nos permitirán distribuir los componentes de nuestra aplicación entre varias máquinas con su localización transparente para los usuarios.
- **Integridad de Datos**: los EJBs nos facilitan el uso de transaciones distribuidas.
- **Variedad de clientes**: si nuestra aplicación va a ser accedida por una variedad de clientes (como navegadores tradicionales o navegadores WAP), se pueden usar los EJBs para almacenar el modelo del negocio, y se puede usar una variedad de clientes para acceder a la misma información.

Describir y Referenciar Servicios J2EE

Ahora que hemos visto una introducción a J2EE y los EJBs, veremos cómo referenciar, acceder y usar dichos servicios. Afortunadamente, la especificación J2EE define una forma estándard para y usar sus sevicios. Los servicos J2EE pueden ser referenciados buscándolos de acuerdo a un nombre único en un directorio. Esta funcionalidad está soportada por el "Java Naming and Directory Interface" o JNDI. Para que esto funcione, cada sistema compatible J2EE proporciona un servicio JNDI llamado un **environment** (entorno) que contiene:

- Variables de entorno
- Referencias a EJBs
- Referencias a Factorías de Recursos

Variables de Entorno

El entorno de nombrado de los componentes de la aplicación permite que estos componentes sean personalizados sin tener que acceder o modificar el código fuente de dichos componentes. Cada componente de la aplicación define su propio conjunto de entrada de entorno. Todos los ejemplares de un componente de la aplicación dentro del mismo contenedor comparten las mismas entradas. Es importante observar que no está permitido que los ejemplares de los componentes de la aplicación modifiquen su entorno durante la ejecución.

Declarar Variables de Entorno

El proveedor de componentes de la aplicación debe declarar todas las entradas de entorno accedidas desde el código del componente de la aplicación. Se declaran usando la etiqueta `<env-entry>` en el descriptor de despliegue (**web.xml** en Tomcat por ejemplo). Los elementos de la etiqueta `<env-entry>` son:

- `<description>`: una descripción opcional de la entrada de entorno.
- `<env-entry-name>`: el nombre de la entrada de entorno.
- `<env-entry-type>`: el tipo de variable de entorno esperada. Puede ser uno de los siguientes tipos Java: `Boolean`, `Byte`, `Double`, `Character`, `Float`, `Integer`, `Long`, `Short`, `String`.
- `<env-entry-value>`: un valor para la entrada de entorno que debe corresponder con el tipo suministrado dentro de `<env-entry-type>`. Este valor puede modificarse posteriormente, pero si no se seecciona deberá especificarse durante el despliegue.

El código del ejemplo 1 muestra una decalración de dos entradas de entorno. Para especificar una declaración de un nuevo entorno, simplemente los añadimos a nuestro descriptor de apliación Web (**web.xml**).

Ejemplo 1: Declarar variables de entorno

```
<env-entry>
<description>welcome message</description>
<env-entry-name>greetings</env-entry-name>
<env-entry-type>java.lang.String</env-entry-type>
<env-entry-value>Welcome to the Inventory Control
   System</env-entry-value>
</env-entry>

<env-entry>
<description>maximum number of products</descriptor>
<env-entry-name>inventory/max</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-type>
<env-entry-value>27</env-entry-value>
</env-entry>
```

Cada etiqueta `<env-entry>` describe una sóla entrada de entorno. Por eso, en este ejemplo, se han definido dos variables de entorno, la primera es una llamada `greetings`, que es del tipo `String` y tiene un valor inicial por defecto de: `Welcome to the Inventory Control System`. La segunda entrada se llama `inventory/max`, y es del tipo `Integer` y tiene un valor inicial por defecto de `27`.

Ahora, un ejemplar de componente de aplicación puede localizar la entrada de entorno usando JNDI. Crea un objeto `javax.naming.InitialContext` usando el constructor sin argumentos. Busca el entorno de nombrado a través del `InitialContext` usando el URL JNDI que empieza con `java:comp/env`. El ejemplo 2 muestra cómo un componente de aplicación accede a sus entradas de entorno:

Ejemplo 2: Acceder a entradas de entorno

```
// obtain the application component's environment
// naming context
javax.naming.Context ctx = new javax.naming.InitialContext();
javax.naming.Context env = ctx.lookup("java:comp/env");

// obtain the greetings message
//configured by the deployer
String str = (String) env.lookup("greetings");

// use the greetings message
System.out.println(greetings);

// obtain the maximum number of products
//configured by the deployer
Integer maximum = (Integer) env.lookup("inventory/max");
//use the entry to customize business logic
```

Observamos que el componente de la aplicación también podría usar las entradas de entorno usando los nombres de paths completos, de esta forma:

```
javax.naming.Context ctx = new javax.naming.InitialContext();
String str = (String) ctx.lookup("java:comp/env/greetings");
```

Este fragmento de código se puede usar en un JSP como se vé en el Ejemplo 3:

ejemplo 3: Acceder a entradas de etorno desde un JSP

```
<HTML>
<HEAD>
<TITLE>JSP Example</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffcc">
<CENTER>
<H2>Inventory System</H2>
<%
```

```
javax.naming.Context ctx = new javax.naming.InitialContext();
javax.naming.Context myenv = (javax.naming.Context) t.lookup("java:comp/env");
java.lang.String s = (java.lang.String) myenv.lookup("greetings");
out.println("The value is: "+greetings);
%>
</CENTER>

</BODY>
</HTML>
```

Sin embargo, por último podrías querer acceder a entradas de entorno desde una etiqueta personalizada. Para eso, desarrollaríamos una etiqueta personalizada o una librería de etiquetas que pudiera ser reutilizada sin tener que cortar y pegar el código. Una librería personalizada puede ser fácilmente desarrollada siguiendo los pasos descritos en la página anterior Desarrollar Etiquetas JSP Personalizadas.

Referencias EJB

Un proveedor de componentes de aplicación puede referirse a los interfaces home EJB usando nombres lógicos (llamados referencias EJB) en lugar de valores de registro JNDI. Estas son referencias espciales en el entorno de nombrado de los componentes de la aplicación. Estas referencias permiten a las aplicaciones Web acceder a los EJBs de una forma personalizada.

Declarar Referencias EJB

Una referencia EJB es una entrada en el entorno del componente de la aplicación. Sin embargo, no se debe usar `<env-entry>` para declararla. En su lugar, debemos declararla usando la etiqueta `<ejb-ref>` del descriptor de despliegue web. Los elementos de la etiqueta `<ejb-ref>` son:

- `<description>`: Una descripción opcional de la entrada de referencia EJB.
- `<ejb-ref-name>`: Un nombre único de referencia EJB.
- `<ejb-ref-type>`: Especifica el tipo esperado del EJB. El valor debe ser `Session` o `Entity`.
- `<home>`: Especifica el nombre completo de la clase del interface home del EJB.
- `<remote>`: Especifica el nombre completo de la clase del interface remoto del EJB.

El ejemplo 4 muestra una declaración de una entrada de referencia EJB. Para especificar una declaración de una nueva entrada, simplemente la añadimos a nuestro descritor de aplicación web (**web.xml**).

Ejemplo 4: Declarar una referencia EJB

```
<ejb-ref>
  <description>A reference to an entity bean to
    access employee records</description>
 <ejb-ref-name>ejb/EmployeeRecord</ejb-ref-name>
 <ejb-ref-type>Entity</ejb-ref-type>
 <home>com.company.Employee.EmployeeRecordHome</home>
 <remote>com.company.Employee.
EmployeeRecordRemote</remote>
</ejb-ref>
```

Cada elemento `<ejb-ref>` describe una sola entrada de referencia EJB. Por lo tanto en este ejemplo, se ha definido una entrada que tiene el nombre `ejb/EmployeeRecord` del tipo `Entity` especificando que el interface `home` es `com.company.Employee.EmployeeRecordHome` y el interface `remote` es `com.company.Employee.EmployeeRecordRemote`.

Ahora, un ejemplar de componente de la aplicación puede localizar la entrada de referencia EJB usando JNDI. Crea un objeto `javax.naming.InitialContext` usando el constructor sin argumentos. Luego busca el entorno de nombrado a través del `InitialContext` usando el URL JNDI que empieza con `java:comp/env/ejb`. El ejemplo 5 muestra cómo un componente de aplicación obtiene el acceso a un EJB:

Ejemplo 5: Acceder a un Bean enterprise

```
// obtain the default JDNI context
javax.naming.Context ctx =
new javax.naming.InitialContext();

// look up the home interface
Object obj = ctx.lookup(
"java:comp/env/ejb/EmployeeRecord");

// Convert the object to a home interface
EmployeeRecordHome = (EmployeeRecordHome)
  javax.rmi.PortableRemoteObject.narrow(
  object, EmployeeRecordHome.class);
```

Podemos usar un código similar directamente en nuestros JSPs, o desarrollar una etiqueta personalizada para

acceder y usar EJBs.

Referencias a Factorías de Recursos

Las referencias de factorías de recursos permiten a las aplicaciones referirse a factorías de conexiones, o a objetos que crean conexions a recursos deseados, usando nombres lógicos como hemos visto en las dos secciones anteriores. Las referencias a factorías de recursos de conexión pueden ser conexiones JDBC, conexiones JMS, conexiones de email, etc. Las URLs JNDI empiezan con: `java:comp/env/jdbc`, `java:comp/env/jms`, y `java:comp/env/mail`, respectivamente.

Declarar Referencias de Factorías de Recursos

Una referencia a una factoría de recursos es una entrada en el descritor de despliegue de una aplicación web. Debe declararse usando la etiqueta `<resource-ref>`. Los elementos de esta etiqueta son:

- `<description>`: un descriptor opcional de la referencia a la factoría de recursos.
- `<res-ref-name>`: contiene el nombre de la entrada de entorno.
- `<res-ref-type>`: especifica la factoría de recursos utilizada. Algunas factorías de recursos estándars de J2EE son: `javax.sql.DataSource` para factorías de conexiones JDBC; `javax.jms.QueueConnectionFactory` y `javax.jms.TopicConnectionFactory` para conexiones JMS; y `javax.mail.Session` para factorías de conexiones JavaMail.
- `<res-auth>`: especifica el tipo de autentificación de recursos y cómo se debería realizar. Por ejemplo, si se selecciona a `Container`, el contenedor hace la autentificación usando propiedades configuradas durante el despliegue. Por otro lado, si se selecciona a `Application`, instruye al contenedor para permitir que la aplicación autentifique programáticamente.

El ejemplo 6 muestra una declaración de una referencia de factoría de recursos email. Para especificar una declaración para una nueva entrada, simplemente la añadimos a nuestro descriptor de aplicación web (**web.xml**).

Ejemplo 6: Declarar una Referencia de Factoría de Recursos

```
<res-ref>
 <description>email session reference
    factory</description>
 <res-ref-name>mail/mailsession</res-ref-name>
 <res-type>javax.mail.Session</res-type>
 <res-auth>Container</res-auth>
</res-ref>
```

Cada elemento `<res-ref>` describe una sóla entrada de referencia a una factoría de recursos. Por lo tanto, en este ejemplo se ha definido una entrada con el nombre `mail/session` del tipo `javax.mail.Session` seleccionando `<res-auth>` al `Container`.

Ahora, un ejemplar de componente de aplicación puede obtener la entrada de referencia a la factoría de recursos usando JNDI. Como con las otras entradas, crea un objeto `javax.naming.InitialContext` usando el constructor sin argumentos. Luego busca el entorno de nombrado a través del `InitialContext` usando el URL JNDI para email que empieza con `java:comp/env/mail`. El ejemplo 7 muestra cómo un componente de aplicación obtiene una referencia a una factoría de recursos para envíar un mensaje de email:

Ejemplo 7: Obtener una Referencia a una factoriía de recursos y enviar un email.

```
// obtain the initial JNDI context
javax.naming.Context ctx =
new javax.naming.InitialContext();
// perform JNDI lookup to retrieve
//the session instance
javax.mail.Session session =
(javax.mail.Session)
    ctx.lookup(
"java:comp/env/mail/mailsession");
// create a new message and set the sender,
// receiver, subject, and content of msg
javax.mail.Message msg = new
     javax.mail.internet.MimeMessage(session);
msg.setFrom("email address goes here");
msg.setRecipient(
Message.RecipientType.TO, "to email address");
msg.setSubject("JavaMail Example");
msg.setContext(
"write the content here", "text/plain");
// send message
javax.mail.Transport.send(msg);
```

De nuevo, este código puede usarse directamente en nuestros JSPs, o podemos desarrollar una librería de etiquetas personalizada para acceder y utilizar varias referencias a factorías de recursos.

Como otro ejemplo, el ejemplo 8 muestra una declaración de una factoría de recursos de base de datos:

Ejemplo 8: Declarar una Factoría de recursos para bases de datos

```
<res-ref>
  <description>database reference
 factory</description>
  <res-ref-name>jdbc/EmployeeDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</res-ref>
```

El ejemplo 9 muestra cómo un componente de aplicación obtiene una referencia a una factoría de conexiones a bases de datos y la usa:

Ejemplo 9: Obtener una referencia a una factoría de conexiones de bases de datos y utilizarla

```
// obtain the initial JNDI context
javax.naming.Context ctx =
new javax.naming.InitialContext();
// perform JDNI lookup to
//obtain database connection factory
javax.sql.DataSource ds =
(javax.sql.DataSource)
    ctx.lookup("java:comp/env/jdbc/EmployeeDB");
// Invoke factory to obtain a resource
javax.sql.Connectin conn = ds.getConnection();
// use the connection....
```

Conclusión

J2EE ofrece muchos servicios que son importantes para las aplicaciones web. Estos servicios van desde la apertura de conexiones a bases de datos usando JDBC, hasta envíar email, pasando por acceder y usar beans enterprise. Esta página junto con los programas de ejemplo, nos ha mostrado cómo acceder a los servicios J2EE desde dentro de JSPs. La incorporación de EBJs dentro de nuestros JSPs puede hacerse fácilmente, creando una solución reutilizable mendiante el desarrollo de etiquetas personalizadas.

¿Por qué no accedemos a los servicios J2EE desde Servlets? Aunque es posible hacerlo, terminaríamos escribiendo mucho código no reutilizable. Si deseamos usar servicios J2EE desde JSPs, desarrollar librerías de etiquetas personalizadas proporciona una solución reutilizable que incluso puede ser usada por desarrolladores de contenido que no tienen experiencia con Java.

Indice de contenidos

Desarrollo de Aplicaciones Web con JSP y XML

Autor: Sun
Traductor: Juan Antonio Palos (Ozito)

Parte V: Crear una Aplicación Web de E-Mail usando Librerías de Etiquetas JSP

Introducción

El API `JavaMail` es un extensión estándar de Java. Proporciona una método estrictamente independiente del protocolo de enviar y recibir emails. La arquitectura de capas de `JavaMail` permite el uso de varios protocolos de acceso a mensajes, como **POP3** e **IMAP**, y protocolos de transferencia de mensajes como **SMTP**. JavaMail Interactúa con el contenido de los mensajes a través del "JavaBeans Activation Framework" (JAF). JAF proporciona una forma uniforme de determinar el tipo de mensaje y su encapsularlo. Para más información puedes ver JavaBeans Activation Framework, en la site de Sun

Las "JavaServer Pages" (JSP) permiten a los desarrolladores crear páginas dinámicas ricas en contenido rápida y fácilmente. JSP utiliza etiquetas al estilo de XML para encapsular la lógica que genera el contenido web. Las páginas JSP separan la lógica de la página de su diseño y estructura, lo que evita el solapamiento de roles entre los diseñadores web y los programadores. Los diseñadores diseñan las páginas web y los programadores les añaden la lógica y el código. Para más información puedes ver Servlets y JSP

Este artículo explora las posibilidades de combinar estas dos tecnologías y crear una aplicación email desplegable en la web que use el API JavaMail y librerías de etiquetas JSP para su presentación. Las páginas JSP están pensadas para proporcionar un "método declarativo y céntrico de presentación del desarrollo de servlets". Un página JSP ideal no contiene ninguna línea de código o scriptles. En su lugar la funcionalidad y la lógica del negocio se ejecutan en etiquetas que o están definidas en el API o en librerías de etiquetas personalizadas. La aplicación "Email Web Application" (EWA) presentada en este artículo usa una librería de etiquetas personalizadas. Las etiquetas están implementadas usando el API JavaMail.

Especificación Funcional

EWA soporta las funcionalidades básicas de una aplicación email. como chequeo de login y envío de email. Más específicamente, se soportan las siguientes funcionalidades:

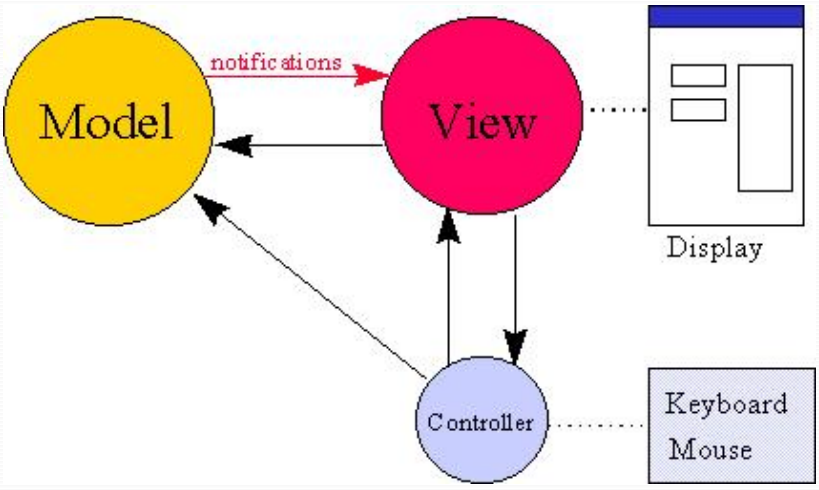
- Login en un servidor IMAP
- Listar todos los mensajes en la carpeta **INBOX**
- Ver los mensajes seleccionados
- Recuperar y ver los attachments de los mensajes seleccionados
- Componer y enviar mensajes

Arquitectura de la Aplicación

EWA es una aplicación web de tres capas. Reside en un servidor de aplicaciones web que interactúa con el servidor correo IMAP. Los clientes son páginas web generadas por páginas JSP desarrolladas en un entorno JSP/Servlet.

EWA utiliza una patrón de diseño derivado de "Model-View-Controller" (MVC). En el paradigma MVC, la entrada del usuario, el modelado del mundo exterior y la realimentación visual al usuario son explícitamente separados y manejados por tres tipos de objetos, cada uno especializado en su propia tarea.

La vista trabaja la salida gráfica y/o textual de la porción de pantalla mapeada que está asignada a esta aplicación. El controlador interpreta las entradas del usuario a través del ratón y del teclado, ordenando al modelo y/o la vista que se modifiquen de la forma apriada. Finalmente, el modelo maneja el comportamiento y los datos de la aplicación dominante, responde a las peticiones de información sobre su estado (normalmente desde la vista), y responde a las instrucciones de cambio de estado (normalmente desde el controlador).



En esta aplicación, el modelo consiste en el mensaje IMAP almacenado en el `MailUserBean`, la librería de etiquetas y el `AttachmentServlet`, que accede/maneja el almacenamiento del mensaje. La vista consta de componentes HTML y JSP que proporcionan el interface de usuario. El controlador es el `FilterServlet`, que valida el estado del login del usuario.

El Modelo

MailUserBean

Este es el JavaBean que almacena la información de email del usuario, como el **hostname**, el **username**, la **password**, la **session** y el **protocol**. Tiene métodos para obtener y almacenar estos parámetros. Para propósitos de esta demostración el protocolo ha sido codificado para ser **IMAP**. Este bean implementa los métodos `login` y `logout` del usuario. Usa los objetos `Session` y `Store` de mail para almacenar la combinación de **hostname**, **username** y **password**

```
import java.util.*;
import javax.mail.*;

/**
 * This JavaBean is used to store mail user information.
 */
public class MailUserBean {
    private Folder folder;
    private String hostname;
    private String username;
    private String password;
    private Session session;
    private Store store;
    private URLName url;
    private String protocol = "imap";
    private String mbox = "INBOX";

    public MailUserBean(){}

    /**
     * Returns the javax.mail.Folder object.
     */
    public Folder getFolder() {
        return folder;
    }

    /**
     * Returns the number of messages in the folder.
     */
    public int getMessageCount() throws MessagingException {
        return folder.getMessageCount();
    }

    /**
```

```
     * hostname getter method.
     */
    public String getHostname() {
        return hostname;
    }

    /**
     * hostname setter method.
     */
    public void setHostname(String hostname) {
        this.hostname = hostname;
    }

    /**
     * username getter method.
     */
    public String getUsername() {
        return username;
    }

    /**
     * username setter method.
     */
    public void setUsername(String username) {
        this.username = username;
    }

    /**
     * password getter method.
     */
    public String getPassword() {
        return password;
    }

    /**
     * password setter method.
     */
    public void setPassword(String password) {
        this.password = password;
    }

    /**
     * session getter method.
     */
    public Session getSession() {
        return session;
    }

    /**
     * session setter method.
     */
    public void setSession(Session s) {
        this.session = session;
    }

    /**
     * store getter method.
     */
    public Store getStore() {
        return store;
    }

    /**
     * store setter method.
     */
    public void setStore(Store store) {
        this.store = store;
    }

    /**
     * url getter method.
     */
    public URLName getUrl() {
        return url;
    }

    /**
     * Method for checking if the user is logged in.
     */
    public boolean isLoggedIn() {
        return store.isConnected();
    }
```

```
    /**
     * Method used to login to the mail host.
     */
    public void login() throws Exception {
        url = new URLName(protocol, getHostname(), -1, mbox,
                          getUsername(), getPassword());
        Properties props = System.getProperties();
        session = Session.getInstance(props, null);
        store = session.getStore(url);
        store.connect();
        folder = session.getFolder(url);

        folder.open(Folder.READ_WRITE);
    }

    /**
     * Method used to login to the mail host.
     */
    public void login(String hostname, String username, String password)
        throws Exception {

        this.hostname = hostname;
        this.username = username;
        this.password = password;

        login();
    }

    /**
     * Method used to logout from the mail host.
     */
    public void logout() throws MessagingException {
        folder.close(false);
        store.close();
        store = null;
        session = null;
    }
}
```

Librería de Etiquetas

La librería de etiquetas contiene las siguientes etiquetas personalizadas:

- `message`: usada para leer un mensaje, clases controladoras: `MessageTag.java`, `MessageTEI.java`
- `listmessages`: usada para iterar a través de la lista de mensajes, clases controladoras: `ListMessagesTag.java`, `ListMessagesTEI.java`
- `sendmail`: usada para enviar mensajes, clase controladora: `SendTag.java`

El fichero descriptor de etiquetas define el nombre de la etiqueta, su clase(o clases) controladoras, atributos, contenidos del cuerpo, etc. Por ejemplo, este fichero "**taglig.tld**" define todas las etiquetas de esta aplicación:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
        PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
        "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>javamail</shortname>
  <uri>http://java.sun.com/products/javamail/demo/webapp</uri>
  <tag>
    <name>listattachments</name>
    <tagclass>ListAttachmentsTag</tagclass>
    <teiclass>ListAttachmentsTEI</teiclass>
    <bodycontent>JSP</bodycontent>
    <info>
        A listattachments tag
    </info>
    <attribute>
      <name>id</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>messageinfo</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
```

```
<tag>
  <name>listmessages</name>
  <tagclass>ListMessagesTag</tagclass>
  <teiclass>ListMessagesTEI</teiclass>
  <bodycontent>JSP</bodycontent>
  <info>
      A listmessages tag
  </info>
  <attribute>
    <name>id</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>folder</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>session</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
<tag>
  <name>message</name>
  <tagclass>MessageTag</tagclass>
  <teiclass>MessageTEI</teiclass>
  <bodycontent>empty</bodycontent>
  <info>
      A message tag
  </info>
  <attribute>
    <name>id</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>folder</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>session</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>num</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
<tag>
  <name>sendmail</name>
  <tagclass>SendTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
      An sendmail tag
  </info>
  <attribute>
    <name>host</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>port</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>recipients</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>sender</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>subject</name>
    <required>false</required>
```

```
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

Cada una de las etiquetas tiene su propia clase controladora que implementa las acciones de la etiqueta. Por ejemplo, la etiqueta `listmessages` está implementada en `ListMessagesTag.java`:

```
import java.io.*;
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.mail.search.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

/**
 * Custom tag for listing messages. The scripting variable is only
 * within the body of the tag.
 */
public class ListMessagesTag extends BodyTagSupport {
    private String folder;
    private String session;
    private int msgNum = 0;
    private int messageCount = 0;
    private Message message;
    private Message[] messages;
    private MessageInfo messageinfo;

    /**
     * folder attribute getter method.
     */
    public String getFolder() {
        return folder;
    }

    /**
     * session attribute getter method.
     */
    public String getSession() {
        return session;
    }

    /**
     * folder setter method.
     */
    public void setFolder(String folder) {
        this.folder = folder;
    }

    /**
     * session attribute setter method.
     */
    public void setSession(String session) {
        this.session = session;
    }

    /**
     * Method for processing the start of the tag.
     */
    public int doStartTag() throws JspException {
        messageinfo = new MessageInfo();

        try {
            Folder folder = (Folder)pageContext.getAttribute(
                getFolder(), PageContext.SESSION_SCOPE);
            FlagTerm ft = new FlagTerm(new Flags(Flags.Flag.DELETED), false);
            messages = folder.search(ft);
            messageCount = messages.length;
        } catch (Exception ex) {
            throw new JspException(ex.getMessage());
        }

        if (messageCount > 0) {
            getMessage();
            return BodyTag.EVAL_BODY_TAG;
        } else
            return BodyTag.SKIP_BODY;
    }
```

```
    /**
     * Method for processing the body content of the tag.
     */
    public int doAfterBody() throws JspException {

        BodyContent body = getBodyContent();
        try {
            body.writeOut(getPreviousOut());
        } catch (IOException e) {
            throw new JspTagException("IterationTag: " + e.getMessage());
        }

        // clear up so the next time the body content is empty
        body.clearBody();

        if (msgNum < messageCount) {
            getMessage();
            return BodyTag.EVAL_BODY_TAG;
        } else {
            return BodyTag.SKIP_BODY;
        }
    }

    /**
     * Helper method for retrieving messages.
     */
    private void getMessage() throws JspException {
        message = messages[msgNum++];
        messageinfo.setMessage(message);
        pageContext.setAttribute(getId(), messageinfo);
    }
}
```

Las clases **TEI** son clases "Tag Extra Info" que proporcionan información sobre las variables de scripting que se crean/ modifican en tiempo de ejecución. Son necesarias para etiquetas que definen variables de scripting. Esta información se usa durante la fase de tradución de JSP. Aquí tenemos la clase TEI de la etiqueta `ListMessages`

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

/**
 * Extra information class to support the scripting variable created by the
 * ListMessagesTag class. The scope of the variable is limited to the body
 * of the tag.
 */
public class ListMessagesTEI extends TagExtraInfo {

    public ListMessagesTEI() {
        super();
    }

    public VariableInfo[] getVariableInfo(TagData data) {
        VariableInfo info = new VariableInfo(data.getId(),"MessageInfo",
            true, VariableInfo.NESTED);
        VariableInfo[] varInfo = { info };
        return varInfo;
    }
}
```

AttachmentServlet

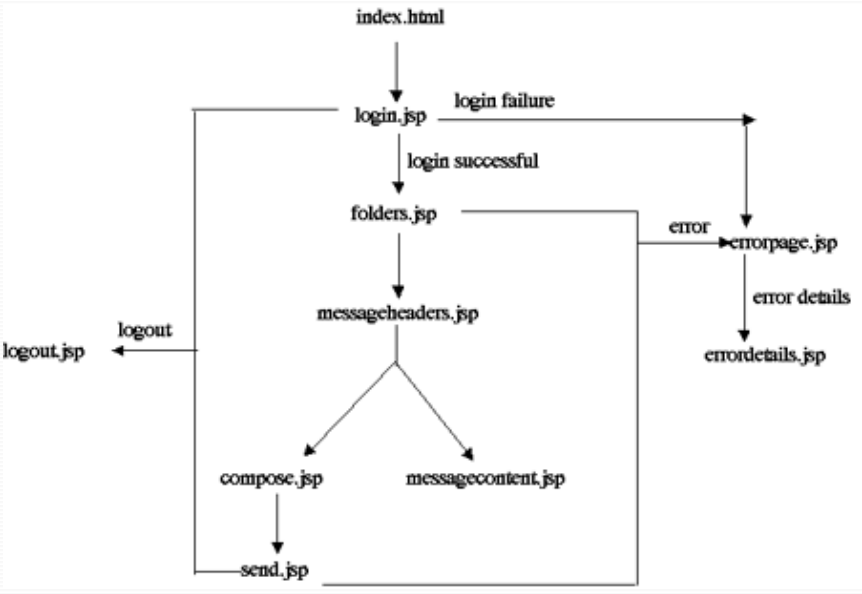
`AttachmentServlet` obtiene un stream de una parte dada de un mensaje multiparte y lo envía hacia el navegador con el tipo de contenido apropiado. Este servlet se usa para mostrar attachments y se relaciona con las capacidades de manejar tipos de contenidos que tiene el navegador.

```
import java.io.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * This servlet gets the input stream for a given msg part and
 * pushes it out to the browser with the correct content type.
 * Used to display attachments and relies on the browser's
 * content handling capabilities.
 */
```

```
public class AttachmentServlet extends HttpServlet {

    /**
     * This method handles the GET requests from the client.
     */
    public void doGet(HttpServletRequest request, HttpServletResponse  response)
        throws IOException, ServletException {

        HttpSession session = request.getSession();
        ServletOutputStream out = response.getOutputStream();
        int msgNum = Integer.parseInt(request.getParameter("message"));
        int partNum = Integer.parseInt(request.getParameter("part"));
        MailUserBean mailuser = (MailUserBean)session.getAttribute("mailuser");

        // check to be sure we're still logged in
        if (mailuser.isLoggedIn()) {
            try {
                Message msg = mailuser.getFolder().getMessage(msgNum);

                Multipart multipart = (Multipart)msg.getContent();
                Part part = multipart.getBodyPart(partNum);

                String sct = part.getContentType();
                if (sct == null) {
                    out.println("invalid part");
                    return;
                }
                ContentType ct = new ContentType(sct);

                response.setContentType(ct.getBaseType());
                InputStream is = part.getInputStream();
                int i;
                while ((i = is.read()) != -1)
                    out.write(i);
                out.flush();
                out.close();

            } catch (MessagingException ex) {
                throw new ServletException(ex.getMessage());
            }
        } else {
            getServletConfig().getServletContext().
                getRequestDispatcher("/index.html").
                forward(request, response);
        }
    }
}
```

La Vista

La vista consta de **JavaMail.html**, que es el punto de entrada inicial. Esta página requiere que el usuario introduzca un `username`, un `password` y un `hostname IMAP`. Después de haber validado la información de login, el usuario navega a traves de una serie de páginas web.



El Controlador

El controlador es el `FilterServlet`. Este servlet se usa para determinar si el usuario había entrado anteriormente de enviar la petición a la URL seleccionada. El método `doPost` maneja el envío **POST** de dos formas: `login.jsp` y `compose.jsp`. El método `doGet` maneja las peticiones **GET** desde el cliente.

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

/**
 * This servlet is used to determine whether the user is logged in before
 * forwarding the request to the selected URL.
 */
public class FilterServlet extends HttpServlet {

    /**
     * This method handles the "POST" submission from two forms: the
     * login form and the message compose form.
     */
    public void doPost(HttpServletRequest request,
                       HttpServletResponse  response)
                       throws IOException, ServletException {

        String servletPath = request.getServletPath();
        servletPath = servletPath.concat(".jsp");

        getServletConfig().getServletContext().
            getRequestDispatcher("/" + servletPath).forward(request, response);
    }

    /**
     * This method handles the GET requests from the client.
     */
    public void doGet(HttpServletRequest request,
                      HttpServletResponse  response)
                      throws IOException, ServletException {

        // check to be sure we're still logged in
        // before forwarding the request.
        HttpSession session = request.getSession();
        MailUserBean mailuser = (MailUserBean)session.getAttribute("mailuser");
        String servletPath = request.getServletPath();
        servletPath = servletPath.concat(".jsp");

        if (mailuser.isLoggedIn())
            getServletConfig().getServletContext().
                getRequestDispatcher("/" + servletPath).
                forward(request, response);
        else
            getServletConfig().getServletContext().
                getRequestDispatcher("/index.html").
                forward(request, response);
    }
}
```

Desarrollo de la aplicación

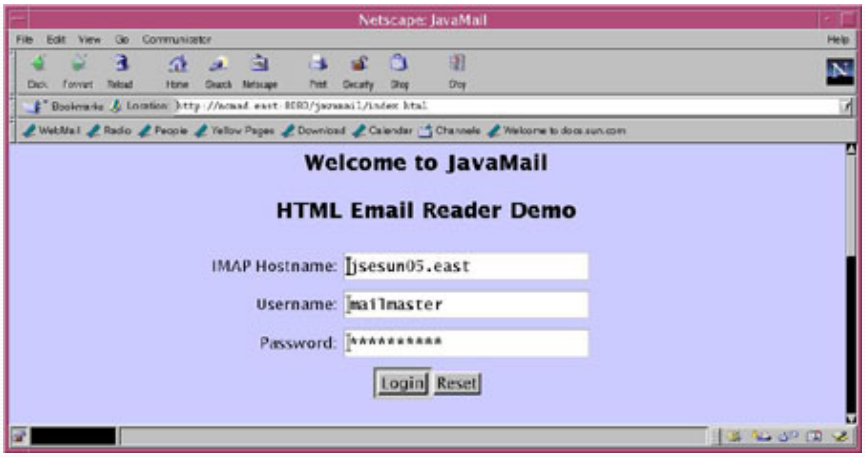
Esta aplicación fue desarrollada usando:

- JavaMail 1.2
- Servlet 2.2
- JavaServer Pages 1.1
- JavaBeans Activation Framework (JAF) 1.0.1

Para ejecutar esta aplicación, necesitamos lo siguiente:

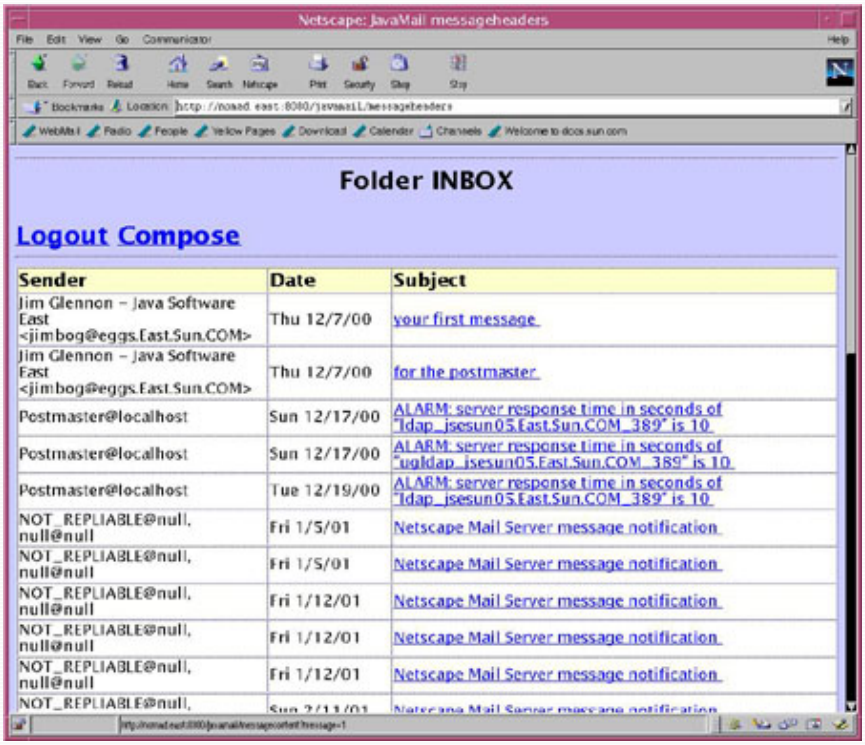
1. Una implementación JSP/Servlet que pueda ser ejecutada como una aplicación solitaria o como una extensión de un servidor Wev. Para está aplicación, usamos Tomcat 3.2.1.
2. Un servidor de mail IMAP.

Aquí tenemos unas cuantas imágenes de la ejecución del ejemplo, con su correspondiente código fuente. El punto de entrada es index.html.



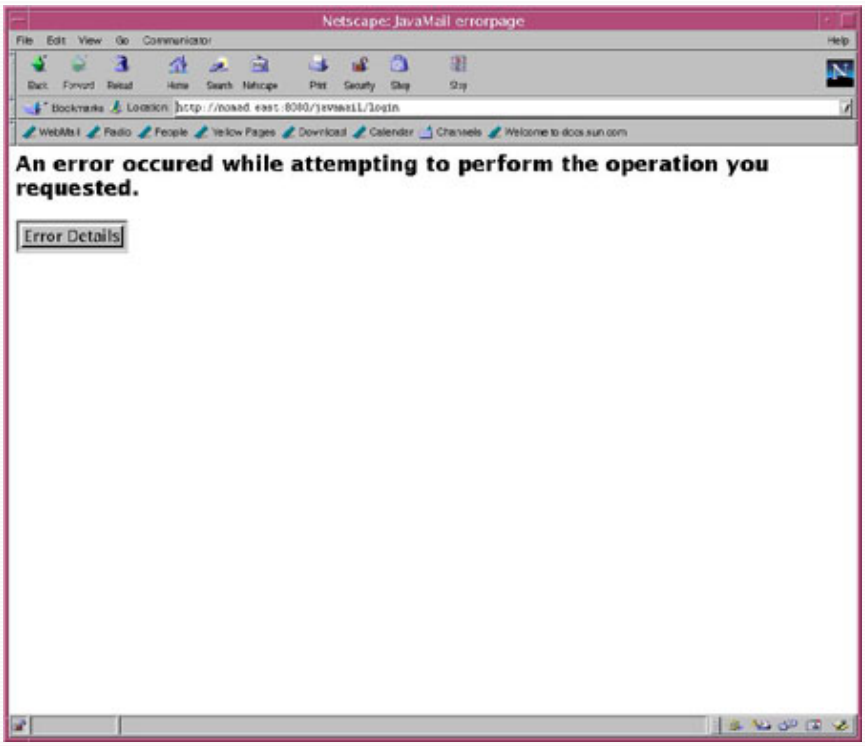

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>

<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html;CHARSET=iso-8859-1">
<TITLE>JavaMail</TITLE>
</HEAD>

<BODY BGCOLOR="#CCCCFF">

<FORM ACTION="login" METHOD=POST ENCTYPE="application/x-www-form-urlencoded">
<P ALIGN="CENTER"><B>Welcome to JavaMail</B></P>

<P ALIGN="CENTER"><B>HTML Email Reader Demo</B></P>
<CENTER>
<P>
<TABLE BORDER="0" WIDTH="100%">
<TR>
<TD WIDTH="40%">
<P ALIGN="RIGHT">IMAP Hostname:
</TD>
<TD WIDTH="60%"><INPUT TYPE="TEXT" NAME="hostname" SIZE="25"></TD>
</TR>
<TR>
<TD WIDTH="40%">
<P ALIGN="RIGHT">Username:
</TD>
<TD WIDTH="60%"><INPUT TYPE="TEXT" NAME="username" SIZE="25"></TD>
</TR>
<TR>
<TD WIDTH="40%">
<P ALIGN="RIGHT">Password:
</TD>
<TD WIDTH="60%"><INPUT TYPE="PASSWORD" NAME="password" SIZE="25"></TD>
</TR>
</TABLE>
<INPUT TYPE="SUBMIT" VALUE="Login">
<INPUT TYPE="RESET" NAME="Reset" VALUE="Reset"></P>
</CENTER>

<P><B><I>Features:</I></B></P>

<UL>
<LI>HTML access to your IMAP mailbox
<LI>Proxy-able anywhere HTTP can be proxied
<LI>Easy to use
<LI>Uses web browser's content handling capabilities
</UL>

<P><B><I>Limitations:</I></B></P>

<UL>
<LI>Only INBOX support (no user folders)
<LI>Can't delete, copy, move, print, save, forward, reply to, search in
messages but it could be done
<LI>Doesn't check for new messages (have to log out and log back it)
</UL>
<P>
<HR ALIGN="CENTER">
</P>
</FORM>
</BODY>
</HTML>
```

Un login con éxito nos lleva al inbox. Si el login falla, se muestra la página de error. Entonces el usuario tiene una opción para ver los detalles del error.

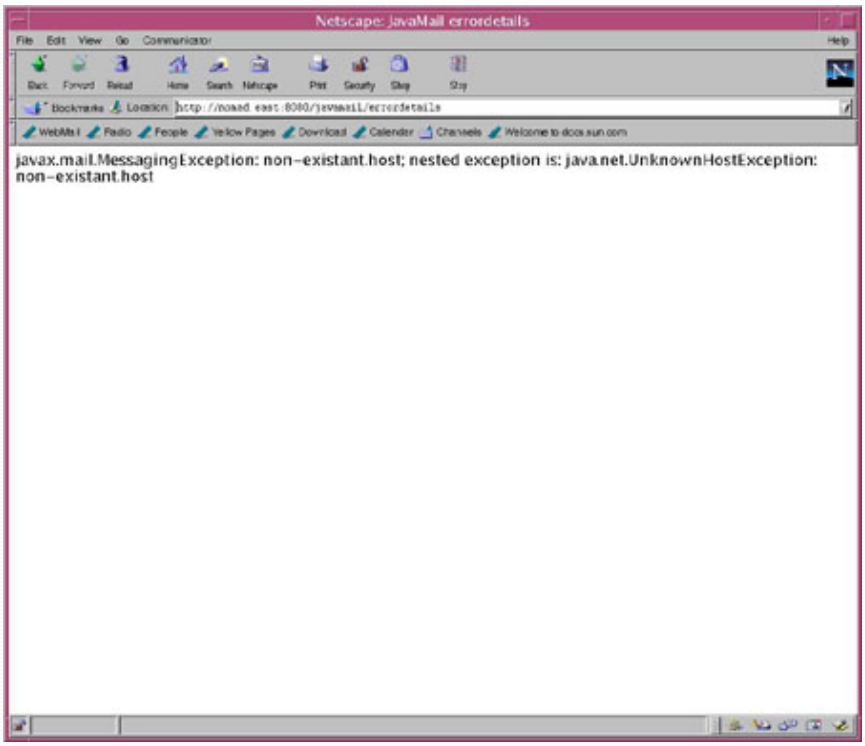



```
<%@ page language="java" import="MailUserBean" %>
<%@ page errorPage="errorpage.jsp" %>
<%@ taglib uri="http://java.sun.com/products/javamail/demo/webapp"
    prefix="javamail" %>

<html>
<head>
        <title>JavaMail messageheaders</title>
</head>

<body bgcolor="#ccccff"><hr>

<center><font face="Arial,Helvetica" font size="+3">
<b>Folder INBOX</b></font></center><p>

<font face="Arial,Helvetica" font size="+3">
<b><a href="logout">Logout</a>
<a href="compose" target="compose">Compose</a>
</b></font>
<hr>

<table cellpadding=1 cellspacing=1 width="100%" border=1>
<tr>
<td width="25%" bgcolor="ffffcc">
<font face="Arial,Helvetica" font size="+1">
<b>Sender</b></font></td>
<td width="15%" bgcolor="ffffcc">
<font face="Arial,Helvetica" font size="+1">
<b>Date</b></font></td>
<td bgcolor="ffffcc">
<font face="Arial,Helvetica" font size="+1">
<b>Subject</b></font></td>
</tr>
<javamail:listmessages
 id="parteV_msginfo"
 folder="folder">
<%-- from --%>
<tr valign=middle>
<td width="25%" bgcolor="ffffff">
<font face="Arial,Helvetica">
<% if (msginfo.hasFrom()) { %>
<%= msginfo.getFrom() %>
</font>
<% } else { %>
<font face="Arial,Helvetica,sans-serif">
Unknown
```

```
<% } %>
</font></td>
<%-- date --%>
<td nowrap width="15%" bgcolor="ffffff">
<font face="Arial,Helvetica">
<%= msginfo.getDate() %>
</font></td>
<%-- subject & link --%>
<td bgcolor="ffffff">
<font face="Arial,Helvetica">
<a href="messagecontent?message=<%= msginfo.getNum() %>">
<% if (msginfo.hasSubject()) { %>
<%=    msginfo.getSubject() %>
<% } else { %>
<i>No Subject</i>
<% } %>
</a>
</font></td>
</tr>
</javamail:listmessages>
</table>
</body>
</html>
```

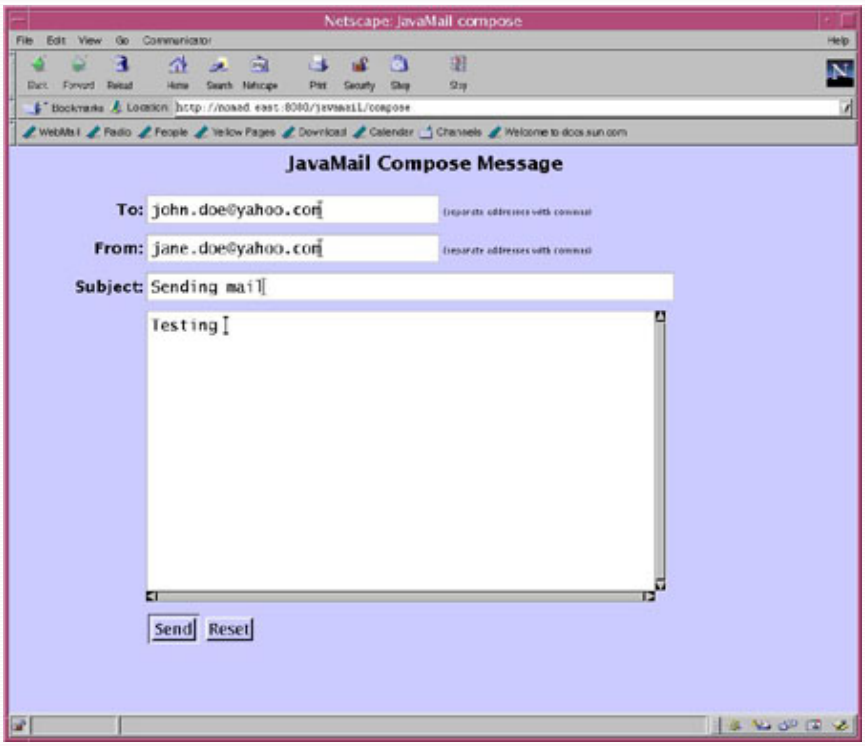



```
<%@ page isErrorPage="true" %>
<html>
<head>
<title>JavaMail errorpage</title>
</head>
<body bgcolor="white">
<form ACTION="errordetails" METHOD=POST>
<% session.putValue("details", exception.toString()); %>
<h2>An error occured while attempting to perform the operation you requested.
</h2>
<input type="submit" name="Error Details" value="Error Details">
</body>
</html>
```



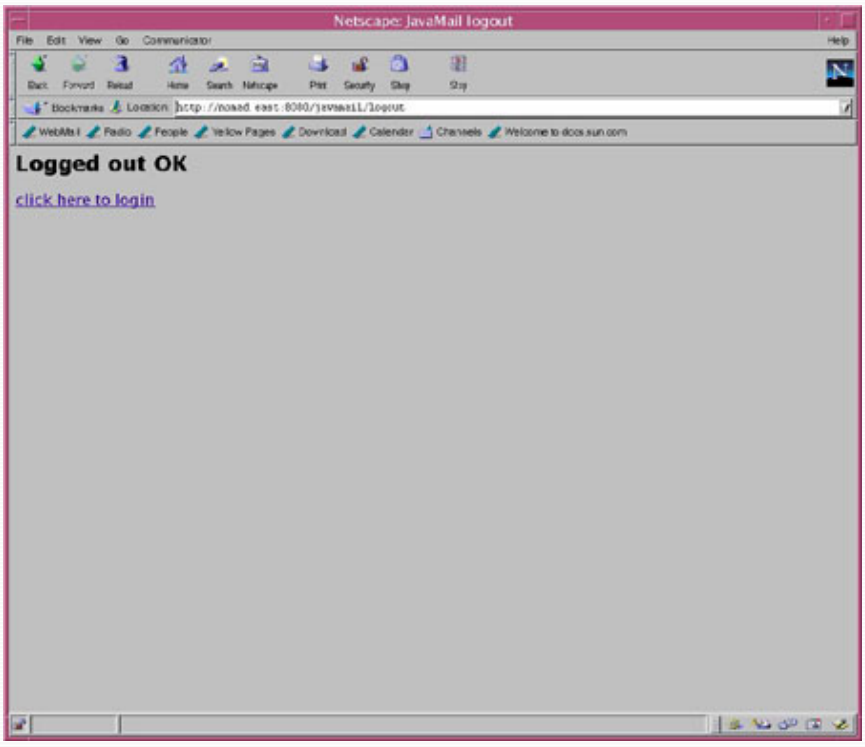

```
<%@ page isErrorPage="true" %>
<html>
<head>
<title>JavaMail errordetails</title>
</head>
<body bgcolor="white">
<%= session.getValue("details") %>
</body>
</html>
```

Desde dentro del inbox, el usuario tiene la opción de componer un mensaje o salir.




```
<%@ page language="java" %>
<%@ page errorPage="errorpage.jsp" %>

<html>
<head>
        <title>JavaMail compose</title>
```

```
</head>

<body bgcolor="#ccccff">
<form ACTION="send" METHOD=POST>
<input type="hidden" name="send" value="send">
<p align="center">
<b><font size="4" face="Verdana, Arial, Helvetica">
JavaMail Compose Message</font></b>
<p>
<table border="0" width="100%">
<tr>
<td width="16%" height="22">
<p align="right">
<b><font face="Verdana, Arial, Helvetica">To:</font></b></td>
<td width="84%" height="22">
<% if (request.getParameter("to") != null) { %>
<input type="text" name="to" value="<%= request.getParameter("to") %>" size="30">
<% } else { %>
<input type="text" name="to" size="30">
<% } %>
<font size="1" face="Verdana, Arial, Helvetica">
 (separate addresses with commas)</font></td></tr>
<tr>
<td width="16%"><p align="right">
<b><font face="Verdana, Arial, Helvetica">From:</font></b></td>
<td width="84%">
<input type="text" name="from" size="30">
<font size="1" face="Verdana, Arial, Helvetica">
 (separate addresses with commas)</font></td></tr>
<tr>
<td width="16%"><p align="right">
<b><font face="Verdana, Arial, Helvetica">Subject:</font></b></td>
<td width="84%">
<input type="text" name="subject" size="55"></td></tr>
<tr>
<td width="16%"> </td>
<td width="84%"><textarea name="text" rows="15" cols="53"></textarea></td></tr>
<tr>
<td width="16%" height="32"> </td>
<td width="84%" height="32">
<input type="submit" name="Send" value="Send">
<input type="reset" name="Reset" value="Reset"></td></tr>
</table>
</form>
</body>
</html>
```




```
<%@ page language="java" import="MailUserBean" %>
<%@ page errorPage="errorpage.jsp" %>
<jsp:useBean id="parteV_mailuser" scope="session" class="MailUserBean" />
```

```
<html>
<head>
        <title>JavaMail logout</title>
</head>

<% mailuser.logout(); %>

<body>
<h2>Logged out OK</h2><a href=index.html>click here to login</a>
</body>
</html>
```

EWA está compuesto por un documento HTML y varios documentos web (servlets y JSP y etiquetas persnalizadas). a parte de estos, tiene dos directorios:**META-INF** y **WEB-INF**.

El directorio META-INF contiene:

- El fichero de manifiesto para la aplicación.

El directorio WEB-INF contiene:

- El fichero **web.xml** que contiene la configuración y la información de despliegue de la aplicación.
- El directorio **classes** que contiene las clases servlet y de utilidad usadas por la aplicación web.
- El directorio **lib** que contiene el archivo (**jtl.jar**) para la librería de etiquetas personalizadas.

Aquí puedes decargar los fuentes de la aplicación EWA

Construir y empaquetar la aplicación JavaMail

Una vez descomprimido el fichero zip de descarga, los ficheros fuente se encontrarán en los siguientes directorios:

- src/classes
- src/docroot
- src/taglib

Dentro del fichero zip también podrás encontrar dos ficheros de scripts (build.sh y build.bat) para construir y empaquetar la aplicación. Antes de ejecutar estos scripts debes asegurarte de que los siguientes ficheros están en tu **CLASSPATH**:

- mail.jar - The JavaMail jar file
- activation.jar - the JAF jar file
- servlet.jar - the servlet/JSP jar file

Si lo haces a mano, debes realizar los siguientes pasos para construir y empaquetar la aplicacion:

1. Crear un directorio llamado "src/docroot/WEB-INF/classes".
2. Crear un directorio llamado "src/docroot/WEB-INF/lib".
3. Compilar los ficheros del directorio "src/classes" y añadirlos al directorio "src/docroot/WEB-INF/classes".
4. Compilar los ficheros del directorio "src/tablib".
5. Crear un archivo jar (jtl.jar) con las clases de la librería **taglib** y añadirlo a "src/docroot/WEB-INF/lib".
6. Crear un archivo web (.war) con los contenidos de "src/docroot" (y todos sus subdirectorios).

Una nota sobre el envío de mail

Para poder envíar correo usando esta aplicación , es necesario especificar un host SMTP. Esto puede hacerse de un par de formas:

1. Usar la variable de entorno **TOMCAT_OPTS**.
   Añadiendo lo siguiente al fichero de configuración de Tomcat:

   ```
   -Dmail.smtp.host=yourSMTPmailservername
   ```

   Rearrancar el servidor.
2. Modificar el fichero **send.jsp** y actualizar el fichero **javamail.war**:
   Añadir el siguiente parámetro a la etiqueta `<javamail:sendmail>`:

   ```
   host="yourSMTPmailservername"
   ```

   Reempaquetar el fichero **javamail.war**para incluir el fichero **send.jsp**modificado.

Conclusión

Esta aplicación demuestra una forma de desarrollar una aplicación usando el API JavaMail y JSP. Se ha hecho un esfuerzo para implementar la mayoría de las acciones en la librería de etiquetas y así minimizar el código en-línea. Sin embargo, el manejo de attachment se ha realizado en un servlet en lugar de en una etiqueta. Esto fue más una materia de conveniencia que de diseño. La funcionalidad de `AttachmentServlet` se podría haber implementado perfectamente en un etiqueta.

La siguiente versión del API JavaMail contendrá una aplicación demo similar a la explicada en este artículo. Contendrá más características y ciertas modificaciones a esta versión. Por favor visita JavaMail API and Internet Mail Resources para ver las actualizaciones.


Indice de contenidos