AJAX en J2EE

2ª EDICIÓN ACTUALIZADA

Antonio J. Martín Sierra

Desde **www.ra-ma.es** podrá descargarse material adicional con los ejercicios del libro.

AJAX en JAVA EE

2.ª Edición

Descarga de Material Adicional

Este E-book tiene disponible un material adicional que complementa el contenido del mismo.

Este material se encuentra disponible en nuestra página Web www.ra-ma.com.

Para descargarlo debe dirigirse a la ficha del libro de papel que se corresponde con el libro electrónico que Ud. ha adquirido. Para localizar la ficha del libro de papel puede utilizar el buscador de la Web.

Una vez en la ficha del libro encontrará un enlace con un texto similar a este:

"Descarga del material adicional del libro"

Pulsando sobre este enlace, el fichero comenzará a descargarse.

Una vez concluida la descarga dispondrá de un archivo comprimido. Debe utilizar un software descompresor adecuado para completar la operación. En el proceso de descompresión se le solicitará una contraseña, dicha contraseña coincide con los 13 dígitos del ISBN del libro de papel (incluidos los guiones).

Encontrará este dato en la misma ficha del libro donde descargó el material adicional.

Si tiene cualquier pregunta no dude en ponerse en contacto con nosotros en la siguiente dirección de correo: ebooks@ra-ma.com

AJAX en JAVA EE

2.ª Edición

Antonio J. Martín Sierra

La ley prohíbe
Copiar o Imprimir este libro

AJAX EN J2EE. 2ª EDICIÓN
© Antonio J. Martín Sierra

© De la Edición Original en papel publicada por Editorial RA-MA
ISBN de Edición en Papel: 978-84-9964-084-6
Todos los derechos reservados © RA-MA, S.A. Editorial y Publicaciones, Madrid, España.

MARCAS COMERCIALES. Las designaciones utilizadas por las empresas para distinguir sus productos (hardware, software, sistemas operativos, etc.) suelen ser marcas registradas. RA-MA ha intentado a lo largo de este libro distinguir las marcas comerciales de los términos descriptivos, siguiendo el estilo que utiliza el fabricante, sin intención de infringir la marca y solo en beneficio del propietario de la misma. Los datos de los ejemplos y pantallas son ficticios a no ser que se especifique lo contrario.

RA-MA es una marca comercial registrada.

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa. Sin embargo, RA-MA Editorial no asume ninguna responsabilidad derivada de su uso ni tampoco de cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene por objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa o de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.

Reservados todos los derechos de publicación en cualquier idioma.

Según lo dispuesto en el Código Penal vigente ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro sin autorización previa y por escrito de RA-MA; su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes, intencionadamente, reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica.

Editado por:
RA-MA, S.A. Editorial y Publicaciones
Calle Jarama, 33, Polígono Industrial IGARSA
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: editorial@ra-ma.com
Internet: www.ra-ma.es y www.ra-ma.com

Maquetación: Gustavo San Román Borrueco
Diseño Portada: Antonio García Tomé

ISBN: 978-84-9964-411-0

E-Book desarrollado en España en septiembre de 2014.

A María José, mi esposa, y a mis hijos Alejandro y Raúl.

ÍNDICE

PRÓLOGO **13**

CAPÍTULO 1. INTRODUCCIÓN A AJAX **17**

1.1 CONTEXTO DE UTILIZACIÓN DE AJAX 17

1.2 ¿QUÉ ES AJAX? 18

1.3 LAS TECNOLOGÍAS AJAX 20

1.4 PRIMERA APLICACIÓN AJAX 22

1.4.1 Descripción de la aplicación 23

1.4.2 Código de servidor 23

1.4.3 La vista cliente 24

1.4.4 Código de script de cliente 26

1.4.4.1 Creación del objeto XMLHttpRequest 26

1.4.4.2 Envío de la petición HTTP 27

1.4.4.3 Procesamiento de la respuesta 28

1.5 APLICACIONES AJAX MULTINAVEGADOR 29

1.5.1 Compatibilidad de código en todos los navegadores 30

1.6 MANIPULAR DATOS EN FORMATO XML 32

CAPÍTULO 2. PROCESO DE EJECUCIÓN DE UNA APLICACIÓN AJAX **37**

2.1 EVENTOS EN UNA PÁGINA WEB Y MANEJADORES 37

2.2 FASES EN LA EJECUCIÓN DE UNA APLICACIÓN AJAX 38

2.3 EL OBJETO XMLHTTPREQUEST 40

2.3.1 Los orígenes del objeto XMLHttpRequest 40

2.3.2 MIEMBROS DE LA INTERFAZ 41
2.3.2.1 Preparación de la petición 41
2.3.2.2 Definición de encabezados de la petición 43
2.3.2.3 Definición de la función de retrollamada 43
2.3.2.4 Estado de la petición 44
2.3.2.5 Envío de la petición 45
2.3.2.6 Encabezado de la respuesta 50
2.3.2.7 Acceso al contenido de la respuesta 52
2.3.2.8 Estado de la respuesta 53
2.4 EL MODELO DE OBJETO DOCUMENTO (DOM) 59
2.4.1 NIVELES DOM 59
2.4.2 INTERFACES DOM 60
2.4.3 EL ÁRBOL DE OBJETOS DE UN DOCUMENTO 61
2.4.4 DOM EN LA PRÁCTICA 62
2.4.4.1 Obtención del objeto Document 62
2.4.4.2 Referencia a los elementos del documento 63
2.4.4.3 Acceso al contenido de un elemento 63
2.4.4.4 Elementos hijos de un elemento 64
2.4.4.5 Tipo, nombre y valor de un nodo 64
2.4.4.6 Desplazamiento por el árbol de objetos 66
2.4.4.7 Acceso a los atributos de un elemento 67
2.4.4.8 Modificación de la estructura de un documento 68
CAPÍTULO 3. UTILIDADES AJAX 75
3.1 ENCAPSULACIÓN DEL OBJETO XMLHTTPREQUEST 75
3.1.1 LA INTERFAZ DE LA CLASE OBJETOAJAX 76
3.1.2 IMPLEMENTACIÓN DE LA CLASE OBJETOAJAX 77
3.1.2.1 Constructor 77
3.1.2.2 El envío de la petición 78
3.1.2.3 Acceso a la respuesta 80
3.1.2.4 Estado de la respuesta 81
3.1.3 UTILIZACIÓN DE LA CLASE OBJETOAJAX 81
3.2 JSON 83
3.2.1 CARACTERÍSTICAS DE JSON 83
3.2.2 ESTRUCTURA DE DATOS JSON 83
3.2.2.1 Objetos JSON 83
3.2.2.2 Arrays 85

3.2.3 INTERPRETACIÓN DE JSON EN CLIENTE 85
3.2.4 EJEMPLO DE UTILIZACIÓN 86
3.2.5 LA LIBRERÍA JSON 88
3.3 DIRECT WEB REMOTING 96
3.3.1 COMPONENTES DWR 96
3.3.2 EL KIT DE DESARROLLO DWR 97
3.3.3 FUNCIONAMIENTO DE UNA APLICACIÓN DWR 97
3.3.4 DESARROLLO DE UNA APLICACIÓN DWR 99
3.3.4.1 Configuración del entorno 99
3.3.4.2 Implementación de la clase del servidor 100
3.3.4.3 Creación del archivo dwr.xml 101
3.3.4.4 Creación de la página cliente 102
3.3.4.5 Utilidades DWR de cliente 104
3.3.5 OPCIONES DE CONFIGURACIÓN DE DWR.XML 106
3.3.5.1 Instancias de objetos sin constructores públicos 106
3.3.5.2 Inclusión y exclusión de métodos 107
3.3.5.3 Convertidores 108
3.4 HERRAMIENTAS AJAX PARA CLIENTE 113
3.4.1 PROTOTYPE 114
3.4.1.1 El núcleo de objetos prototype 114
3.4.1.2 Utilidades Prototype 118
3.4.2 LA LIBRERÍA SCRIPT.ACULO.US 129
3.4.3 EL FRAMEWORK JQUERY 137
3.4.3.1 Utilización de jQuery 137
3.4.3.2 Componentes jQuery 137

CAPÍTULO 4. AJAX EN APLICACIONES JAVA EE 153
4.1 ARQUITECTURA DE TRES CAPAS 154
4.1.1 CAPA CLIENTE 154
4.1.2 CAPA INTERMEDIA 155
4.1.3 CAPA DE DATOS 155
4.2 ARQUITECTURA MODELO VISTA CONTROLADOR 156
4.2.1 EL CONTROLADOR 157
4.2.2 LA VISTA 157
4.2.3 EL MODELO 158
4.2.4 APLICACIÓN MVC BÁSICA 158
4.2.4.1 Solución sin AJAX 160

4.2.4.2 Solución con AJAX 170
4.2.5 IMPLEMENTACIÓN DE UN CARRITO DE COMPRA 175
4.2.5.1 El modelo 176
4.2.5.2 El controlador 181
4.2.5.3 La vista 183
4.3 JAVASERVER FACES 187
4.3.1 COMPONENTES DE LA TECNOLOGÍA JSF 188
4.3.2 ARQUITECTURA DE UNA APLICACIÓN JSF 189
4.3.2.1 El controlador 189
4.3.2.2 LA VISTA 190
4.3.2.3 El modelo 191
4.3.3 PROCESO DE CONSTRUCCIÓN DE UNA APLICACIÓN JSF 192
4.3.3.1 Configuración del entorno 192
4.3.3.2 Lógica de negocio de la aplicación 193
4.3.3.3 Bean gestionados 195
4.3.3.4 Componentes de la interfaz de usuario 197
4.3.3.5 Navegación entre páginas 202
4.3.4 AJAX EN APLICACIONES JSF 203
4.3.4.1 Funcionalidad AJAX en JSF 2 205

APÉNDICE A. EL ESTÁNDAR XHTML 235
CARACTERÍSTICAS BÁSICAS DE XHTML 236
ETIQUETAS HTML 236
Tipos de etiquetas 236
ATRIBUTOS 237
COMENTARIOS 237
ESTRUCTURA DE UN DOCUMENTO XHTML 237
PRINCIPALES ETIQUETAS DE XHTML 239
ORGANIZACIÓN DE TEXTO 239
FORMATO DEL TEXTO 240
ENCABEZADOS 240
SEPARADORES 241
HIPERTEXTO 241
LISTAS 242
TABLAS 244
IMÁGENES 245
FRAMES 246

FORMULARIOS HTML 247
El formulario HTML 247
Los controles HTML 248
Control Text 248
Control TextArea 248
Control PassWord 249
Control Submit 249
Control Button 249
Control CheckBox 249
Control Radio 250
Control Select 250
HOJAS DE ESTILO CSS 251
Tipos de hojas de estilo 252
Hojas de estilo externas 252
Hojas de estilo internas 253
Hojas de estilo en línea 254

APÉNDICE B. XML 255
FUNDAMENTOS SOBRE XML 255
¿Qué es XML? 255
Documentos XML 256
¿Por qué XML? 257
XML vs. HTML 257
CARACTERÍSTICAS DE XML 258
Comprensible 258
Basado en texto 259
Independiente 260
APLICACIONES DEL XML 260
Intercambio de datos entre aplicaciones (B2B) 260
Almacenamiento intermedio en aplicaciones Web 261
Presentación en la Web 262
Utilización como base de datos 262
TECNOLOGÍAS BASADAS EN XML 263
Construcción de documentos XML 264
Estructura de un documento XML 264
Reglas sintácticas XML 265
Documentos bien formados 266

APÉNDICE C. EL LENGUAJE JAVASCRIPT 267
JAVASCRIPT EN DOCUMENTOS XHTML 268
SINTAXIS DEL LENGUAJE 268
SINTAXIS BÁSICA 268
TIPOS DE DATOS Y VARIABLES 269
OPERADORES 270
INSTRUCCIONES DE CONTROL 271
if 271
switch 272
for 273
while 274
Las sentencias break y continue 275
FUNCIONES 275
Funciones del lenguaje 275
Cuadros de diálogo 276
Definición de funciones 277
EVENTOS 279
MANEJADORES DE EVENTO 279
TIPOS DE EVENTO 280
OBJETOS 281
TIPOS DE OBJETOS 281
OBJETOS DEL LENGUAJE 282
Objeto String 282
Objeto Array 283
Objeto Math 285
Objeto Date 288
OBJETOS XHTML 289
Referencia a los objetos etiqueta 289
Principales propiedades 289
OBJETOS CONTROLES XHTML 291
Propiedades 291
OBJETOS DEL NAVEGADOR 295
Objeto window 296
Objeto document 300
Objeto Navigator 301

PRÓLOGO

Cuando Jesse James Garret, en un artículo publicado en febrero de 2005, decidió dar el nombre de AJAX a la combinación de una serie de tecnologías Web existentes, no imaginó la repercusión que ese término iba a tener en la comunidad de desarrolladores.

Es incluso paradójico que una técnica de programación, basada en el uso conjunto de tecnologías existentes, haya tenido más éxito y difusión que algunas de las propias tecnologías en las que se apoya.

Pero la razón de este éxito es bien sencilla: con AJAX es posible trasladar la experiencia de usuario con las aplicaciones de escritorio a la Web, reduciendo enormemente los tiempos de espera y permitiendo que las páginas actualicen su contenido de manera casi inmediata como respuesta a las acciones de usuario.

Esta nueva filosofía de trabajo permite orientar los desarrollos Web hacia lo que se conoce como aplicaciones de una página. Hasta la llegada de AJAX, las aplicaciones Web consistían en una serie de recursos de servidor (páginas JSP, ASP, PHP, etc.), cuya función consistía en generar y enviar una nueva página al cliente como respuesta a una solicitud realizada por éste a la aplicación. Con AJAX se pretende cambiar ligeramente esa forma de trabajo, haciendo que las peticiones al servidor devuelvan únicamente datos al cliente para que éste los muestre sobre la vista actual, en vez de generar nuevas páginas completas.

OBJETIVO DEL LIBRO

El objetivo de este libro es acercar al lector esta técnica y todas las tecnologías en las que se basa, presentándole todos los elementos necesarios para que pueda desarrollar auténticas aplicaciones Web interactivas.

Dado que AJAX es una técnica de programación orientada a la capa cliente, puede ser utilizada en cualquier desarrollo Web, independientemente de la tecnología utilizada para la implementación de la capa intermedia. En nuestro caso, tal y como se refleja en el título del libro, centraremos nuestro estudio en el uso de AJAX dentro de aplicaciones Web JAVA EE, si bien los contenidos analizados en los tres primeros capítulos del libro son igualmente aplicables en cualquier otra arquitectura de desarrollo.

A QUIÉN VA DIRIGIDO

Este libro está dirigido fundamentalmente a programadores Web en entorno Java que deseen mejorar la funcionalidad de sus aplicaciones, añadiendo a sus desarrollos la potencia y capacidades que ofrece AJAX.

Puesto que el código de servidor de los ejemplos y prácticas presentados en el libro están escritos en Java, será necesario el conocimiento tanto de este lenguaje como de la tecnología JAVA EE para su comprensión y puesta en funcionamiento de los mismos. No obstante, un lector con conocimientos de otro lenguaje de programación podrá adaptar a éste la mayoría de los ejemplos y prácticas, manteniendo el código AJAX de cliente.

Así pues, cualquier persona con conocimientos de programación será capaz, utilizando este libro, de comprender la mecánica de funcionamiento AJAX y de adaptarla a su entorno particular.

ESTRUCTURA DEL LIBRO

El libro está organizado en cuatro capítulos y tres apéndices.

En el capítulo 1 se exponen los fundamentos básicos de AJAX y sus aplicaciones, analizando a través de un ejemplo el funcionamiento a nivel global de este tipo de aplicaciones.

Es durante el capítulo 2 donde se estudia con detenimiento el proceso de desarrollo y ejecución de la aplicaciones AJAX, así como las tecnologías en que se basa, abordando, además, la problemática de la compatibilidad del código para distintos tipos y versiones de navegadores. Se analizarán con detalle en este capítulo todas las propiedades y métodos que expone el objeto XMLHttpRequest.

Una vez completado el análisis del desarrollo con AJAX, el capítulo 3 nos presenta las distintas herramientas y librerías de libre distribución creadas por la comunidad de desarrolladores AJAX, orientadas a facilitar la construcción de este tipo de aplicaciones y mejorar su rendimiento.

Por último, el capítulo 4 aborda la integración de AJAX dentro de las aplicaciones JAVA EE, enfocando el análisis dentro del contexto de las aplicaciones basadas en arquitectura Modelo Vista Controlador. En esta línea, el capítulo finaliza presentando el marco de trabajo Java Server Faces y la utilización dentro del mismo de componentes gráficos con capacidades AJAX.

En cuanto a los tres apéndices finales, pretenden ser una guía de referencia para el usuario sobre las tecnologías básicas en la que se apoya AJAX, en este orden: XHTML-CSS, XML y JavaScript. Puede encontrar un desarrollo más amplio de estos temas en otros títulos publicados por esta misma editorial.

MATERIAL ADICIONAL

Los ejercicios prácticos desarrollados en los distintos capítulos del libro pueden ser descargados desde la Web de Ra-Ma. Debe dirigirse a la ficha correspondiente a esta obra, dentro de la ficha encontrará el enlace para poder realizar la descarga. Dicha descarga consiste en un fichero ZIP con una contraseña de este tipo: XXX-XX-XXXX-XXX-X la cual se corresponde con el ISBN de este libro.

Podrá localizar el número de ISBN en la página IV (página de créditos). Para su correcta descompresión deberá introducir los dígitos y los guiones.

Cuando descomprima el fichero obtendrá los archivos que complementan al libro para que pueda continuar con su aprendizaje.

Estas aplicaciones han sido creadas con el entorno de desarrollo Java NetBeans, de modo que si el lector cuenta con dicho entorno instalado en su máquina podrá abrir directamente los proyectos y probar su funcionamiento. No

obstante, cada proyecto incluye una carpeta con los códigos fuente a fin de que puedan ser utilizados en otro entorno.

CONVENCIONES UTILIZADAS

Se han utilizado las siguientes convenciones a lo largo del libro:

- Uso de **negrita** para resaltar ciertas definiciones o puntos importantes a tener en cuenta.
- Utilización de *cursiva* para nombres y propiedades de objetos, así como para presentar el formato de utilización de algún elemento de código.
- Empleo de estilo `courier` para listados, tanto de código como de etiquetado HTML. Para destacar los comentarios dentro del código se utilizará el tipo *`courier en cursiva`*, mientras que para resaltar las instrucciones importantes se empleará **`courier en negrita`**.

DIRECCIÓN DE CONTACTO

Espero que este libro resulte de utilidad al lector y le ayude a integrar en sus desarrollos todas las capacidades que ofrece esta tecnología.

Si desea realizar alguna consulta u observación, puede contactar conmigo a través de la siguiente dirección de correo electrónico:

ajms66@gmail.com

Capítulo 1

INTRODUCCIÓN A AJAX

A pesar de que AJAX sea un término de reciente aparición en el mundo del desarrollo Web, no se trata realmente de una nueva tecnología.

Las siglas AJAX, Asynchronous JavaScript And XML, hacen referencia a un conjunto de tecnologías de uso común en la Web que, combinadas adecuadamente, permiten mejorar enormemente la experiencia del usuario con las aplicaciones de Internet.

Esto, sin duda, representa una gran noticia para un programador Web, pues, al estar basado en tecnologías y estándares utilizados habitualmente, la curva de aprendizaje para incorporar AJAX al desarrollo de aplicaciones se reduce notablemente, y le evita el coste, tanto en tiempo como en esfuerzo, de aprender nuevos lenguajes, librerías o herramientas de programación.

1.1 CONTEXTO DE UTILIZACIÓN DE AJAX

Hasta la llegada de AJAX, el proceso de generación de las páginas que forman la vista de la aplicación a partir de datos extraídos del backend, y su posterior envío al cliente, se realizaba desde algún componente del servidor como una operación única e indivisible.

Esto implicaba, por ejemplo, que una vez cargada la página en el navegador, si el usuario realizaba una operación sobre ésta que requiriera la actualización de tan sólo una parte de la misma con nuevos datos del servidor, el navegador se viera obligado a lanzar de nuevo una petición al componente del

servidor para que éste volviera a realizar de nuevo todo el proceso de obtención de datos, construcción de la vista completa y su envío posterior al cliente, proceso que es conocido habitualmente como **recarga de la página**.

En un escenario donde la vista está formada por un elevado número de datos y elementos gráficos, la cantidad de información que hay que transmitir por la red durante la recarga de la página puede llegar a ser muy elevada, lo cual se traduce en varios segundos de espera para el usuario; demasiado tiempo para actualizar una pequeña porción de la página.

Son precisamente este tipo de situaciones las que AJAX puede ayudarnos a solucionar, al permitir que se actualicen ciertas partes de una página Web sin necesidad de recargar todo el contenido de la misma.

1.2 ¿QUÉ ES AJAX?

El término AJAX hace referencia a un mecanismo de combinación de tecnologías y estándares de cliente, consistente en la solicitud asíncrona de datos al servidor desde una página Web y la utilización de éstos para actualizar una parte de la misma, **sin obligar al navegador a realizar una recarga completa de toda la página**.

Además de transmitirse una menor cantidad de información por la red, al enviarse en la respuesta únicamente los datos que hay que actualizar, la realización de la operación en modo asíncrono permite al usuario de la aplicación seguir trabajando sobre la interfaz mientras tiene lugar el proceso de recuperación de los datos actualizados; como se puede sospechar, esto supone una enorme reducción del tiempo de espera para el usuario, percibiendo éste la sensación de estar trabajando sobre una aplicación de escritorio.

Pensemos, por ejemplo, en el siguiente escenario:

Se dispone de una página Web en la que además de una serie de elementos gráficos, como imágenes y menús, tenemos un formulario mediante el que vamos a solicitar una serie de datos al usuario.

Entre estos datos, necesitamos que nos proporcione la provincia y localidad a la que pertenece, para lo cual utilizaremos unos controles de tipo lista donde se podrá elegir estos datos entre un conjunto de valores dados. La idea es que, una vez elegida la provincia en la primera lista, se cargue la lista de localidades con los nombres de todas aquellas pertenecientes a la provincia elegida, facilitando así al usuario la elección de ese dato y se evitan incoherencias (figura 1).

En una aplicación Web tradicional, la carga de las localidades en la segunda lista supondría tener que volver a solicitar la página de nuevo al servidor y volver a cargar en el navegador las imágenes, menús y demás elementos que componen la vista. Aplicando la solución AJAX, un bloque de código JavaScript que se ejecute en el navegador al producirse la selección de un elemento de la lista podría encargarse de solicitar en segundo plano al servidor los datos de las localidades mientras el usuario continúa trabajando sobre la página.

Una vez recibidos los datos, otro bloque de código JavaScript se encargaría de actualizar el contenido de la lista a partir de la información recibida, manteniendo intactos el resto de los componentes de la página, lo que sin duda alguna aumentará notablemente la sensación de velocidad de la aplicación.

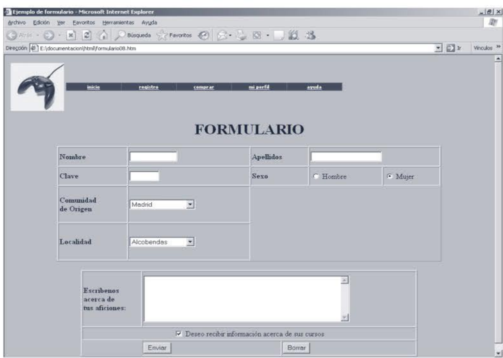


Fig. 1. *Formulario para solicitud de datos*

Otro ejemplo clásico de aplicación basada en AJAX es el Google maps. Se trata de una aplicación de tipo callejero que permite al usuario desplazarse por el plano de una localidad, ampliando y reduciendo las distintas vistas del mismo (figura 2). La imagen que forma la vista actual es realmente un puzzle de pequeñas imágenes de información distribuidas a modo de matriz bidimensional.

Cuando el usuario se desplaza por el mapa, se ejecuta en el cliente una función de JavaScript que solicita de forma asíncrona al servidor el subconjunto de imágenes necesario para actualizar la vista, evitando la recarga de todas las imágenes en la página y permitiendo al usuario seguir interactuando con la misma mientras se recuperan los datos.

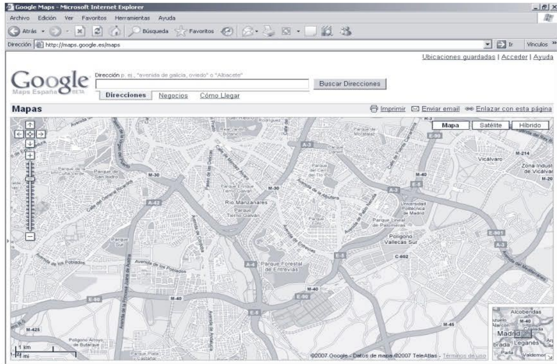


Fig. 2. *Aspecto de Google maps*

1.3 LAS TECNOLOGÍAS AJAX

Básicamente, el mecanismo de funcionamiento de una aplicación AJAX consiste en realizar peticiones HTTP de forma asíncrona al servidor desde la página Web cliente, utilizando los datos recibidos en la respuesta para actualizar determinadas partes de la página.

Para poder realizar estas operaciones, las aplicaciones AJAX se apoyan en las siguientes tecnologías:

- **XHTML y CSS**. La interfaz gráfica de una aplicación AJAX es una página Web cargada en un navegador. XHTML y CSS son los dos estándares definidos por el W3C para la construcción de páginas Web; mientras que XHTML se basa en la utilización de un conjunto de marcas o etiquetas para la construcción de la página, el estándar CSS define una serie de propiedades de estilo que pueden aplicarse sobre las etiquetas XHTML, a fin de mejorar sus capacidades de presentación.

- **JavaScript**. Las peticiones HTTP que lanza la página Web en modo asíncrono al servidor son realizadas por un bloque de script implementado con cualquier lenguaje capaz de ser interpretado por el navegador; este lenguaje es, en la gran mayoría de los casos, JavaScript.

Los bloques de código JavaScript que forman la aplicación AJAX se encuentran embebidos dentro de la propia página Web, o bien en archivos independientes con extensión .js que son referenciados desde ésta. Estos bloques de código son traducidos y ejecutados por el navegador , bien cuando se produce la carga de la página en él o bien cuando tiene lugar alguna acción del usuario sobre la misma. Mediante código JavaScript las aplicaciones AJAX pueden realizar solicitudes al servidor desde la página Web, recuperar los datos enviados en la respuesta y modificar el aspecto de la interfaz.

- **XML**. Aunque podría utilizarse cualquier otro formato basado en texto, cuando la aplicación del servidor tiene que enviar una serie de datos de forma estructurada al cliente, XML resulta la solución más práctica y sencilla, ya que podemos manipular fácilmente un documento de estas características y extraer sus datos desde el código JavaScript cliente.

- **DOM**. El Modelo de Objeto de Documento (DOM) proporciona un mecanismo estándar para acceder desde código a la información contenida en un documento de texto basado en etiquetas.

  Mediante el DOM, tanto páginas Web como documentos XML pueden ser tratados como un conjunto de objetos organizados de forma jerárquica, cuyas propiedades y métodos pueden ser utilizados desde código JavaScript para modificar el contenido de la página en un caso, o para leer los datos de la respuesta en otro.

- **El objeto XMLHttpRequest**. Se trata del componente fundamental de una aplicación AJAX. **A través de sus propiedades y métodos es posible lanzar peticiones en modo asíncrono al servidor y acceder a la cadena de texto enviada en la respuesta**. Para poderlo utilizar, deberá ser previamente instanciado desde la aplicación.

Las anteriores tecnologías y componentes constituyen el núcleo fundamental de AJAX, sin embargo, estas aplicaciones se apoyan también para su funcionamiento en los siguientes estándares y tecnologías:

- **HTTP**. Al igual que el propio navegador, el objeto XMLHttpRequest utiliza el protocolo HTTP para realizar las solicitudes al servidor, manejando también las respuestas recibidas mediante este protocolo.

- **Tecnologías de servidor**. Una aplicación AJAX no tendría sentido sin la existencia de un programa en el servidor que atendiera las peticiones enviadas desde la aplicación y devolviera resultados al cliente.

  Las mismas tecnologías que se utilizan para crear procesos en el servidor en una aplicación Web convencional, como Java EE, ASP.NET o PHP, pueden utilizarse igualmente en el caso de que la capa cliente esté basada en AJAX. Dado que éste es un libro enfocado a la utilización de AJAX con aplicaciones Java EE, todos los ejemplos de código de servidor que se presenten serán implementados con esta tecnología.

1.4 PRIMERA APLICACIÓN AJAX

Seguidamente, vamos a desarrollar una sencilla aplicación AJAX de ejemplo que nos va a servir para comprender la mecánica de funcionamiento de este tipo de programas y comprender el papel que juegan las distintas tecnologías mencionadas anteriormente.

La aplicación consistirá en una página Web, mediante la que se solicitará al usuario la elección del título de un libro entre una serie de valores presentados dentro de una lista desplegable. Cuando el usuario seleccione uno de estos títulos, se deberá mostrar en la parte central de la página un texto descriptivo sobre el libro elegido (figura 3).

Fig. 3. *Aspecto de la página de ejemplo*

1.4.1 Descripción de la aplicación

La vista de la aplicación está formada por una página XHTML. A fin de simplificar el desarrollo, los títulos de los libros serán incluidos en el propio código XHTML de la página.

Por otro lado, la lógica de servidor será implementada mediante un servlet que mantendrá en un array los textos asociados a cada libro. Cuando el usuario seleccione un título en la lista que aparece en la página XHTML, se invocará al servlet pasándole como dato el índice del libro seleccionado, valor que será utilizado para localizar el texto asociado en el array y enviárselo al cliente.

A continuación, vamos a analizar los distintos bloques que forman la aplicación a fin de comprender su funcionamiento.

1.4.2 Código de servidor

Como acabamos de indicar, el código de servidor de esta aplicación está implementado mediante un servlet que genera una respuesta a partir de la información contenida en un array. Esta respuesta estará formada por un texto plano que se corresponderá con la descripción asociada al título cuyo índice ha sido enviado en la petición.

El siguiente listado corresponde al código de este servlet:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ComentarioLibro extends HttpServlet {
    protected void service (HttpServletRequest request,
                            HttpServletResponse response)
                                 throws ServletException,
                                  IOException {
        response.setContentType("text/plain");
        PrintWriter out = response.getWriter();
        //array de descripciones
        String [] comentarios=
        {"Requiere conocimientos básicos de programación
                                   orientada a objetos",
        "Puede construir fácilmente aplicaciones para
                                   la Web",
        "Aprenderá rápidamente los principales trucos
```

```
                                        de Ajax",
          "Introduce las principales tecnologías de
                                        la plataforma"};
          int sel;
         //tit es el nombre del parámetro enviado en
         //la petición
          sel=Integer.parseInt(request.getParameter("tit"));
          out.println(comentarios[sel]);
          out.close();
     }
}
```

1.4.3 La vista cliente

La vista cliente está formada por una página XHTML estática. Además de las etiquetas, en esta página se incluye el código JavaScript que implementa la funcionalidad cliente de la aplicación AJAX.

Para facilitar la comprensión de la vista cliente, el siguiente listado muestra el contenido HTML de la página, omitiéndose el código JavaScript:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                                  Transitional//EN">
<html>
  <head>
    <title>Página de información sobre libros</title>
    <script language="JavaScript">
        //aquí iría el código JavaScript para AJAX
    </script>
  </head>
  <body bgcolor="#8beff1">
     <center>
         <table cellspacing="30">
             <tr>
                 <td><img src="libros.jpg" /></td>
                 <td><h1>Información sobre libros</h1></td>
             </tr>
         </table>
      <table width="30%">
          <tr><td><b>Elija título de libro:</b></td></tr>
          <tr>
              <td >
```

```
                    <select id="titulo"
                             onchange="buscarComentario();">
                        <option>- Seleccione título -</option>
                        <option value="0">Programación con C#
                        </option>
                        <option value="1">ASP.NET</option>
                        <option value="2">AJAX en un día
                        </option>
                        <option value="3">JAVA EE</option>
                    </select>
                </td>
            </tr>
            <tr>
                <td>
                    <br/><br/>
                    <div id="comentario"></div>
                </td>
            </tr>
        </table>
       </center>
    </body>
  </html>
```

Como se puede ver, la lista con los títulos de los libros es generada mediante un control XHTML de tipo *select*. Cada elemento <option> del control incluye en su atributo *value* un número que se corresponde con la posición dentro del array del servidor donde se encuentra el texto descriptivo asociado a ese título.

Así mismo, podemos comprobar la existencia de una etiqueta de tipo <div>, cuyo identificador es “comentario”, que será utilizada para mostrar el texto descriptivo del título seleccionado.

En cuanto a la conexión entre la vista XHTML y el código script de cliente, observamos que el control <select> incluye un manejador de evento *onchange* con una llamada a la función de JavaScript *buscarComentario()*. Esta función estará definida dentro del bloque <script> de la página y será invocada en el momento en que se produzca la selección de un elemento dentro de la lista, debiendo ser la encargada de poner en marcha todo el mecanismo de ejecución AJAX en el cliente.

1.4.4 Código de script de cliente

Todo el proceso de ejecución de una aplicación AJAX está controlado por código JavaScript que se ejecuta en el cliente. A grandes rasgos, este código debe encargarse de solicitar unos datos al servidor y, una vez obtenidos, insertarlos en la parte de la página destinada a mostrar los resultados.

Esta sencilla aplicación, pero práctica a efectos didácticos, está estructurada en tres funciones cuyo código se muestra en el siguiente listado y que serán explicadas en los siguientes subapartados:

```
var xhr;
function buscarComentario(){
      //creación del objeto XMLHttpRequest
      try{
           xhr=new ActiveXObject("Microsoft.XMLHttp");
           enviaPeticion();
      }
      catch(ex){
           alert("Su navegador no soporta este objeto");
      }
}
function enviaPeticion(){
      //configuración de los parámetros de la
      //petición y envío de la misma
      var lista=document.getElementById("titulo");
      var tit=lista.options[lista.selectedIndex].value;
      xhr.open("GET","comentariolibro?tit="+tit,true);
      xhr.onreadystatechange=procesaDatos;
      xhr.send(null);
}
function procesaDatos(){
      //tratamiento de la respuesta
      if(xhr.readyState==4)
                document.getElementById("comentario").
                    innerHTML="<i>"+xhr.responseText+"</i>";
}
```

1.4.4.1 CREACIÓN DEL OBJETO XMLHTTPREQUEST

La función *buscarComentario()* representa el punto de arranque de la aplicación AJAX. La primera tarea que realiza esta función es la creación del

objeto XMLHttpRequest, cuya instancia es almacenada en la variable pública "xhr" a fin de que pueda ser utilizada por el resto de las funciones del programa.

La instrucción de creación del objeto es encerrada dentro de un bloque *try*, de modo que si se produce un error en esta línea se mostrará un mensaje de aviso al usuario y no se ejecutará el resto del código.

Tras la creación del objeto XMLHttpRequest se invoca a la función *enviaPeticion()* para continuar con el proceso de ejecución de la aplicación.

1.4.4.2 ENVÍO DE LA PETICIÓN HTTP

La función *enviaPeticion()* se encarga de realizar la petición de datos al servidor utilizando el objeto XMLHttpRequest. Aunque más adelante analizaremos con detenimiento este objeto, vamos a comentar brevemente las propiedades y métodos que se utilizan en esta función, ya que representan la parte más importante de la aplicación:

- **open()**. Mediante este método el cliente se prepara para realizar la petición al servidor. Puede recibir hasta cinco parámetros, siendo los tres primeros los más importantes; en este orden:

    - **método de envío**. Método de envío de datos utilizado por la petición HTTP. Puede ser "GET" o "POST".

    - **url**. Este parámetro contiene una cadena de caracteres que representa la URL del recurso que se solicita en la petición. En este caso se trata de la URL del servlet "comentariolibro", al que se pasa un parámetro llamado "tit" que contiene el código asociado al título del libro seleccionado y que es obtenido con las dos primeras instrucciones de la función.

    - **modo asíncrono**. Indica si la operación se realizará en modo síncrono (*false*) o asíncrono (*true*).

- **onreadystatechange**. Propiedad en la que se indica la función JavaScript que va a procesar los datos de la respuesta.

- **send()**. El método *open()* no realiza la petición al servidor, simplemente prepara al cliente para ello, es la llamada al método *send()* la que lleva a cabo el proceso según los parámetros

configurados en *open()*. Posteriormente, veremos el significado del parámetro que se pasa al método, en este ejemplo su valor es *null*.

1.4.4.3 PROCESAMIENTO DE LA RESPUESTA

La última de las funciones de la aplicación de ejemplo es *procesaDatos()*. Esta es la función indicada en la propiedad *onreadystatechange*, y es la encargada de la recepción y procesamiento de los datos enviados por el servidor en la respuesta.

Cuando la petición se realiza en modo asíncrono, como es el caso de esta aplicación, la función *procesaDatos()* es invocada varias veces por el objeto XMLHttpRequest desde el momento en que se realiza la petición hasta que la respuesta ha sido recibida en el cliente.

Normalmente, al programador sólo le interesará que se ejecute su código una vez que se haya completado la recepción de los datos en cliente. Para comprobar en qué momento estarán disponibles estos datos, utilizamos la propiedad *readyState* del objeto XMLHttpRequest.

Una vez disponibles, el acceso a los datos de la respuesta enviada por el servlet se lleva a cabo utilizando las distintas propiedades del objeto XMLHttpRequest, según sea el formato que se haya dado a los datos de dicha respuesta. En este caso, dado que el servlet envía los datos como texto plano, debemos utilizar la propiedad *responseText* para recuperar la información, información que es presentada dentro del elemento <div> de la página XHTML utilizando las propiedades y métodos DOM:

```
document.getElementById("comentario").
            innerHTML="<i>"+xhr.responseText+"</i>";
```

Como se puede apreciar, el método DOM *getElementById()* nos devuelve el objeto HTML cuyo identificador se le proporciona como parámetro. A partir de la propiedad *innerHTML* del objeto podemos acceder a su contenido HTML, pudiendo así añadir información en su interior. Además de esta propiedad, las etiquetas HTML proporcionan otras propiedades que permiten alterar las características de la etiqueta, como, por ejemplo, *style*, que nos da acceso a sus propiedades de estilo.

Este Capítulo ha servido para ilustrarnos los fundamentos de la implementación de una aplicación AJAX básica. En posteriores capítulos profundizaremos en el estudio de las diferentes tecnologías y componentes implicados en su desarrollo.

1.5 APLICACIONES AJAX MULTINAVEGADOR

Si la aplicación de ejemplo que hemos creado anteriormente la ejecutamos con un navegador distinto a Internet Explorer, nos encontraremos con una desagradable sorpresa y es que la aplicación no funcionará, mostrándose un cuadro de diálogo con el mensaje "Su navegador no soporta este objeto" al realizar la selección de un elemento en la lista (figura 4).

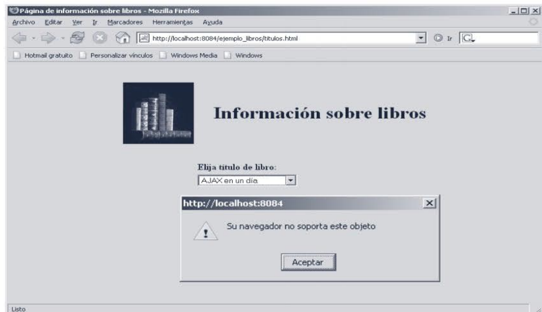


Fig. 4. *Error al ejecutar la aplicación en Firefox*

El problema se encuentra en la función que crea el objeto XMLHttpRequest, pues es precisamente esta instrucción la que provoca el error, dado que el objeto **ActiveXObject únicamente es reconocido por el navegador Internet Explorer**.

El objeto ActiveXObject es utilizado por Internet Explorer para crear instancias de componentes COM registrados en la máquina cliente, a partir de la cadena de registro de los mismos. En el caso de XMLHttpRequest, el componente COM que lo implementa es Microsoft.XMLHttp, aunque es muy posible que el cliente disponga además de versiones más modernas de éste, como la MSXM2.XMLHttp, la MSXM2.XMLHttp.3.0 o incluso la MSXM2.XMLHttp.5.0.

El resto de los navegadores más comúnmente utilizados por los usuarios de Internet, como Opera, Firefox, Safari e incluso Internet Explorer a partir de la versión 7, implementan XMLHttpRequest como un objeto nativo, lo que significa que su creación se debe llevar a cabo instanciando directamente la clase mediante el operador new:

```
xhr=new XMLHttpRequest();
```

Ésta es la manera en que el W3C, organismo encargado de regular las especificaciones para la Web, recomienda que debe crearse este objeto. Así pues, es de suponer que todas las futuras versiones de navegadores que se desarrollen en adelante soporten este mecanismo de creación.

1.5.1 Compatibilidad de código en todos los navegadores

Mientras sigan utilizándose navegadores Internet Explorer versión 6 y anteriores, las aplicaciones AJAX están obligadas a incluir cierta lógica que garantice la compatibilidad del código en todos los tipos de navegadores, lo que implica combinar en una misma función las dos formas analizadas de crear el objeto XMLHttpRequest.

Para ello, bastará con comprobar qué tipo de objeto nativo soporta el navegador donde se está ejecutando el código, ActiveXObject o XMLHttpRequest. Esto puede realizarse consultando las propiedades del mismo nombre del objeto window, ya que cuando un navegador soporta uno de estos objetos lo expone como una propiedad de window.

En el caso de IE6 y versiones anteriores la expresión:

window.ActiveXObject

contendrá un valor distinto de *null* y *undefined*, por lo que al ser utilizada como condición en una instrucción condicional de tipo *if* **el resultado de la evaluación será *true***. Lo mismo puede decirse para el resto de navegadores de la expresión:

window.XMLHttpRequest

Así pues, la estructura de código que habría que utilizar para garantizar la compatibilidad de la aplicación AJAX en todos los navegadores debería ser:

```
if(window.ActiveXObject){

        //Navegador IE o versiones anteriores

}

else if (window.XMLHttpRequest){

        //Resto de navegadores

}
```

else{

//Navegador sin soporte AJAX

}

En ciertas versiones del navegador, pudiera ocurrir que, aun implementando XMLHttpRequest como objeto nativo, no estuviera expuesto como propiedad del objeto window. En este caso habría que comprobar si el navegador soporta el objeto a través del operador *typeof* de JavaScript, en cuyo caso la expresión:

typeof XMLHttpRequest

devolverá un valor distinto de *undefined*.

Ahora podemos reescribir la función *buscarComentario()* del ejercicio anterior para garantizar la compatibilidad del código con prácticamente todas las versiones de navegadores utilizados por los usuarios de Internet, quedando como se indica en el siguiente listado:

```
function buscarComentario(){
     if(window.ActiveXObject){
         //navegador IE
         xhr=new ActiveXObject("Microsoft.XMLHttp");
     }
     else if((window.XMLHttpRequest) ||
                    (typeof XMLHttpRequest)!=undefined){
         //navegadores Firefox, Opera y Safari
         xhr=new XMLHttpRequest();
     }
     else{
         //navegadores sin soporte AJAX
         alert("Su navegador no tiene soporte para AJAX");
         return;
     }
     enviaPeticion();
}
```

1.6 MANIPULAR DATOS EN FORMATO XML

En el ejercicio AJAX de ejemplo que acabamos de desarrollar se ha utilizado la propiedad *responseText* del objeto XMLHttpRequest para acceder a la cadena de texto recibida en la respuesta del servidor.

Cuando estos datos van a ser tratados como texto plano el uso de esta propiedad resulta adecuado, sin embargo, hay muchas otras circunstancias en las que la información enviada en la respuesta debe tener cierta estructura, a fin de que los datos que la componen puedan ser recuperados de una forma individualizada por el código de script del cliente. En estos casos, el servidor **debe enviar los datos al cliente en un formato que permita su manipulación, este formato es XML**.

Cuando el servidor envía los datos como un documento XML bien formado, puede utilizarse la propiedad *responseXML* del objeto XMLHttpRequest para recuperar dicho documento y poder manipularlo desde el código JavaScript cliente utilizando el **Modelo de Objeto Documento (DOM)**, cuyo estudio será abordado en el próximo capítulo.

Como ejemplo de aplicación de esto, vamos a realizar una nueva versión de la aplicación de libros en la que incluiremos una mejora, consistente en obtener a partir del título seleccionado en la lista no sólo un comentario asociado al libro sino también su precio, mostrándose los datos en una tabla HTML tal y como se ve reflejado en la figura 5.

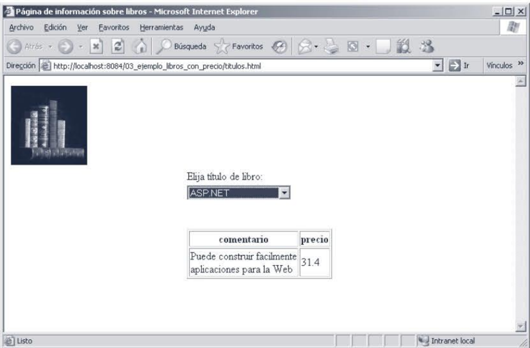


Fig. 5. *Nuevo aspecto de la aplicación de libros*

Para poder mantener esta información, el servlet incluirá un nuevo array en el que se almacenen los precios asociados a los libros.

A fin de poder enviar los datos al cliente de una forma estructurada, serán formateados como una cadena de texto XML que tendrá la siguiente estructura:

```
<?xml version="1.0"?>
<libro>
      <comentario>....</comentario>
      <precio>...</precio>
</libro>
```

El código del servlet se muestra en el siguiente listado:

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

public class Comentariolibro extends HttpServlet {
    protected void service (HttpServletRequest request,
              HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/xml;charset=UTF-8");
        PrintWriter out = response.getWriter();
        //textos de los comentarios
        String [] comentarios=
        {"Requiere conocimientos básicos de programación
                               orientada a objetos",
        "Puede construir fácilmente aplicaciones
                               para la Web",
        "Aprenderá rápidamente los principales
                               trucos de Ajax",
        "Introduce las principales tecnologías
                               de la plataforma"};
        //precios de cada libro
        String [] precios={"23.5","31.4","32.0","27.5"};
        int seleccionado=
             Integer.parseInt(request.getParameter("tit"));
```

```
            //formación del documento XML de respuesta
            String textoXML="<?xml version='1.0'?>";
            textoXML+="<libro>";
            textoXML+="<comentario>"+comentarios[seleccionado]+
                                         "</comentario>";
            textoXML+="<precio>"+
                           precios[seleccionado]+"</precio>";
            textoXML+="</libro>";
            //inserción de los datos XML en la respuesta
            out.println(textoXML);
            out.close();
        }
    }
```

Desde el punto de vista del cliente, tendremos que añadir una serie de instrucciones a la función *procesaDatos()* que, a partir del documento XML recuperado, extraigan la información del mismo y generen una tabla HTML con los datos para después mostrarla en la página, operación que se realizará, como se ha comentado anteriormente, utilizando el DOM.

El código de la función quedará como se indica en el siguiente listado:

```
function procesaDatos(){
     if(xhr.readyState==4){
          var resp=xhr.responseXML;
          var libro=resp.getElementsByTagName("libro").
                                  item(0);
          //recupera la colección de elementos
          //hijos de libro
          var elementos=libro.childNodes;
          var textoHTML="<table border='1'>";
          textoHTML+="<tr>";
          //genera la fila con los nombres
          //de los elementos
          for(var i=0;i<elementos.length;i++){
               textoHTML+="<th>"+elementos.item(i).nodeName+
                      "</th>";
          }
          textoHTML+="</tr>";
```

```
                textoHTML+="<tr>";
                //genera la fila con los datos del
                //libro incluidos en el documento
                for(var i=0;i<elementos.length;i++){
                    textoHTML+="<td>"+
                          elementos.item(i).firstChild.nodeValue+
                                                   "</td>";
                }
                textoHTML+="</tr></table>";
                //muestra la tabla en la página
                document.getElementById("info").innerHTML=
                                                      textoHTML;
            }
        }
```

Más adelante, analizaremos el Modelo de Objeto Documento y comprenderemos mejor algunas de las instrucciones que aparecen en el listado anterior. Por ahora, hay que comentar que el código de la función *procesaDatos()* se basa en generar una tabla con dos filas; en la primera de ellas se muestran los nombres de todos los subelementos de <libro>, mientras que en la segunda se cargan los valores contenidos en dichos elementos.

Para realizar estas operaciones, el código recorre la colección de elementos hijos de <libro>, por lo que la estructura podría servir no sólo para obtener la información de un documento de estas características en el que cada libro tiene un comentario y un precio, sino que, en general, podría ser aplicado para cualquier número de elementos hijos.

Capítulo 2

PROCESO DE EJECUCIÓN DE UNA APLICACIÓN AJAX

Después de lo visto hasta el momento, el lector ya debe tener una idea bastante aproximada del funcionamiento de una aplicación AJAX y del rol que desempeña cada una de las tecnologías implicadas.

En este capítulo vamos a exponer las distintas fases que podemos distinguir en la ejecución de una aplicación AJAX y a analizar los procesos que tienen lugar en cada una de ellas, prestando especial atención al estudio de las propiedades y métodos del objeto XMLHttpRequest y de las interfaces más importantes que forman el Modelo de Objeto Documento (DOM).

2.1 EVENTOS EN UNA PÁGINA WEB Y MANEJADORES

Una de las principales características de las aplicaciones de escritorio modernas es la de disponer de una interfaz rica en elementos gráficos con la que el usuario pueda interaccionar, codificándose bloques de código para que puedan ser ejecutados como respuesta a las distintas acciones de usuario o eventos que se producen sobre la interfaz.

La principal finalidad de una aplicación AJAX es simular este comportamiento en entorno Web, gestionando los eventos a través de los llamados **manejadores de evento**.

Los manejadores de evento no son más que atributos proporcionados por las etiquetas XHTML, a través de los cuales se pueden asociar los posibles eventos que se produzcan sobre la etiqueta, con funciones de JavaScript definidas por el programador para ser ejecutadas como respuesta a dichos eventos (figura 6).

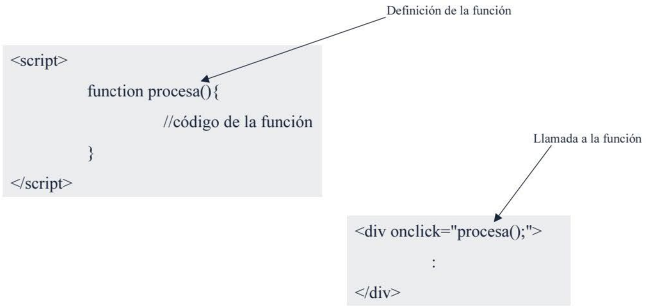


Fig. 6. *Ejemplo de gestión de evento*

Estas funciones pueden estar definidas en la propia página o en algún archivo .js externo referenciado desde la misma.

2.2 FASES EN LA EJECUCIÓN DE UNA APLICACIÓN AJAX

Normalmente, el proceso de ejecución de una aplicación AJAX se desencadena al producirse un evento sobre la página Web, como el clic de un botón, la selección de un elemento en una lista o la carga de la propia página.

Durante este proceso de ejecución pueden distinguirse las siguientes fases o etapas:

1. **Creación y configuración del objeto XMLHttpRequest**. El primer paso a realizar cuando se produce el evento que desencadena la ejecución de una aplicación AJAX consiste en obtener el objeto XMLHttpRequest. Como hemos visto en los ejemplos presentados en el capítulo anterior, esto implica añadir cierta lógica con código JavaScript que garantice la correcta creación del objeto, independientemente del tipo y versión de navegador en el que se esté ejecutando la página.

Una vez creado el objeto, deben configurarse una serie de parámetros del mismo, como la URL del recurso a solicitar o la función que va a procesar la respuesta.

2. **Realización de la petición**. Tras configurar los parámetros adecuados del objeto XMLHttpRequest, se procede a lanzar la petición al servidor, operación ésta que puede realizarse en modo síncrono o asíncrono, siendo este último el modo de funcionamiento mayoritariamente utilizado por las aplicaciones AJAX.

3. **Procesamiento de la petición en el servidor**. El servidor recibe la petición y ejecuta el componente correspondiente que, a partir de los datos recibidos, deberá realizar algún tipo de procesamiento, incluyendo consultas a bases de datos, y generar una respuesta con los resultados obtenidos.

   Este componente del servidor puede ser un servlet, una página JSP o ASP, un archivo de PHP o cualquier otro tipo de aplicación que se pueda ejecutar en un servidor Web.

   Cuando la petición se realiza en **modo asíncrono**, mientras el componente del servidor se está ejecutando para realizar su función, **el usuario puede seguir interactuando con la página sin necesidad de mantenerse bloqueado a la espera de recibir la respuesta**.

4. **Recepción de los datos de respuesta**. Una vez completada la ejecución del código de servidor, se envía una respuesta HTTP al cliente con los resultados obtenidos en el formato adecuado para su manipulación. En ese momento, el navegador invoca a la función de retrollamada definida por el objeto XMLHttpRequest.

5. **Manipulación de la página cliente**. A partir de los datos recibidos en la respuesta y mediante código JavaScript de cliente, se modifican las distintas zonas de la página XHTML que sea necesario actualizar.

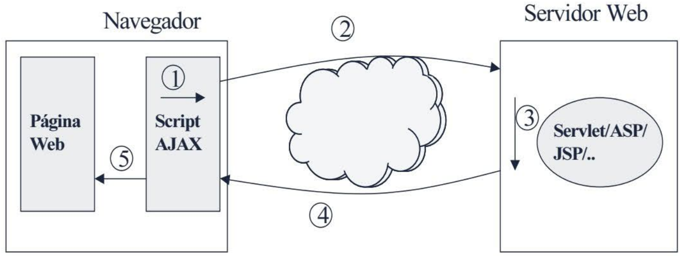


Fig. 7. *Ciclo de vida de una aplicación AJAX*

2.3 EL OBJETO XMLHTTPREQUEST

Como se ha venido indicando, el objeto XMLHttpRequest representa la pieza fundamental de una aplicación AJAX, ya que a través de sus propiedades y métodos se ejecuta y controla el proceso de petición asíncrona al servidor y la correspondiente manipulación de resultados en el cliente.

Es por ello que en esta sección vamos a realizar un estudio más detallado de este objeto, analizando con detenimiento sus principales propiedades y métodos.

2.3.1 Los orígenes del objeto XMLHttpRequest

Como el resto de las tecnologías que forman AJAX, el objeto XMLHttpRequest no es ningún elemento novedoso propio de los navegadores de última generación.

El primer navegador que empezó a proporcionar soporte para este objeto fue Internet Explorer 5.0, el cual incorporaba una librería llamada MSXML que incluía toda una serie de objetos e interfaces, basados en tecnología COM/ActiveX, para trabajar con XML desde aplicaciones JavaScript que se ejecutasen en ese navegador. Uno de los objetos incluidos en esa librería era el XMLHttp.

XMLHttp era un componente que permitía realizar peticiones HTTP desde cualquier aplicación que se pudiera ejecutar en un sistema operativo Windows. Por tanto, su uso no se limitaba a páginas Web cargadas en Internet Explorer, también podía ser utilizado, por ejemplo, desde aplicaciones de escritorio escritas en Visual Basic.

A partir de ahí, el objeto XMLHttp empezó a adquirir una gran popularidad, animando al resto de los fabricantes de navegadores a incluir en sus productos sus propias implementaciones de este objeto. Incluso se le cambió el nombre, pasándose a llamar XMLHttpRequest en las implementaciones proporcionadas por los restantes tipos de navegadores.

Actualmente, la mayoría de navegadores soportan este objeto, aunque como vimos en el capítulo anterior, **la forma de crearlo puede variar de un tipo de navegador a otro**. Incluso puede haber ligeras variaciones en las propiedades y métodos soportados en cada navegador, de ahí que el W3C esté realizando un esfuerzo de estandarización del objeto XMLHttpRequest, especificando tanto la interfaz que debe exponer como la forma en que ha de ser creado.

2.3.2 Miembros de la interfaz

A fin de tener un conocimiento más global y preciso del funcionamiento del objeto XMLHttpRequest, vamos a presentar las distintas propiedades y métodos que según el W3C debe exponer este objeto, analizándolos dentro del contexto de utilización de los mismos.

2.3.2.1 PREPARACIÓN DE LA PETICIÓN

Tras la creación del objeto XMLHttpRequest, el siguiente paso a realizar por una aplicación AJAX consiste en preparar el objeto para la realización de la petición, operación que se realiza invocando al método *open()* del mismo.

Según la interfaz definida por el W3C, el método *open()* se encuentra sobrecargado, proporcionándose cuatro versiones de este método. La versión más completa contempla la utilización de cinco parámetros en la llamada a *open()*:

void open (in DOMString method, in DOMString uri, in boolean async,

in DOMString user, in DOMString password)

Como se puede comprobar, todos los parámetros vienen precedidos por la palabra "in", lo que significa que deben ser parámetros de entrada, es decir, proporcionados en la llamada. El significado de estos parámetros es el siguiente:

- **method**. Se trata de una cadena de caracteres que establece el tipo de petición que se va a realizar. Los más utilizados son:
  - **GET**. Se utiliza habitualmente para obtener datos del servidor, por ejemplo, es el tipo de petición realizada

cuando escribimos directamente la dirección del recurso en la barra de direcciones del navegador, también cuando la solicitud se hace mediante un enlace de hipervínculo.

En caso de que se deba suministrar algún parámetro en una petición "GET", se deberá incluir como parte de la URL del recurso solicitado, siguiendo el siguiente formato:

url?nombre1=valor1&nombre2=valor2

Cada parámetro se incluye como una pareja nombre-valor y se separa del resto por el carácter "&".

- **POST**. Suele emplearse en aquellas peticiones en las que se deben enviar al servidor un conjunto de datos recogidos a través de un formulario HTML. Estos datos viajarán en el cuerpo de la petición y, en el caso concreto de una aplicación AJAX, **deberán ser incluidos en la llamada al método *send()*** del objeto XMLHttpRequest, para lo cual deberán ser ensamblados en una cadena de caracteres siguiendo el formato anterior de parejas nombre-valor:

  nombre1=valor1&nombre2=valor2

- **url**. Cadena de caracteres que representa la dirección relativa del programa de servidor que se va a solicitar. Como hemos indicado anteriormente, en el caso de las peticiones de tipo "GET", los parámetros deberán anexarse a esta URL siguiendo el formato especificado.

- **async**. Este tercer parámetro consiste en un dato de tipo lógico o booleano, utilizado para establecer el modo en que se va a procesar la petición, esto es, de forma asíncrona (*true*) o síncrona (*false*).

- **user**. Es una cadena de caracteres que representa el nombre del usuario que se debe proporcionar para ser autenticados en el servidor, en caso de que sea necesario.

- **password**. Al igual que el anterior, este parámetro se utiliza para la autenticación de un usuario, representando en este caso la contraseña del mismo.

Además de la comentada, el objeto XMLHttpRequest proporciona otras tres versiones del método *open()*:

open (in DOMString method, in DOMString uri)

open (in DOMString method, in DOMString uri, in boolean async)

open (in DOMString method, in DOMString uri, in boolean async, in DOMString user)

2.3.2.2 DEFINICIÓN DE ENCABEZADOS DE LA PETICIÓN

Una petición HTTP está formada por una cabecera y un cuerpo. La cabecera, además de la URL del recurso que hay que solicitar, puede incluir otros datos adicionales que permitan informar al servidor sobre determinadas características del cliente. Estos datos, conocidos como encabezados o datos de cabecera, se definen mediante un nombre y un valor asociado.

Para establecer los encabezados que van a ser incluidos en una petición AJAX utilizaremos el método *setRequestHeader()* del objeto XMLHttpRequest:

void setRequestHeader(in DOMString header, in DOMString value)

El primero de los parámetros es una cadena de caracteres que representa el nombre del encabezado, mientras que el segundo es el valor asignado a éste.

Aunque el W3C define una serie de encabezados tipo como "ACCEPT", "USER-AGENT", etc., que permiten establecer ciertas características estándares de los clientes, tales como el tipo de datos aceptados o la cadena identificativa, una aplicación AJAX es libre de poder enviar encabezados personalizados siempre y cuando la aplicación del servidor sea capaz de interpretarlos.

El siguiente ejemplo utiliza el encabezado "ACCEPT" para indicar al servidor en la cabecera de la petición que el cliente es capaz de interpretar como respuesta datos en formato XML y texto plano:

```
//xhr es una referencia válida al objeto XMLHttpRequest
xhr.setRequestHeader("ACCEPT", "text/xml;text/plain");
```

2.3.2.3 DEFINICIÓN DE LA FUNCIÓN DE RETROLLAMADA

Cuando la petición realizada desde la aplicación AJAX se lleva a cabo en modo asíncrono, es necesario indicarle al objeto XMLHttpRequest qué función se

encargará del procesamiento de la respuesta cuando ésta esté disponible en el cliente.

La propiedad *onreadystatechange* es la responsable de esta tarea:

attribute EventListener onreadystatechange

El tipo de la propiedad está definido como EventListener, lo que significa que **debe contener la definición de una función manejadora de evento**.

En este caso, el evento que genera la llamada a la función definida por *onreadystatechange* es el **evento cambio de estado**. Este evento se produce varias veces desde el momento en que se envía la petición hasta que se recibe la respuesta en el cliente, por lo que será necesario incluir cierta lógica adicional en la función que garantice que las instrucciones encargadas del procesamiento de la respuesta se ejecuten únicamente cuando ésta esté disponible.

2.3.2.4 ESTADO DE LA PETICIÓN

El tipo de control del que hablábamos en el punto anterior se basa en comprobar el estado de la petición en curso, tarea que puede realizarse consultando la propiedad de sólo lectura *readyState* del objeto XMLHttpRequest dentro de la función de retrollamada, pues es precisamente el cambio de valor de esta propiedad lo que genera el evento *onreadystatechange*.

Los posibles valores que puede tomar *readyState* durante el ciclo de vida de una petición AJAX son:

- **0**. El objeto XMLHttpRequest se ha creado pero aún no se ha configurado la petición.
- **1**. La petición se ha configurado pero aún no se ha enviado.
- **2**. La petición se acaba de enviar, aunque aún no se ha recibido respuesta.
- **3**. Se ha recibido la cabecera de la respuesta pero no el cuerpo.
- **4**. Se ha recibido el cuerpo de la respuesta. Es el momento en que ésta puede procesarse.

La mayoría de las ocasiones sólo nos interesará conocer si el estado de la petición ha adquirido el valor 4 para ver si es posible procesar los datos de la respuesta.

Respecto a esta propiedad, hemos de saber también que los valores 2 y 3 no son reconocidos por algunas versiones de navegadores.

2.3.2.5 ENVÍO DE LA PETICIÓN

Una vez configurados todos los parámetros que afectan a la petición, procederemos a su envío mediante la llamada al método *send()* del objeto XMLHttpRequest. Según la interfaz definida por el W3C para el objeto, se proporcionarán tres versiones de este método:

void send()

void send(in DOMString data)

void send(in Document data)

El parámetro indicado en la segunda versión del método representa los datos que van a ser enviados al servidor en el cuerpo de la respuesta, por lo que sólo se utilizará cuando la petición sea de tipo "POST".

En este caso, los datos son enviados como una cadena de caracteres formada por la unión de parejas *nombre=valor*. Se debe indicar además en el encabezado "content-type" que el tipo de información que figura en el cuerpo corresponde a datos procedentes de un formulario HTML, para lo cual deberá establecerse su valor a "application/x-www-form-urlencoded".

Tal y como se indica en la versión tercera del método, también es posible enviar en el cuerpo de la petición un documento XML, indicando en este caso el objeto Document que apunta al nodo raíz del árbol de objetos DOM que representa al documento. Este tipo de envío resulta especialmente interesante cuando desde una aplicación AJAX se quiere invocar a un servicio Web, ya que **el intercambio de datos entre estos componentes y sus aplicaciones clientes se realiza en formato XML**.

PRÁCTICA 2.1. ENVÍO DE DATOS DE UN FORMULARIO

Descripción

En esta práctica vamos a crear una página Web que recoja una serie de datos introducidos por el usuario en un formulario HTML y los envíe a un servlet para su procesamiento (figura 8).

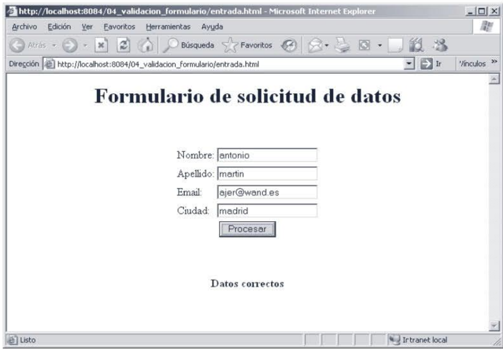


Fig. 8. *Aspecto de la página Web después de procesar los datos*

Desarrollo

Como queremos centrarnos únicamente en el aspecto del envío de los datos, la misión del servlet consistirá simplemente en comprobar que todos los datos recibidos son distintos de una cadena vacía. La respuesta generada desde el servidor consistirá en un mensaje de texto con el resultado de la comprobación, mensaje que será mostrado en la propia página Web.

Listado

El código XHTML de la página será el indicado en el siguiente listado:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<html>
  <head>
    <title></title>
    <script>
      //aquí el código AJAX
```

```
  </script>
</head>
<body>
    <center>
        <h1>Formulario de solicitud de datos</h1>
        <br/>
        <br/>
        <form action="validador">
        <table>
            <tr>
                <td>
                    Nombre:
                </td>
                <td>
                    <input type="text" name="txtnombre"/>
                </td>
            </tr>
            <tr>
                <td>
                    Apellido:
                </td>
                <td>
                    <input type="text" name="txtapellido"/>
                </td>
            </tr>
            <tr>
                <td>
                    Email:
                </td>
                <td>
                    <input type="text" name="txtemail"/>
                </td>
            </tr>
            <tr>
                <td>
                    Ciudad:
                </td>
                <td>
                    <input type="text" name="txtciudad"/>
                </td>
            </tr>
```

```
                <tr>
                    <td colspan="2" align="center">
                        <input type="button" name="btnvalidar"
                             onclick="validar();"
                      value="Procesar"/>
                    </td>
                </tr>
            </table>
            </form>
            <br/>
            <br/>
            <div id="resultado">

            </div>
        </center>
    </body>
</html>
```

Según podemos observar, cada uno de los controles HTML de la página incluye un valor para el atributo *name* que representa el nombre con el que será enviado el dato del control. Así mismo, debajo del formulario se incluye un elemento <div>, cuyo valor de atributo *id* es "resultado", donde se mostrará el contenido de la respuesta enviada por el servidor.

El envío de los datos se realizará desde la aplicación AJAX mediante "POST". Para ello, se deberá implementar una función que recupere todos los valores introducidos por el usuario en los distintos controles del formulario y genere una cadena de texto con los pares nombre-valor.

En nuestro ejemplo esta función se llama *obtenerDatos()* y, como veremos en el listado que mostraremos a continuación, tanto el nombre del control como el valor del mismo deberán ser codificados adecuadamente para que la cadena de caracteres resultante pueda ser procesada por el servidor, lo que implica sustituir espacios y caracteres especiales por su equivalente codificación URL. Esta tarea la realiza de manera automática la función *encodeURIComponent()* de JavaScript.

A continuación, se muestra todo el código AJAX de la aplicación:

```
<script>
    var xhr;
    function validar(){
       if(window.ActiveXObject){
            xhr=new ActiveXObject("Microsoft.XMLHttp");
```

```
      }
      else if((window.XMLHttpRequest) ||
                   (typeof XMLHttpRequest)!=undefined){
               xhr=new XMLHttpRequest();
      }
      else{
           alert("Su navegador no tiene soporte para AJAX");
           return;
      }
      enviaPeticion();
   }
   function enviaPeticion(){
       xhr.open("POST",document.forms[0].action,true);
       xhr.onreadystatechange=procesaResultado;
       //definición del tipo de contenido del cuerpo
       //para formularios HTML
       xhr.setRequestHeader("Content-Type",
                   "application/x-www-form-urlencoded");
       var datos=obtenerDatos();
       //se envían los datos en el cuerpo de la petición
       xhr.send(datos);
    }
    function obtenerDatos(){
        //se obtiene la colección de controles que hay
        //en el formulario
        var controles=document.forms[0].elements;
        var datos=new Array();
        var cad="";
        for(var i=0;i<controles.length;i++){
            cad=encodeURIComponent(controles[i].name)+"=";
            cad+=encodeURIComponent(controles[i].value);
            datos.push(cad);
        }
        //se forma la cadena con el método join() del array
        //para evitar múltiples concatenaciones
        cad=datos.join("&");
        return cad;
    }
    function procesaResultado(){
        if(xhr.readyState==4){
            document.getElementById("resultado").
```

```
                    innerHTML=xhr.responseText;
            }
        }
    </script>
```

En cuanto a la aplicación del servidor, estará implementada, como hemos dicho, mediante un servlet, cuya URL está especificada en el atributo *action* del formulario. Seguidamente se muestra el código de este componente:

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Validador extends HttpServlet {
    protected void service(HttpServletRequest request,
                           HttpServletResponse response)
                throws ServletException, IOException {
        response.setContentType("text/plain;charset=UTF-8");
        PrintWriter out = response.getWriter();
        String resultado="<b>Datos correctos</b>";
        //recupera todos los nombre de parámetros
        //recibidos en la petición
        Enumeration<String> nombres=
                          request.getParameterNames();
        while(nombres.hasMoreElements()){
            if(request.getParameter(nombres.nextElement())
                                            .equals("")){
                resultado="<b>Error en los datos. ";
                resultado+="Debe rellenar todos los
                                            campos</b>";
            }
        }
        out.println(resultado);
        out.close();
    }
}
```

2.3.2.6 ENCABEZADO DE LA RESPUESTA

Al igual que las peticiones, las respuestas HTTP están formadas por una cabecera y un cuerpo. En este caso, el cuerpo incluye los datos generados desde la

aplicación del servidor en el formato de texto correspondiente, mientras que la cabecera proporciona información adicional sobre la respuesta, como es el tipo de formato del cuerpo, cookies enviadas por el servidor, etc.

Si queremos acceder a los datos de encabezado de una respuesta desde una aplicación AJAX podemos hacer uso del método *getResponseHeader()* del objeto XMLHttpRequest. El formato de este método es el siguiente:

DOMString getResponseHeader(in DOMString header)

Al igual que en el caso de las peticiones, los datos de encabezados de respuesta se caracterizan por un nombre y un valor asociado. A partir del nombre del encabezado proporcionado como parámetro, *getResponseHeader()* devuelve el valor correspondiente.

Además de *getResponseHeader()*, el objeto XMLHttpRequest proporciona un segundo método para recuperar encabezados de una respuesta, se trata de *getAllResponseHeaders()*. Este método devuelve una cadena de texto con todos los encabezados recibidos. Por ejemplo, si en la práctica anterior quisiéramos mostrar en una segunda capa <div> de la página Web la cadena de texto con todos los encabezados recibidos (figura 9), deberíamos incluir en la función *procesaResultado()* las instrucciones que aparecen sombreadas:

```
function procesaResultado(){
      if(xhr.readyState==4){
           document.getElementById("resultado").
                   innerHTML=xhr.responseText;
           var encabezados="<h2>Encabezados de
                                        respuesta</h2>";
           encabezados+=xhr.getAllResponseHeaders();
           document.getElementById("encabezados").
                   innerHTML=encabezados;
      }
}
```

siendo "encabezados" el valor del atributo *id* del nuevo elemento <div>:

```
<div id="resultado">
</div>
<div id="encabezados">
</div>
```

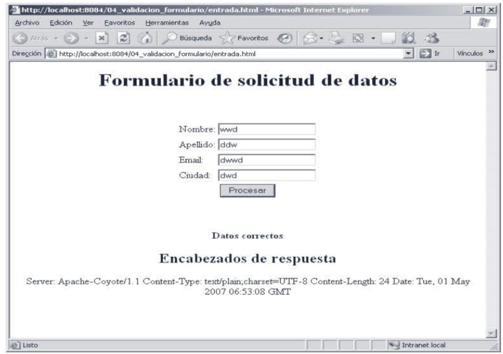


Fig. 9. *Visualización de encabezados enviados por el servidor*

2.3.2.7 ACCESO AL CONTENIDO DE LA RESPUESTA

Según hemos tenido oportunidad de comprobar en los ejercicios que se han ido desarrollando hasta el momento, el objeto XMLHttpRequest es capaz de recuperar la respuesta recibida desde el servidor como una cadena de caracteres (propiedad *responseText*) o como un documento XML (propiedad *responseXML*), en función del formato que se haya dado a ésta desde el servidor.

La sintaxis definida por el W3C para estas propiedades es la que se muestra a continuación:

readonly attribute DOMString responseText;

readonly attribute Document responseXML;

Como podemos comprobar, en el caso de *responseXML* vemos que el contenido de esta propiedad es un objeto Document, lo que permite manipular su contenido mediante el DOM.

Aunque XML es el estándar de *facto* para la transmisión de información estructurada en la Web, como veremos en el capítulo siguiente no siempre resulta la solución más adecuada para el envío de grandes cantidades de datos desde el servidor al cliente, **pudiendo utilizarse otros formatos alternativos y mucho más eficientes para las aplicaciones AJAX como es el caso de JSON**. Como ya

estudiaremos en su momento, los datos JSON están formateados en una cadena de texto plano, debiendo emplearse la propiedad *responseText* para su recuperación.

2.3.2.8 ESTADO DE LA RESPUESTA

En ocasiones puede suceder que debido a algún error interno que se produce en el servidor durante el procesamiento de petición, la respuesta enviada por éste no contendrá los datos esperados. Por ejemplo, si se produce una excepción durante la ejecución del servlet, la propiedad *responseText* **contendrá el volcado de pila de la excepción**.

A través de la propiedad *status* del objeto XMLHttpRequest es posible conocer el estado de la respuesta, lo que nos permite saber si se ha producido algún tipo de error en el servidor o si, por el contrario, los datos recibidos son correctos:

readonly attribute unsigned short status;

El valor contenido en esta propiedad representa el código de estado del servidor. Un código de estado igual a 200 significa que la petición se ha procesado correctamente, por lo que simplemente será necesario preguntar por este valor para saber si la información recibida es correcta.

Por otro lado, la propiedad *statusText* nos devuelve, en caso de error, un mensaje descriptivo asociado al mismo.

Así pues, si queremos preparar la función *procesaResultado()* de la práctica 2.1 para que en caso de error muestre un cuadro de diálogo con el mensaje descriptivo y no intente procesar los resultados de la respuesta, debería realizarse la siguiente modificación en la misma:

```
function procesaResultado(){
     if(xhr.readyState==4){
          //sólo si se ha recibido correctamente
          //la respuesta se procederá a procesarla
          if(xhr.status==200){
               document.getElementById("resultado").
                    innerHTML=xhr.responseText;
               var encabezados="<h2>Encabezados de
                                        respuesta</h2>";
               encabezados+=xhr.getAllResponseHeaders();
               document.getElementById("encabezados").
                    innerHTML=encabezados;
          }
```

```
            else
                alert(xhr.statusText);
        }
}
```

PRÁCTICA 2.2. CAMPO DE TEXTO CON AUTOSUGERENCIA

Descripción

Quizás alguna vez nos hayamos encontrado con una página en la que se nos solicita un determinado dato y, según vamos escribiendo en el campo de texto, nos va apareciendo en el mismo una palabra sugerida por la propia página a fin de que, en caso de que sea esa la palabra o frase que queremos escribir, podamos seleccionarla directamente y evitar el tener que teclear todos sus caracteres.

Esta facilidad la encontramos, por ejemplo, en una de las modalidades del buscador Google, conocida como Google suggest (figura 10).

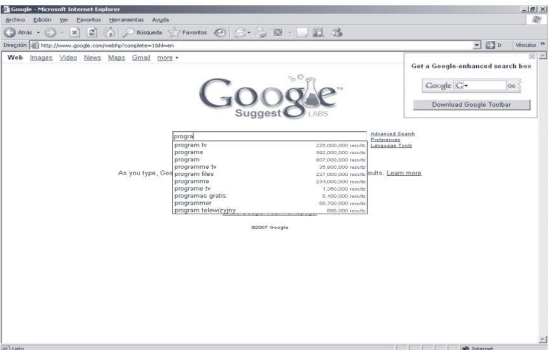


Fig. 10. *Página de inicio de Google suggest*

Google suggest es un tipo de buscador Web que ofrece sugerencias a la hora de realizar una búsqueda. Así, según vamos escribiendo la palabra o frase dentro del control de texto, nos va apareciendo una lista con los elementos más buscados y que coinciden con los caracteres introducidos.

En esta práctica vamos a desarrollar una versión reducida de una página de autosugerencias. Su aspecto se muestra en la figura 11.

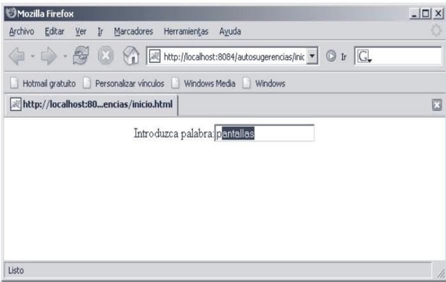


Fig. 11. *Página de autosugerencias*

A fin de simplificar la aplicación y centrarnos en el aspecto de la búsqueda de sugerencias, la página contará sólo con un único campo de texto en el que, según se van escribiendo caracteres, se mostrará en el interior del control la palabra o frase sugerida, seleccionándose los caracteres aún no escritos.

Desarrollo

Se trata de una aplicación típica AJAX en la que, mientras el usuario escribe caracteres en el control, un código JavaScript cliente debe comunicar con una aplicación del servidor para recuperar, a partir de los caracteres escritos por el usuario, la palabra sugerida desde alguna fuente de datos.

La aplicación del servidor estará formada por un servlet que accederá a una base de datos donde estarán almacenadas todas las posibles palabras a sugerir (figura 12).

id	palabra
1	programación
2	programación orienta a objetos
3	programas de televisión
4	píldoras informativas
5	procesos formativos
6	procesadores de texto
7	pantallas
8	precintos
9	precios netos
10	proceso de ventas

Fig. 12. *Tabla con valores de ejemplo*

Para este ejemplo, la base de datos será muy simple, dispone de una única tabla (“palabras”) con dos campos: “id” para almacenar el identificador de la palabra, que será de tipo autonumérico, y “palabra”, que almacenará el texto de la palabra o frase sugerida. En nuestro ejemplo, el acceso a la base de datos se realiza a través del driver puente jdbc-odbc, siendo “palabras” el DSN asociado a dicha base de datos.

Desde el punto de vista del cliente, se lanzará la petición al servidor cada vez que se introduce un carácter alfanumérico en el campo de texto, lo que se controla desde una función que es invocada al producirse el evento *onkeyup* en el control.

Listado

A continuación se muestran los listados del código tanto del servlet como de la página cliente.

Servlet “generador”

```
package servlets;

import java.io.*;
import java.net.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Generador extends HttpServlet {
    protected void service(HttpServletRequest request,
              HttpServletResponse response)
                   throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        String dato=request.getParameter("texto");
        Connection cn=null;
        Statement st=null;
        ResultSet rs=null;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            cn=DriverManager.
                   getConnection("jdbc:odbc:palabras");
            st=cn.createStatement();
```

```
                //busca la palabra/frase que se ajuste
                //al texto recibido, ordenándose alfabéticamente
                //los resultados
                String sql="Select palabra from palabras where ";
                sql+="palabra like '"+dato;
                sql+="%' order by palabra";
                rs=st.executeQuery(sql);
                if(rs.next()){
                    out.println(rs.getString("palabra"));
                }
                else{
                    out.println("");
                }
            }
            catch(Exception e){e.printStackTrace();}
            out.close();
        }
    }
```

Página inicio.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                     Transitional//EN">
<html>
  <head>
     <script language="javascript">
        var xhr;
        function busqueda(ev){
            //comprueba si es un caraácter alfanumérico
            if((ev.keyCode>=48 && ev.keyCode<=57) ||
               (ev.keyCode>=65 && ev.keyCode<=90)){
                if(window.ActiveXObject){
                    xhr=new ActiveXObject(
                          "Microsoft.XMLHttp");
                }
                else if((window.XMLHttpRequest) ||
                          (typeof
                   XMLHttpRequest)!=undefined){
                    xhr=new XMLHttpRequest();
                }
                else{
                    alert("Su navegador no tiene
```

```
                                    soporte para AJAX");
                return;
            }
            localizaPalabra();
        }
    }
    function localizaPalabra(){
            //recupera el texto introducido en el campo
            //de texto y lo pasa como parámetro en la URL
            //al servlet
            var caja=document.getElementById("texto");
            var texto=caja.value;
            xhr.open("GET",
                       "generador?texto="+texto,true);
            xhr.onreadystatechange=procesaDatos;
            xhr.send(null);
    }
    function procesaDatos(){
        if(xhr.readyState==4){
            var resp=xhr.responseText;
            var caja=document.getElementById("texto");
            //si se ha recibido una palabra de respuesta
            //se introduce en el control y se seleccionan
            //los caracteres aún no tecleados
            if(resp!=""){
                var inicioSel=caja.value.length;
                caja.value=resp;
                caja.selectionStart=inicioSel;
                caja.selectionEnd=caja.value.length;
            }
        }
    }
  </script>
 </head>
 <body>
     <center>
         Introduzca palabra:<input type="text" id="texto"
                            onkeyup="busqueda(event)">
     </center>
 </body>
</html>
```

2.4 EL MODELO DE OBJETO DOCUMENTO (DOM)

Una de las tareas fundamentales a realizar por una aplicación AJAX consiste en acceder a los datos recibidos en la respuesta y, a partir de ellos, modificar parte del contenido de la página.

Cuando estos datos se encuentran en formato XML, la aplicación debe desplazarse por el documento de respuesta para localizar la información deseada y, una vez obtenida, desplazarse por el documento XHTML que representa la página Web cargada en el navegador, a fin de acceder al punto de la misma en donde hay que modificar el contenido. Es en estos desplazamiento a través de documentos de texto estructurados donde el DOM resulta de gran utilidad.

Todos los navegadores compatibles con DOM generan, a partir de un documento XHTML bien formado, un conjunto de objetos que representan las distintas partes del documento y que proporcionan una serie de propiedades y métodos para que, desde una aplicación JavaScript, se pueda acceder al contenido del mismo e incluso manipular su estructura. En el caso de respuestas en formato XML, será el propio objeto XMLHttpRequest el que genere el conjunto de objetos a partir de los datos recibidos en el documento.

Estos métodos y propiedades expuestos por los distintos objetos del documento no dependen de un tipo o versión de navegador específica, sino que se encuentran, por decirlo de alguna manera, **estandarizados por el W3C** en la especificación DOM.

Pues eso es precisamente el DOM, una especificación en la que se definen una serie de interfaces (interfaces DOM) con las propiedades y métodos que deben exponer los distintos objetos que componen un documento con estructura XML bien formado, ya sea XHTML o simplemente XML. Así pues, el conocimiento y comprensión del DOM resulta de enorme importancia para un programador AJAX, por lo que vamos a dedicar este apartado a estudiar las interfaces más importantes que componen este modelo y la forma de utilizarlas en la práctica.

2.4.1 Niveles DOM

El W3C ha establecido tres niveles de soporte del DOM:

- **Nivel 1**: proporciona las interfaces necesarias para manipular documentos XML. Se encuentra en estado de recomendación desde 1998.

- **Nivel 2**: además de las capacidades del nivel 1, incluye soporte para DTD, hojas de estilo y tratamientos de eventos. Adquirió el estado de recomendación en noviembre de 2000.

- **Nivel 3**: incluye algunas mejoras respecto al nivel 2. Adquirió el estado de recomendación en 2004.

La mayor parte de los navegadores utilizados actualmente proporcionan un amplio soporte para el DOM nivel 2.

2.4.2 Interfaces DOM

En la página *http://www.w3.org/dom* podemos encontrar toda la información referida a la especificación DOM, entre otras cosas, la definición formal de todas las interfaces que se incluyen en cada nivel. A fin de facilitar su estudio, vamos a centrarnos en aquellas de uso más habitual. La figura 13 muestra las interfaces DOM más importantes y la relación entre las mismas.

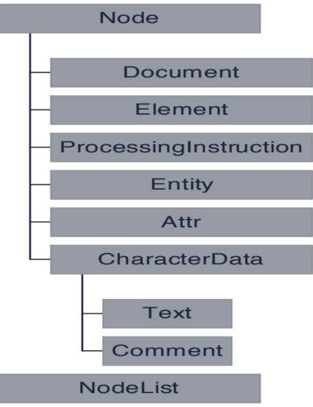


Fig. 13. *Principales interfaces DOM*

A continuación, comentamos brevemente sus características:

- **Node**. Representa cualquier nodo del árbol. Todos los componentes del documento son nodos del árbol, por lo que se les puede aplicar los métodos y propiedades definidos en esta interfaz al ser heredada por el resto de las interfaces.

- **Document**. Representa el nodo documento. Además de heredar los métodos y propiedades de Node, incorpora otros propios para modificar la estructura del documento (agregar nuevos nodos y eliminar existentes) y proporcionar información general del mismo.

- **Element**. Especifica un elemento del documento. Incorpora métodos y propiedades para acceder a los atributos del elemento y a sus descendientes.

- **Attr**. Representa un atributo. Dispone de propiedades para obtener los valores y nombres de los atributos.

- **Text**. Representa cualquier texto contenido en un elemento o atributo.

- **NodeList**. Representa una colección de nodos.

2.4.3 El árbol de objetos de un documento

Cuando un navegador carga en memoria un documento XML o XHTML y, tras comprobar la buena formación del mismo, genera en memoria un árbol de objetos con la información contenida en el mismo. Cada tipo de estructura contenida en el documento genera un tipo diferente de objeto que implementa la correspondiente interfaz DOM.

En el ejemplo que se muestra en la figura 14 podemos apreciar el árbol de objetos generado a partir del documento XML del ejemplo. Así es como la aplicación JavaScript "ve" los documentos.

Como se desprende de esta figura, todo documento genera un árbol de objetos cuyo nodo raíz es un objeto de tipo Document, dependerán de él el resto de los componentes del documento (instrucciones de procesamiento, elementos, comentarios, etc.).

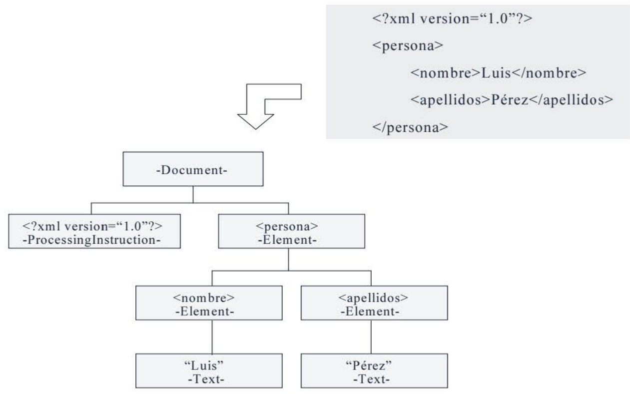


Fig. 14. *Documento XML y su árbol de objetos DOM*

2.4.4 DOM en la práctica

A continuación, vamos a estudiar los principales métodos y propiedades que proporcionan las interfaces DOM, y lo vamos a hacer analizando las principales operaciones que sobre un documento XML o XHTML se realizan en una aplicación.

2.4.4.1 OBTENCIÓN DEL OBJETO DOCUMENT

Dado que se trata del punto de entrada al documento, lo primero que suele hacer una aplicación AJAX que va a utilizar el DOM para operar sobre éste es obtener una referencia al objeto Document.

Cuando se trata del documento XML recibido en la respuesta generada desde la aplicación del servidor, es la misma propiedad *responseXML* del objeto XMLHttpRequest la que contiene una referencia a este objeto:

```
var doc = xhr.responseXML;
```

En el caso de la página Web, es el objeto Document del navegador el que proporciona la referencia al nodo raíz del documento:

```
var doc = Document;
```

2.4.4.2 REFERENCIA A LOS ELEMENTOS DEL DOCUMENTO

Partiendo de una referencia al objeto Document, podemos aplicar una serie de métodos incluidos en esta interfaz que nos permiten acceder a los distintos elementos contenidos en el mismo. Estos métodos son:

- **getElementsByTagName(nameTag)**. Se trata de uno de los métodos más utilizados del DOM. Dado un determinado nombre de etiqueta, nos devuelve la colección de elementos de ese tipo. La colección devuelta por este método es de tipo NodeList, esta interfaz proporciona la propiedad *length* y el método *item(index)*, que nos permitirán recorrer la colección desde código y acceder a cada uno de los elementos individuales que la componen.

  Por ejemplo, las siguientes instrucciones se encargarían de mostrar un mensaje indicando el número de elementos de tipo <email> que han sido recibidos en el documento XML de respuesta:

  ```
  var doc=xhr.responseXML;
  var emails=doc.getElementsByTagName("email");
  alert ("Total emails: "+emails.length);
  ```

- **getElementById(id)**. Se emplea cuando necesitamos acceder a ciertos elementos de un documento XHTML, pues nos devuelve una referencia a un elemento a partir del valor de su atributo *id*. Este método lo hemos utilizado ampliamente en los ejemplos y prácticas realizadas hasta el momento.

2.4.4.3 ACCESO AL CONTENIDO DE UN ELEMENTO

Una vez que se dispone de la referencia a un determinado elemento del documento, se puede acceder al contenido del mismo. La forma de hacerlo dependerá de que se trate de un documento XML, o bien del documento XHTML, cargado en el navegador.

2.4.4.3.1 Elementos XML

Según se desprende de la figura 14, el texto contenido dentro de un elemento XML es tratado por el DOM como un objeto independiente del elemento, siendo el único descendiente de éste. Así pues, para obtener una referencia a este objeto habrá que aplicar sobre el nodo elemento una propiedad que contenga su nodo hijo, esta propiedad es *firstChild*.

Una vez obtenida la referencia al nodo de texto, aplicaríamos la propiedad *nodeValue* definida en la interfaz Node para obtener el valor del texto.

El siguiente bloque de instrucciones mostraría todos los email recibidos en un documento XML de respuesta:

```
var doc=xhr.responseXML;
var emails=doc.getElementsByTagName("email");
var i;
for(i=0;i<emails.length;i++){
      alert(emails.item(i).firstChild.nodeValue;
}
```

2.4.4.3.2 Elementos XHTML

Como hemos venido apreciando en el código AJAX de los distintos ejemplos realizados, los elementos XHTML cuentan con una propiedad llamada *innerHTML* que permite acceder al contenido de la etiqueta.

Se trata de una propiedad de tipo texto que permite modificar dinámicamente el contenido de una etiqueta XHTML, incluyendo tags XHTML anidados en su interior que serán interpretados por el navegador.

2.4.4.4 ELEMENTOS HIJOS DE UN ELEMENTO

Un elemento XML o XHTML puede tener anidados otros subelementos en su interior. Se puede obtener una referencia a éstos a través de la propiedad *childNodes* definida en la interfaz Node.

La propiedad *childNodes* devuelve una colección NodeList con todos los nodos hijos del elemento, independientemente de que se trate de otros nodo elemento o de cualquier otro tipo (comentarios, instrucciones de procesamiento, etc.).

2.4.4.5 TIPO, NOMBRE Y VALOR DE UN NODO

Cuando se están manipulando documentos XML mediante el DOM, puede resultar útil en algunas ocasiones conocer el tipo de determinado nodo que está siendo referenciado. Para ello, la interfaz Node proporciona la propiedad *nodeType*, cuyos posibles valores se muestran en la tabla de la figura 15.

Tipo	Código
Element	1
Attribute	2
Text	3
CDATA section	4
Entity reference	5
Entity	6
Processing instruction	7
Comment	8
Document	9

Fig. 15. *Posibles tipos de un nodo XML*

El siguiente bloque de instrucciones mostraría el número total de nodos de tipo elemento que dependen de un determinado elemento referenciado por la variable “elm”:

```
var nodos=elm.childNodes;
var total=0;
for (var i=0;i<nodos.length;i++){
        if(nodos.item(i).nodeType==1){
                total++;
        }
}
alert ("Total de elementos hijos "+total);
```

Por su parte, la propiedad *nodeName* de un nodo nos proporciona el nombre del mismo. Esta propiedad resulta bastante útil cuando se está recorriendo una determinada colección de elementos con distinto nombre y se quiere localizar aquellos con un nombre específico.

Finalmente, para obtener el valor asociado a un determinado nodo podemos utilizar la propiedad *nodeValue* proporcionada por la interfaz Node. Como hemos visto anteriormente, esta propiedad puede utilizarse para acceder al contenido de los elementos en un documento XML.

La información contenida en la propiedad *nodeValue* dependerá del tipo de nodo sobre el que se aplique. En la tabla de la figura 16 se indica esta relación.

Tipo Nodo	Contenido
Atributo	Su valor
Texto	Su valor
Comentario	Su contenido
Instrucción de procesamiento	Su contenido
Resto	null

Fig. 16. *Contenido de la propiedad nodeValue según el tipo de nodo*

2.4.4.6 DESPLAZAMIENTO POR EL ÁRBOL DE OBJETOS

Una de las operaciones más habituales en una aplicación que manipula documentos XML es la realización de desplazamientos por el árbol del documento. Para ello la interfaz Node proporciona una serie de propiedades que, dada una referencia a un determinado nodo, nos permiten acceder al resto de nodos con los que está conectado el objeto. Estas propiedades son:

- **firstChild**. Contiene una referencia al primer nodo hijo. Si el nodo sobre el que se aplica la propiedad no tiene ningún descendiente, su valor será *null*.

- **lastChild**. Contiene una referencia al último nodo hijo. De la misma manera que la propiedad anterior, su valor será *null* si el objeto no tiene descendientes.

- **nextSibling**. Contiene una referencia al siguiente nodo que se encuentra en el mismo nivel que el objeto.

- **previousSibling**. Contiene una referencia al nodo anterior que se encuentra en el mismo nivel que el objeto.

- **parentNode**. Contiene una referencia al nodo padre del que desciende el objeto.

Además de estas propiedades, el método *hasChildNodes()* nos permite conocer si un determinado nodo tiene descendientes, devolviendo el valor *true* en caso afirmativo y *false* si no los tiene.

El siguiente bloque de instrucciones JavaScript implementa una función recursiva que, para un determinado documento XML de respuesta, muestra en un

cuadro de diálogo el nombre y contenido de cada uno de los elementos que lo forman:

```
var doc=xhr.responseXML;
busca(doc);
function busca(nodo){
      if(nodo.hasChildNodes()){
            for(var i=0;i<nodo.childNodes.length;i++){
                  busca(nodo.childNodes.item(i));
            }
      }
      //si es texto
      if(nodo.nodeType==3){
            alert(nodo.parentNode.nodeName+" : "+
                              nodo.nodeValue);
      }
}
```

2.4.4.7 ACCESO A LOS ATRIBUTOS DE UN ELEMENTO

De forma general, el acceso a los atributos de un elemento mediante DOM, tanto en documentos XML como XHTML, puede realizarse a través del método *getAttribute()* proporcionado por la interfaz Element:

getAttribute(nombre_atributo)

A partir del nombre del atributo, el método nos devuelve una cadena de caracteres con el valor del mismo para el elemento sobre el que se aplica. En caso de que el elemento no disponga de este atributo, el valor devuelto por *getAttribute()* será *null*.

El siguiente ejemplo mostraría en un cuadro de diálogo el precio de un libro cuyo valor se encuentra en el atributo “precio” del primer elemento <libro> recibido en el documento XML de respuesta:

```
var docRespuesta=xhr.responseXML;
var libro=docRespuesta.getElementsByTagName("libro").item(0);
alert("Precio del libro "+libro.getAttribute("precio"));
```

En el caso concreto de XHTML, los elementos exponen para cada uno de sus atributos una propiedad con el mismo nombre que permite acceder directamente desde código al valor del mismo. Esto lo hemos podido comprobar en

el ejemplo desarrollado en el capítulo 1, donde utilizábamos la propiedad *value* de los objetos de tipo <option> para acceder al atributo *value* del elemento.

El siguiente listado corresponde a una nueva versión del ejemplo anterior, en la que el precio del libro se almacena en un control XHTML de tipo *text* cuyo identificador es "resultado". Como se puede apreciar, el acceso al contenido del control se realiza a través de su atributo *value*:

```
var docRespuesta=xhr.responseXML;
var docPagina=document;
var libro=docRespuesta.getElementsByTagName("libro").item(0);
var texto=docPagina.getElementById("resultado");
texto.value=libro.getAttribute("precio");
```

2.4.4.8 MODIFICACIÓN DE LA ESTRUCTURA DE UN DOCUMENTO

Las interfaces DOM no solamente proporcionan miembros para acceder al contenido de un documento XML o XHTML, también proporcionan una serie de métodos que permiten alterar la estructura del mismo, permitiendo añadir nuevos componentes (elementos, comentarios, etc.) o eliminar otros ya existentes.

Esto resulta especialmente interesante para las aplicaciones AJAX, ampliando las posibilidades de las mismas de cara a la manipulación de las páginas XHTML, ya que, además de modificar el contenido de las etiquetas, podemos utilizar el DOM para añadir dinámicamente nuevos elementos a la página.

Básicamente, añadir un nuevo elemento a un documento XML o XHMTL requiere realizar dos operaciones:

- Crear el objeto elemento.
- Agregar el objeto al árbol del documento.

2.4.4.8.1 Creación de nuevos elementos

Para crear un nuevo objeto de tipo elemento debemos hacer uso del método *createElement()* definido en la interfaz Document:

createElement(nombre)

donde *nombre* es una cadena de texto con el nombre del elemento a crear.

En caso de que el elemento deba contener algún nodo de texto, éste deberá ser creado de manera independiente y anexado después al elemento. Para crear un nodo de texto, utilizaríamos el siguiente método incluido también en la interfaz Document:

createTextNode(cadena)

siendo el parámetro *cadena* el texto asociado al nodo.

El siguiente código de ejemplo crearía un nuevo elemento llamado "telefono" y un texto con el valor de éste, siendo "doc" la variable que contiene la referencia al objeto Document:

```
var tel = doc.createElement("telefono");
var valor = doc.createTextNode("911111144");
```

Tras la ejecución de estas instrucciones, los objetos elemento y texto recién creados ni forman parte aún de la estructura del documento ni tampoco están relacionados entre ellos. Estas operaciones se llevan a cabo en el paso siguiente.

2.4.4.8.2 Agregar objetos al árbol del documento

Para realizar esta tarea podemos recurrir a los siguientes métodos incluidos en la interfaz Node:

- **appendChild(nodo)**. Agrega el nodo especificado en el parámetro al conjunto de nodos hijos del nodo sobre el que se hace la llamada al método, situándolo como último elemento de dicho conjunto.

- **insertBefore(nodo, referencia)**. Agrega el nodo especificado en el parámetro al conjunto de nodos hijos del nodo sobre el que se hace la llamada al método, situándolo por delante del subelemento que se indica en referencia.

El siguiente bloque de instrucciones de ejemplo realiza dos tareas: por un lado, añade el texto al nuevo elemento <telefono> creado anteriormente y por otro, añade el elemento a la lista de descendientes del primer elemento de tipo <persona> existente en el documento, situándolo en el penúltimo lugar de esa lista:

```
tel.appendChild(valor);
var persona = doc.getElementsByTagName("persona").item(0);
persona.insertBefore(tel, persona.lastChild);
```

2.4.4.8.3 Ejemplo práctico: generación dinámica de lista de títulos

Vamos a presentar una nueva versión de la página de selección de libros desarrollada en el capítulo 1. Se trata de generar dinámicamente la lista de títulos presentada en la página a partir de datos proporcionados en el servidor, en vez de codificar dichos títulos directamente en el código XHTML.

Para ello, se implementará un nuevo servlet que se encargue de generar un documento XML con los títulos de los libros, los cuales estarán almacenados en un array dentro del propio servlet. A continuación, se muestra el código de este nuevo componente del servidor:

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

public class TitulosLibro extends HttpServlet {
    protected void service(HttpServletRequest request,
                    HttpServletResponse response)
                        throws  ServletException,  IOException
                {
        response.setContentType("text/xml;charset=UTF-8");
        PrintWriter out = response.getWriter();
         //array de títulos
        String [] titulos={"Programación con C#",
                          "ASP.NET",
                          "AJAX en un día",
                          "JAVA EE"};
        String textoXML="<?xml version='1.0'?>";
        textoXML+="<listado>";
        for(int i=0;i<titulos.length;i++){
            textoXML+=
             "<libro><titulo>"+titulos[i]+"</titulo>";
            textoXML+="<orden>"+i+"</orden></libro>";
        }
        textoXML+="</listado>";
        out.println(textoXML);
        out.close();
    }
}
```

El documento XML generado incluirá también el orden que ocupa el título en el array, a fin de que pueda ser utilizado por la página XHTML cliente para localizar el comentario asociado. La estructura de este documento es la siguiente:

```
<?xml version="1.0"?>
<listado>
      <libro>
            <titulo>...</titulo>
            <orden>...</orden>
      </libro>
      <libro>
            <titulo>...</titulo>
            <orden>...</orden>
      </libro>
      :
</listado>
```

Respecto a la página Web, ésta deberá comunicarse vía AJAX con el servidor en dos situaciones concretas: una al cargar la página en el cliente para recuperar los títulos de los libros, y la otra, tal como ya estaba establecida, al producirse la selección de uno de los elementos de la lista de títulos para recuperar el comentario correspondiente.

La primera de las peticiones AJAX se realiza en el evento asociado a la carga de la página en el navegador, para lo que se incluirá la llamada a la función correspondiente en el manejador de evento *onload* de la etiqueta <body>. El siguiente listado muestra el código XHTML de la página, indicando en fondo sombreado los cambios respecto a la primera versión:

```
<html>
 <head><script>...</script></head>
 <body onload="buscarTitulos();">
  <img src="libros.jpg"/>
     <center>
      <table width="30%">
          <tr><td>Elija título de libro:</td></tr>
          <tr>
              <td>
                  <select id="titulo"
                            onchange="buscarComentario();">
                  </select>
              </td>
```

```
                </tr>
                <tr>
                    <td>
                        <br/><br/>
                        <div id="comentario"></div>
                    </td>
                </tr>
            </table>
          </center>
      </body>
    </html>
```

En cuanto al código JavaScript de cliente, se han aislado las instrucciones para la creación del objeto XMLHttpRequest en una función independiente. Esta función será invocada desde cada una de la funciones manejadoras de evento en el momento en que necesiten crear el objeto para comunicarse con el servidor. Cada una de estas funciones, establecerá su propia configuración del objeto XMLHttpRequest y realizará la petición correspondiente.

Es la función *buscarTitulos()* la encargada de recuperar la lista de títulos del servidor y, mediante la utilización del DOM, de construir dinámicamente los elementos de tipo <option> y de añadirlos al árbol de objetos DOM que representa el documento XHTML cliente, concretamente, como descendientes del elemento <select>.

El siguiente listado corresponde con el código JavaScript de la página Web, indicándose en fondo sombreado los cambios respecto a la primera versión:

```
<script language="javascript">
        var xhr;
        function crearObjeto(){
            if(window.ActiveXObject){
                xhr=new ActiveXObject("Microsoft.XMLHttp");
            }
            else if((window.XMLHttpRequest) ||
             (typeof XMLHttpRequest)!=undefined){
                xhr=new XMLHttpRequest();
            }
            else{
                alert("Su navegador no tiene soporte
                                 para AJAX");
            }
```

```
}
function buscarComentario(){
    crearObjeto();
    //manipula la respuesta si se ha conseguido
    //crear el objeto
    if(xhr!=undefined){
        var titulo=document.getElementById("titulo");
        var tit=
            titulo.options[titulo.selectedIndex].
                                    value;
        xhr.open("GET",
            "comentariolibro?tit="+tit,true);
        xhr.onreadystatechange=presentaComentario;
        xhr.send(null);
    }
}
function buscarTitulos(){
    crearObjeto();
    if(xhr!=undefined){
        xhr.open("GET","tituloslibro",true);
        xhr.onreadystatechange=presentaTitulos;
        xhr.send(null);
    }
}
function presentaComentario(){
    if(xhr.readyState==4){
        document.getElementById("comentario").
                            innerHTML=xhr.responseText;
        //elimina la referencia al objeto
        xhr=undefined;
    }
}
function presentaTitulos(){
    if(xhr.readyState==4){
        var results=xhr.responseXML;
        var titulos=results.
                getElementsByTagName("titulo");
        var valores=
          results.getElementsByTagName("orden");
```

```
                //crea la primera opción de la lista
                //y la añade a ésta
                var opcionInicio=
                   document.createElement("option");
                var textoOpcion=
                   document.createTextNode(
                         "-Seleccione Título-");
                opcionInicio.appendChild(textoOpcion);
                var lista=document.getElementById("titulo");
                lista.appendChild(opcionInicio);
                //genera el resto de opciones de
                //la lista a partir de los datos
                //recibidos en la respuesta
                for(var elm = 0;elm < titulos.length;elm++){
                    var item=
                          document.createElement("option");
                    item.setAttribute(
                      "value",valores.item(elm).
                                 firstChild.nodeValue);
                    var texto=
                          document.createTextNode(titulos.
                             item(elm).firstChild.nodeValue);
                    item.appendChild(texto);
                    document.getElementById("titulo").
                                     appendChild(item);
                }
                //elimina la referencia al objeto
                xhr=undefined;
            }
        }
</script>
```

Capítulo 3

UTILIDADES AJAX

AJAX es un concepto de programación que, gracias a la enorme potencia, que ofrece en el desarrollo de interfaces Web, está siendo utilizado por un número cada vez más importante de desarrolladores.

Esto ha permitido que muchos de ellos, de forma organizada o independiente, hayan desarrollado sus librerías y utilidades para facilitar y mejorar el trabajo con esta tecnología. A lo largo de este capítulo vamos a estudiar algunas de esas utilidades más populares, llamadas algunas de ellas a convertirse en estándares de desarrollo AJAX.

Además de utilidades creadas por terceros, prestaremos también especial atención al empleo de técnicas de programación en AJAX que nos permitan simplificar los desarrollos y reutilizar código en aplicaciones. Por aquí será por donde comenzaremos el estudio de este capítulo.

3.1 ENCAPSULACIÓN DEL OBJETO XMLHTTPREQUEST

En los ejemplos y prácticas que hemos ido desarrollando a lo largo del libro, cada vez que hacíamos uso del objeto XMLHttpRequest en una página Web, debíamos proceder a comprobar el tipo de navegador en el que se estaba ejecutando la aplicación y a configurar cada una de las propiedades del objeto antes de realizar la petición.

Estas tareas, monótonas y algunas de ellas incómodas, pueden ser encapsuladas dentro de una nueva clase de objeto creada por el propio programador, a fin de poder ser utilizado en todas las aplicaciones y simplificar así el proceso de configuración y realización de peticiones asíncronas.

Así pues, más que una utilidad creada por terceros, la encapsulación del objeto XMLHttpRequest representa una buena práctica de programación que nos va a permitir reutilizar código y facilitar los desarrollos.

Aunque no existen unas reglas fijas para realizar esta tarea y cada programador puede realizar la encapsulación de funciones como mejor le convenga, en este punto vamos a presentar una posible solución que consistirá en crear un nuevo tipo de objeto que envuelva al propio objeto XMLHttpRequest y exponga una interfaz más sencilla e intuitiva que la de éste. El código de esta clase se incluirá en un archivo de código externo (.js), a fin de que pueda ser importado desde cualquier página XHTML o JSP.

3.1.1 La interfaz de la clase ObjetoAJAX

ObjetoAJAX es el nombre que le daremos a la clase del nuevo objeto que vamos a crear, cuya interfaz va a estar formada por los siguientes métodos:

- **enviar()**. Realiza el envío de la petición.
- **respuestaTexto()**. Devuelve la respuesta recibida en formato de texto plano.
- **respuestaXML()**. Devuelve la respuesta recibida en formato XML.
- **obtenerEncabezados()**. Devuelve todos los encabezados recibidos en la respuesta.
- **estado()**. Devuelve el estado HTTP de la respuesta.
- **textoEstado()**. Devuelve el texto asociado al estado HTTP de la respuesta.

Como podemos ver, se trata de una interfaz más simple que la del propio objeto XMLHttpRequest, manteniendo toda su funcionalidad. Seguidamente, vamos a analizar la implementación propuesta de esta interfaz y la forma en la que estos métodos se utilizan.

3.1.2 Implementación de la clase ObjetoAJAX

Dentro de un archivo de texto con extensión .js, por ejemplo utilidades.js, incluiremos el código completo de la clase. A continuación vamos a describir las distintas partes que la componen.

3.1.2.1 CONSTRUCTOR

El constructor de una clase se ejecuta cada vez que se crea una instancia utilizando el operador *new*. En él se incluirán las sentencias encargadas de crear el objeto XMLHttpRequest en función del tipo de navegador, de esta manera el programador no tendrá que preocuparse más de esta tarea.

Un constructor se implementa mediante una función cuyo nombre coincide con el de la clase de objeto que se está definiendo. En nuestro caso, el constructor de ObjetoAJAX quedaría de la siguiente manera:

```
function ObjetoAJAX(){
     var xhr;
     if(window.ActiveXObject){
          xhr=new ActiveXObject("Microsoft.XMLHttp");
     }
     else if((window.XMLHttpRequest) ||
               (typeof XMLHttpRequest)!=undefined){
          xhr=new XMLHttpRequest();
     }
     else{
          xhr=null;
     }
     //declaración de métodos de la clase
     this.enviar=m_enviar;
     this.respuestaTexto=m_texto;
     this.respuestaXML=m_XML;
     this.obtenerEncabezados=m_encabezados;
     this.estado=m_estado;
     this.textoEstado=m_textoEstado;
     //funciones para la implementación de los métodos
     :
}
```

En el código anterior podemos observar dos cosas, la primera es que dentro de la función que hace de constructor se incluirán también las demás funciones que

implementan el resto de métodos de la clase, tal y como lo exige la sintaxis de JavaScript.

El otro aspecto a destacar es el hecho de que el objeto XMLHttpRequest se incluye dentro de una variable definida dentro del propio constructor. Esto, por un lado, permite al resto de los métodos de la clase hacer uso del objeto y, por otro, evita que pueda ser manipulado desde el exterior, lo que podría desvirtuar el funcionamiento del propio objeto ObjetoAJAX.

3.1.2.2 EL ENVÍO DE LA PETICIÓN

Tanto las operaciones de configuración del objeto XMLHttpRequest previas al envío de la petición HTTP al servidor como el propio envío de ésta se encuentran encapsuladas en el método *enviar()* del nuevo objeto. De esta manera el programador no tendrá que invocar a distintas propiedades y métodos del objeto para poder realizar la petición, con una única llamada a este método se realizará todo el trabajo de configuración y petición AJAX.

El método *enviar()* tendrá el siguiente formato:

enviar(url, method, funcionRetorno, objForm)

siendo el significado de sus parámetros el siguiente:

- **url**. Representa la dirección del recurso del servidor que se va a solicitar.
- **method**. Indica el método de envío de la petición.
- **funcionRetorno**. Cadena de caracteres que representa el nombre de la función de retorno donde se implementarán las instrucciones para el tratamiento de la respuesta.
- **objForm**. Representa el objeto formulario HTML que contiene los datos de usuario que se enviarán en la petición, debiéndose indicar el valor *null* en este parámetro si no se van a enviar datos de ningún formulario.

Como se puede apreciar en la estructura de clase mostrada anteriormente, el método *enviar()* es implementado mediante la función interna *m_enviar()*. El siguiente listado corresponde al código de dicha función:

```
function m_enviar(url, method, funcionRetorno, objForm){
    var dataSend=null;
    //si el método es "POST" serializa los datos del
    //formulario
    if(method.toLowerCase()=="post"&&objForm!=null){
        dataSend=obtenerDatos(objForm);
    }
    //si el método es "GET" serializa los datos del
    //formulario y los anexa a la URL
    else if(method.toLowerCase()=="get"&&objForm!=null){
        var dataUrl="";
        dataUrl=obtenerDatos(objForm);
        if(url.indexOf("?")==-1){
            url+="?"+dataUrl;
        }
        else{
            url+="&"+dataUrl;
        }
    }
    //realiza siempre la petición en modo
    //asíncrono
    xhr.open(method,url,true);
    xhr.onreadystatechange=function(){
        //si la respuesta está disponible
        //se ejecuta la función de retorno
        if(xhr.readyState==4){
            eval(funcionRetorno+"("+")");
        }
    };
    if(objForm!=null){
        //establece el encabezado apropiado para el
        //envío de datos de un formulario
        xhr.setRequestHeader("Content-Type",
                "application/x-www-form-urlencoded");
    }
    xhr.send(dataSend);
}
:
```

Dado que la mayoría de las aplicaciones AJAX realizan las peticiones en modo asíncrono, el método *enviar()* utiliza siempre este sistema, evitándole al

programador tener que indicar esta circunstancia cada vez que va a hacer uso del objeto. Además, dado que en la inmensa mayoría de la ocasiones el código de la función de retrollamada sólo deberá ser ejecutado cuando la respuesta esté disponible, la llamada a esta función definida por el programador se realiza desde *onreadystatechange*, únicamente cuando se da esta situación, evitándole al programador tener que comprobar el estado de la petición antes de proceder a manipular la respuesta.

Por otro lado, podemos comprobar en el código anterior como, en función de qué tipo de método de envío de la petición se haya indicado, la cadena con los datos del formulario se incluye en la propia URL (método "GET") o como parámetro del método *send()* del objeto XMLHttpRequest (método "POST"). En cualquiera de los dos casos, la formación de la cadena se lleva a cabo a través de la función *obtenerDatos()* definida también en la clase:

```
:
function obtenerDatos(objForm){
   var controles=objForm.elements;
   var datos=new Array();
   var cad="";
   for(var i=0;i<controles.length;i++){
        cad=encodeURIComponent(controles[i].name)+"=";
        cad+=encodeURIComponent(controles[i].value);
        datos.push(cad);
   }
   //se forma la cadena con el método join() del array
   //para evitar múltiples concatenaciones
   cad=datos.join("&");
   return cad;
}
:
```

3.1.2.3 ACCESO A LA RESPUESTA

Para acceder a la información enviada por el servidor como respuesta a la petición, el objeto ObjetoAJAX define los métodos *respuestaTexto()*, *respuestaXML()* y *obtenerEncabezados()*, que devuelven la respuesta en formato de texto plano, la respuesta en formato XML y el array de encabezados, respectivamente.

Las funciones que realizan la implementación de estos métodos se encargan simplemente de devolver los valores contenidos en las propiedades del objeto XMLHttpRequest correspondientes:

```
:
function m_texto(){
    return xhr.responseText;
}
function m_XML(){
    return xhr.responseXML;
}
function m_encabezados(){
    return xhr.getAllResponseHeaders();
}
:
```

3.1.2.4 ESTADO DE LA RESPUESTA

El estado HTTP de la respuesta se obtiene mediante el método *estado()* de ObjetoAJAX, siendo *textoEstado()* el método encargado de proporcionarnos el mensaje de error generado por el servidor. Las funciones que implementan estos métodos simplemente derivan las llamadas a propiedades del objeto XMLHttpRequest:

```
:
function m_estado(){
    return xhr.status;
}
function m_textoEstado(){
    return xhr.statusText;
}
:
```

3.1.3 Utilización de la clase ObjetoAJAX

Para poder hacer uso de este tipo de objeto desde una página Web, tan sólo habrá que importar el archivo .js que contiene la definición de la clase, utilizando la siguiente sintaxis de la etiqueta <script>:

```
<script type=”text/javascript”

        language=”javascript”

        src=”directorio/nombre_archivo.js”>

</script>
```

siendo la ruta indicada en el atributo *src* la dirección del archivo .js, y puede utilizarse una *url* absoluta o relativa al directorio en el que se encuentra la página.

Una vez importado el archivo en la página, en un siguiente bloque <script> se podrá instanciar el objeto a través del operador *new* y hacer uso del mismo. Como ejemplo, vamos a presentar una nueva versión de la práctica 2.1 en la que se realizaba el envío de un formulario de datos al servidor desde una aplicación AJAX y se mostraban en la página Web los encabezados de la respuesta, utilizando esta vez ObjetoAJAX. El siguiente listado corresponde al código cliente de la página:

```
<!--importa el archivo con el código del objeto-->
<script src="utilidades.js" language="javascript"></script>
<script>
    var obj;
    function validar(){
        //crea una instancia del objeto y realiza la petición
        obj=new ObjetoAJAX();
        var oForm=document.forms[0];
        obj.enviar(oForm.action,oForm.method,
                          "procesaResultado",oForm);
    }
    //función de retrollamada
    function procesaResultado(){
        if(obj.estado()==200){
             document.getElementById("resultado").innerHTML=
                          obj.respuestaTexto();
             var encabezados=
                    "<h2>Encabezados de respuesta</h2>";
             encabezados+=obj.obtenerEncabezados();
             document.getElementById("encabezados").
                    innerHTML=encabezados;
        }
        else{
               alert(obj.textoEstado());
        }
    }
</script>
```

Podemos comprobar la enorme simplificación en el código script de cliente que supone la utilización de esta técnica.

3.2 JSON

Su facilidad de comprensión e independencia de plataformas y tecnologías han convertido a XML en el estándar universalmente aceptado para el intercambio de datos entre aplicaciones en la Web.

Sin embargo, la utilización de XML no está exenta de inconvenientes; su elevado ancho de banda y la lentitud en el procesamiento de los datos codificados en un documento con este formato pueden mermar el rendimiento de las aplicaciones, especialmente en aquellos contextos donde es necesario transmitir un alto volumen de información.

Este hecho ha llevado a los programadores de aplicaciones AJAX a experimentar con otras alternativas a XML para el envío de datos estructurados entre el servidor y el cliente. Es en este contexto en el que surge JSON.

3.2.1 Características de JSON

JSON son las siglas de **JavaScript Object Notation** y hace referencia a un nuevo formato de datos, cuya sintaxis se basa en definir objetos JavaScript como cadenas de texto que encapsulen los datos que van a ser transferidos entre el servidor y la aplicación AJAX cliente.

Este nuevo formato, mucho más compacto y ligero que XML, reduce el número de bytes que hay que transmitir respecto a éste, además, dado que la información se encuentra codificada directamente en JavaScript, el proceso de manipulación de los datos JSON resulta mucho más rápido que el tratamiento de documentos XML mediante DOM.

3.2.2 Estructura de datos JSON

Son dos las estructuras de datos que podemos utilizar para codificar la información en JSON:

- **Objetos**
- **Arrays**

3.2.2.1 OBJETOS JSON

Los objetos JSON almacenan la información como una cadena de caracteres formada por parejas nombre-valor, de modo que al reconstruir el objeto

JavaScript a partir del objeto JSON cada pareja estará representada por una propiedad con su correspondiente valor.

La sintaxis para definir un objeto JSON es la siguiente:

{nombre:valor, nombre:valor, nombre:valor,....}

donde *nombre* representa el nombre del dato o propiedad y *valor* su valor correspondiente, que puede ser cualquier tipo válido JavaScript: texto, numérico, lógico, objeto o array.

Por ejemplo, el siguiente objeto JSON contiene la definición de los datos de una persona:

```
{nombre:"Juan", dni:34345, edad:35}
```

Esta sintaxis para la definición de objetos JSON es la utilizada en la creación de literales de objetos JavaScript, así el objeto JSON anterior se definiría de la siguiente forma como literal de objeto JavaScript:

```
var objPersona;
objPersona = {
       nombre:"Juan", dni:34345, edad:35
}
```

De esta forma, si quisiéramos, por ejemplo, mostrar en un cuadro de diálogo la edad de la persona tendríamos que escribir la instrucción:

```
alert(objPersona.edad);
```

Los objetos JSON también admiten la definición de una función como valor en una pareja nombre-valor, lo que equivale a crear un método en el objeto JavaScript equivalente:

```
var objPunto;
objPunto = {
             x:10, y:40, dibujar: function(){
                          alert (this.x + "," +this.y);
                          }
}
```

Por lo que al ejecutar la siguiente instrucción:

```
objPunto.dibujar();
```

se mostraría un cuadro de diálogo con las coordenadas del objeto punto.

Obsérvese en el ejemplo anterior el uso de *this* para acceder desde la función a las propiedades del propio objeto.

3.2.2.2 ARRAYS

Además de los objetos, el otro tipo de estructura para representación de datos utilizada por JSON es el array. Un array JSON no es más que una sucesión de valores cuya sintaxis es:

[valor1, valor2, valor3,...]

pudiendo ser cada uno de estos valores cualquier tipo válido JavaScript, incluso de distinto tipo cada uno:

```
["Juan", "Alcalá", 25, true]
```

Los arrays JSON se definen utilizando la sintaxis de literales de array de JavaScript, así el array anterior equivale en código JavaScript a:

```
var datos = ["Juan", "Alcalá", 25, true];
```

Pudiendo acceder individualmente a cada dato a través de un índice:

```
alert(datos[0]);
```

La instrucción anterior mostraría un cuadro de diálogo con el valor “Juan”.

Es posible combinar ambas estructuras, objetos y arrays, para representar datos en JSON. En este sentido, es posible crear objetos JSON en los que el valor de alguna de sus propiedades sea un array de datos o tener un array en el que alguno de sus valores sea un objeto.

El siguiente ejemplo representa un objeto JSON en el que la propiedad *telefonos* es un array de números:

```
{nombre:"María", telefonos:[99678345, 6543434]}
```

3.2.3 Interpretación de JSON en cliente

Un objeto JSON es generado desde una aplicación del servidor y enviado al cliente como una cadena de caracteres en la respuesta, por tanto, para ser

recuperado desde el script de cliente deberá recurrirse a la propiedad *responseText* del objeto XMLHttpRequest.

Para que la cadena de texto contenida en esta propiedad sea interpretada como un objeto JavaScript, será necesario recurrir a la función *eval()* de JavaScript. Esta función evalúa como una expresión JavaScript el contenido de la cadena de texto que se le pasa como parámetro. Así, si queremos manipular en la página cliente el objeto recibido en la respuesta, deberíamos realizar algo similar a lo siguiente:

```
var obJson;
obJson = eval("("+xhr.responseText+")");
```

Obsérvense los paréntesis adicionales que se han añadido a ambos extremos de la cadena dentro del argumento de la función. El motivo es que, al estar delimitado el objeto mediante llaves, *eval()* interpretará que se trata de un bloque de instrucciones JavaScript e intentará ejecutarlas. Para que esto no suceda y se limite a evaluar la expresión, es necesario encerrar todo el texto entre paréntesis.

Una vez interpretado el objeto JSON, podemos utilizar la expresión:

obJson.propiedad

para acceder a las distintas propiedades del mismo.

3.2.4 Ejemplo de utilización

Como ejemplo de utilización de JSON, vamos a desarrollar la aplicación de ejemplo presentada en el capítulo 1, consistente en mostrar en una página Web una tabla con los datos asociados al libro, cuyo título se selecciona en una lista.

En este caso, los datos del libro que se enviarán en la respuesta desde el servidor al cliente vendrán encapsulados dentro de un objeto JSON. El siguiente listado corresponde a la nueva implementación del servlet “comentariolibro”:

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

public class Comentariolibro extends HttpServlet {
    protected void service(HttpServletRequest request,
```

```
            HttpServletResponse response) throws
                ServletException,IOException {
        response.setContentType("text/plain;charset=UTF-8");
        PrintWriter out = response.getWriter();
        //comentarios
        String [] comentarios=
        {"Requiere conocimientos básicos de programación
                                orientada a objetos",
        "Puede construir fácilmente aplicaciones
                                para la Web",
        "Aprenderá rápidamente los principales trucos
                                de Ajax",
        "Introduce las principales tecnologías
                                de la plataforma"};
        //precios
        String [] precios={"23.5","31.4","32.0","27.5"};
        int seleccionado=Integer.
                    parseInt(request.getParameter("tit"));
        //formación del objeto JSON de respuesta
        //con los datos del libro
        String textoJSON="{comentario:'";
        textoJSON+=comentarios[seleccionado]+"',";
        textoJSON+="precio:'"+precios[seleccionado]+"'}";
        out.println(textoJSON);
        out.close();
    }
}
```

En cuanto a las modificaciones a realizar en la página cliente, tan sólo se debería reescribir la función encargada de manipular la respuesta, así el nuevo código de la función *procesaDatos()* quedaría como se indica a continuación:

```
function procesaDatos(){
    if(xhr.readyState==4){
        //recuperación del objeto JSON recibido en la
        //respuesta
        var obj=eval("("+xhr.responseText+")");
        var textoHTML="<table border='1'>";
        textoHTML+="<tr>";
        textoHTML+="<th>Comentario</th>";
        textoHTML+="<th>Precio</th>";
        textoHTML+="</tr><tr>";
```

```
            textoHTML+="<td>"+obj.comentario+"</td>";
            textoHTML+="<td>"+obj.precio+"</td>";
            textoHTML+="</tr></table>";
            document.getElementById("info").
                 innerHTML=textoHTML;
        }
    }
```

Como se puede comprobar en este ejemplo, no solamente se transmite menos cantidad de información desde el servlet al cliente al no utilizar etiquetas de marcado, sino que además el acceso a los datos del libro resulta mucho más sencillo si utilizamos un objeto JSON que si vinieran codificados en un documento XML.

3.2.5 La librería JSON

El hecho de que la función *eval()* pueda interpretar cualquier código JavaScript que se le pase como parámetro supone un riesgo de seguridad importante cuando se trata de evaluar una cadena de caracteres recibida desde la Web, puesto que podría darse el caso de que esta cadena fuera manipulada de manera mal intencionada, incluyéndola algún tipo de código que pudiera tener efectos nocivos en el cliente.

A fin de evitar este problema, es posible hacer uso de un analizador JSON de libre distribución para evaluar el contenido de la cadena de respuesta AJAX, en vez de utilizar la función *eval()*.

En la página www.json.org podemos encontrar diversas herramientas software para trabajar con JSON, tanto en cliente como en servidor. Para el caso que nos ocupa, existe una librería de código JavaScript que se distribuye en un archivo llamado json.js, que proporciona algunos métodos para convertir cadenas JSON en objetos JavaScript y viceversa.

Para poder utilizar esta librería deberíamos agregar la siguiente referencia al archivo json.js en la cabecera de la página:

```
<script src="http://www.json.org/json.js"></script>
```

A continuación, se describen los métodos más interesantes de esta librería:

- **parseJSON()**. Se trata de un método que puede ser aplicado sobre un objeto de tipo String, realizando el análisis de la cadena de caracteres y su posterior conversión en objeto JSON. Así pues, si en

el ejercicio anterior queremos hacer uso de este método para obtener el objeto JSON de la cadena de respuesta, en vez de la instrucción:

```
var obj=eval("("+xhr.responseText+")");
```

deberíamos utilizar:

```
var obj=xhr.responseText.parseJSON();
```

- **toJSONString()**. Realiza la operación contraria a *parseJSON()*, es decir, dado un determinado objeto JavaScript formado por una serie de propiedades con sus valores correspondientes, este método devuelve la cadena JSON equivalente. Por ejemplo, dado el siguiente objeto:

```
var objPersona=new Object();
objPersona.nombre="Maria";
objPersona.edad=30;
objPersona.telefonos=new Array(34444,98458);
```

La siguiente instrucción:

```
var cadJson=objPersona.toJSONString();
```

almacenaría en la variable "cadJson" la cadena:

```
{nombre: "Maria", edad:30,
 telefonos:[34444,98458]}
```

PRÁCTICA 3.1. CATÁLOGO DE LIBROS

Descripción

En esta práctica se trata de desarrollar una aplicación donde el usuario pueda consultar un catálogo de libros a través de una página Web. El catálogo estará dividido en temas, pudiéndose elegir a través de una lista desplegable el tema cuyos libros quiere visualizar (figura 17).

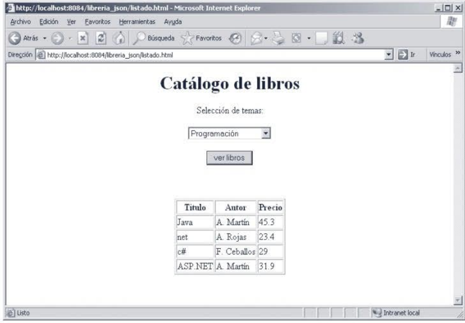


Fig. 17. *Página del catálogo de libros*

Desarrollo

Para el almacenamiento de los datos del catálogo, utilizaremos una base de datos con dos tablas, una para los temas del catálogo y otra para los datos de los libros. En la figura 18 se muestra la estructura de dichas tablas.

Temas

Campo	Tipo
IdTema	numérico entero
Tema	cadena

Libros

Campo	Tipo
ISBN	cadena
IdTema	numérico entero
Titulo	cadena
Autor	cadena
Precio	numérico decimal

Fig. 18. *Base de datos biblios*

Desde el punto de vista del servidor, la aplicación contará con dos servlets, uno de ellos, llamado "temas", generará un array de objetos JSON con los datos de todos los temas almacenados en la tabla, donde cada objeto tendrá dos propiedades: el identificador del tema y el nombre. La estructura de este array será la siguiente:

[{id:1, tema:"Programación"}, ...]

El otro servlet, llamado "libros", generará otro array de objetos JSON con los datos de los libros correspondientes al tema seleccionado. En este caso cada objeto del array contendrá tres propiedades: *titulo*, *autor* y *precio*:

[{titulo:"VB.NET", autor:"A.Martín", precio:23}, ...]

En cuanto al código cliente, en el evento *onload* de la página se procederá a cargar los temas del catálogo en la lista de selección, lanzando una petición en modo asíncrono del servlet temas. Una vez seleccionado un tema de la lista, a través del evento *onclick* del botón "ver libros" se produce la petición del servlet "libros", pasándole el identificador del tema seleccionado en la lista.

En esta práctica haremos uso de la clase ObjetoAJAX creada en el apartado anterior para implementar la funcionalidad cliente.

Código

A continuación, se muestra el código de los distintos componentes y páginas de la aplicación.

Servlet temas

```
package servlets;

import java.io.*;
import java.net.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Temas extends HttpServlet {
    protected void service(HttpServletRequest request,
                           HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        Connection cn=null;
        Statement st=null;
        ResultSet rs=null;
        try{
            //se utiliza un driver de tipo puente JDBC-ODBC
            //para acceder a la base de datos, siendo biblios
            //el DSN asociado a ésta
```

```
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            cn=DriverManager.
                          getConnection("jdbc:odbc:biblios");
            st=cn.createStatement();
            rs=st.executeQuery("select * from temas");
            StringBuilder objSecciones;
            //crea un array de objetos JSON con cada
            //tema de la tabla
            objSecciones=new StringBuilder("[");
            while(rs.next()){
                objSecciones.append("{id:");
                objSecciones.append(rs.getString("idTema"));
                objSecciones.append(",nombre:'");
                objSecciones.append(rs.getString("tema"));
                objSecciones.append("'},");
            }
            //sustituye la última coma por el carácter de
            //cierre del array
            objSecciones.replace(objSecciones.length()-1,
                                 objSecciones.length(),"]");
            out.println(objSecciones.toString());
        }
        catch(Exception e){e.printStackTrace();}
        out.close();
    }
}
```

Servlet libros

```
package servlets;

import java.io.*;
import java.net.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Libros extends HttpServlet {
   protected void service(HttpServletRequest request,
                          HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/plain;charset=UTF-8");
```

```
            PrintWriter out = response.getWriter();
            String tema=request.getParameter("tema");
            Connection cn=null;
            Statement st=null;
            ResultSet rs=null;
            try{
                //se utiliza un driver de tipo puente JDBC-ODBC
                //para acceder a la base de datos, siendo biblios
                //el DSN asociado a ésta
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                cn=DriverManager.
                       getConnection("jdbc:odbc:biblios");
                st=cn.createStatement();
                //busca los libros asociados al tema elegido
                String tsql="select * from libros ";
                tsql+="where idTema="+tema;
                rs=st.executeQuery(tsql);
                StringBuilder objLibros;
                //crea un array de objetos JSON con cada
                //libro de la tabla
                objLibros=new StringBuilder("[");
                while(rs.next()){
                    objLibros.append("{titulo:'");
                    objLibros.append(rs.getString("titulo"));
                    objLibros.append("', autor:'");
                    objLibros.append(rs.getString("autor"));
                    objLibros.append("', precio:");
                    objLibros.append(rs.getString("precio"));
                    objLibros.append("},");
                }
                //sustituye la última coma por el carácter de
                //cierre del array
                objLibros.replace(objLibros.length()-1,
                                     objLibros.length(),"]");
                out.println(objLibros.toString());
            }
            catch(Exception e){e.printStackTrace();}
            out.close();
        }
    }
```

Página listado.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                           Transitional//EN">

<html>
  <head>
     <title></title>
     <script src="utilidades.js"
              language="javascript"></script>
     <script language="javascript">
         var obj;
         function cargarTemas(){
            obj=new ObjetoAJAX();
            //invoca al servlet que devuelve el
             //array de temas
            obj.enviar("temas","GET","procesaTemas", null);
         }
         function cargarLibros(){
            obj=new ObjetoAJAX();
            var oForm=document.forms[0];
            //invoca al servlet que devuelve el array de
            //libros, pasando los datos del tema seleccionado
           //en la petición
            obj.enviar("libros","post",
                          "procesaLibros", oForm);
         }
         function procesaTemas(){
             var temas=eval("("+obj.respuestaTexto()+")");
             //crea la opción de selección inicial
             var opcionInicio=
                    document.createElement("option");
             var textoOpcion=document.createTextNode(
                                        "-Seleccione Tema-");
             opcionInicio.appendChild(textoOpcion);
             var lista=document.getElementById("temas");
             lista.appendChild(opcionInicio);
             //genera el resto de opciones de la lista a
             //partir de los datos recibidos en la respuesta
             for(var elm = 0;elm < temas.length;elm++){
                    var item=
```

```
                        document.createElement("option");
                    item.setAttribute("value",temas[elm].id);
                    var texto=document.
                        createTextNode(temas[elm].nombre);
                    item.appendChild(texto);
                    document.getElementById("temas").
                        appendChild(item);
                }
            }
            function procesaLibros(){
                //recupera el objeto JSON
                var libros=eval("("+obj.respuestaTexto()+")");
                var tabla="<table border='1'>";
                tabla+="<tr><th>Titulo</th><th>Autor</th>";
                tabla+="<th>Precio</th></tr>";
                //genera el contenido de la tabla a partir
                //de los datos recibidos en la respuesta
                for(var elm = 0;elm < libros.length;elm++){
                   tabla+="<tr>";
                   tabla+="<td>"+libros[elm].titulo+"</td>";
                   tabla+="<td>"+libros[elm].autor+"</td>";
                   tabla+="<td>"+libros[elm].precio+"</td>";
                   tabla+="</tr>";
                }
                tabla+="</table>";
                document.
                    getElementById("listado").innerHTML=tabla;
            }
        </script>
    </head>
    <body onload="cargarTemas();">
        <center>
          <h1>Catálogo de libros</h1>
          Selección de temas:
          <form action="libros" method="post">
              <select id="temas" name="tema">
              </select>
              <br/><br/>
              <input type="button" onclick="cargarLibros();"
                     value="ver libros"/>
          </form>
```

```
            <br/>
            <br/>
            <div id="listado">
            </div>
            <h2>
          </center>
      </body>
    </html>
```

3.3 DIRECT WEB REMOTING

Direct Web Remoting (DWR) es un kit de desarrollo para la construcción de aplicaciones AJAX en Java. Se basa en la invocación remota a métodos de objetos del servidor desde una página cliente, utilizando como proxy un objeto JavaScript.

Estos **objetos JavaScript son creados dinámicamente en tiempo de ejecución** por DWR a partir de la información de asociación suministrada en un archivo de configuración de la aplicación, empleándose internamente el objeto XMLHttpRequest en modo asíncrono para realizar las llamadas a los métodos de los objetos del servidor.

Resulta evidente la enorme potencia que tiene esta herramienta, pues no solamente se proporcionan al programador una serie de objetos en cliente con toda la funcionalidad del servidor sin escribir una sola línea de código, sino que además estos objetos aprovechan toda la potencia de AJAX haciendo totalmente transparente el uso de esta tecnología al programador.

Seguidamente, describiremos el funcionamiento de una aplicación DWR, así como los pasos a seguir para poder utilizar DWR en un desarrollo, analizando un ejemplo práctico de uso.

3.3.1 Componentes DWR

Los principales elementos que componen el conjunto de utilidades DWR son:

- **El servlet DWRServlet**. Formará parte del código de servidor de la aplicación. Se trata del componente principal de la misma, ya que a él se dirigirán las peticiones del cliente cada vez que se quiera invocar a algún método del objeto Java del servidor.

- **Archivo de configuración dwr.xml**. Forma parte también de la aplicación Web del servidor. Su misión es registrar las equivalencias entre los objetos Java de servidor y los JavaScript de cliente, definiendo además distintos parámetros de configuración adicionales que analizaremos más adelante.

- **Librería cliente**. Las aplicaciones DWR utilizan dos archivos JavaScript que necesitarán ser referenciados desde la página cliente. Ambos archivos son generados dinámicamente por DWR. Se trata de:

  - **util.js**. Incorpora funciones de utilidades de cliente.

  - **engine.js**. Constituye el motor de AJAX que encapsula la funcionalidad del objeto XMLHttpRequest.

- **Objeto cliente**. Se trata de una librería .js, también generada dinámicamente por DWR y que incorpora el objeto JavaScript cliente que se utilizará de proxy para invocar a los métodos del objeto servidor.

3.3.2 El kit de desarrollo DWR

Para implementar una aplicación DWR es necesario descargar del sitio Web *http://getahead.ltd.uk/dwr* un archivo llamado dwr.jar que proporciona el conjunto de utilidades mencionadas anteriormente. Como después indicaremos, este archivo deberá ser alojado en un determinado directorio de la aplicación del servidor.

3.3.3 Funcionamiento de una aplicación DWR

Según hemos indicado anteriormente, la creación de una aplicación DWR requiere la utilización de determinados componentes, tanto en cliente como en servidor, para poder funcionar.

En la figura 19 podemos ver el esquema de funcionamiento de una aplicación basada en DWR, donde se puede apreciar el rol de cada uno de estos componentes dentro de la arquitectura.

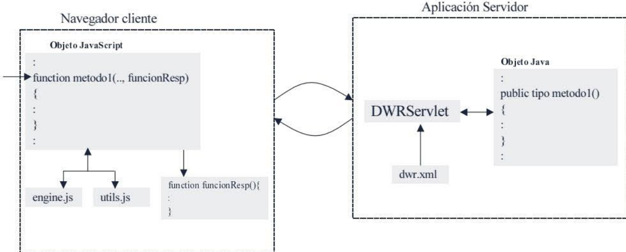


Fig. 19. *Aplicación DWR*

A continuación describiremos los procesos que tienen lugar durante la ejecución de una aplicación de estas características:

1. **Llamada al método del objeto cliente**. Desde algún script de la página cliente se invoca a alguno de los métodos del objeto JavaScript asociado con el objeto de servidor que se quiere ejecutar.

2. **Lanzamiento de la petición AJAX**. La llamada anterior es capturada por el motor de AJAX engine.js, el cual realiza una petición en modo asíncrono al servlet DWRServlet, pasándole en la llamada, además de los parámetros utilizados por el propio método, otra serie de datos que permiten al servlet identificar el método y objeto cliente invocado.

3. **Llamada al método del objeto servidor**. A partir de los datos recibidos en la petición y de la información registrada en el fichero de configuración dwr.xml, el servlet crea una instancia de la clase del servidor asociada al objeto cliente y ejecuta la llamada al método correspondiente.

4. **Envío de la respuesta al cliente**. Una vez que se completa la ejecución del método, los datos devueltos por éste son enviados como respuesta al cliente.

5. **Tratamiento de la respuesta**. Dado que la petición cliente se ha realizado utilizando AJAX en modo asíncrono, la respuesta será recibida en una función de retrollamada definida previamente en la página cliente.

3.3.4 Desarrollo de una aplicación DWR

Seguidamente, vamos a explicar el funcionamiento del proceso a seguir para desarrollar una aplicación DWR. Para ello, utilizaremos un ejemplo consistente en rellenar una lista de selección de una página XHTML con una serie de datos (figura 20), datos que serán obtenidos invocando a un método de un objeto de servidor utilizando DWR.

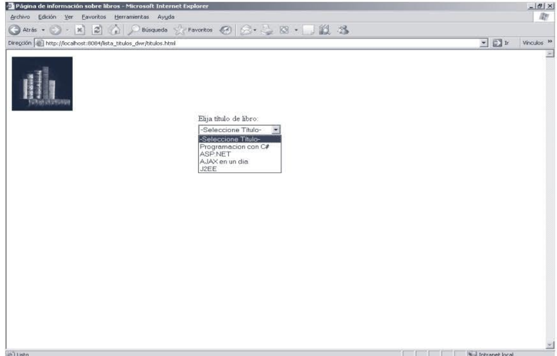


Fig. 20. *Página con la lista de títulos obtenida del servidor*

Como se aprecia en la figura anterior, el ejemplo utilizado se corresponde con la lista de selección de títulos de libros empleada en prácticas y ejemplos precedentes.

3.3.4.1 CONFIGURACIÓN DEL ENTORNO

Dado que la capa del servidor de una aplicación DWR incluye como elemento principal un servlet, lo primero que habrá que hacer es crear una aplicación Web Java en el servidor, para lo que podemos utilizar cualquier IDE JAVA EE existente, por ejemplo, NetBeans.

Una vez creada la estructura de la aplicación, copiamos el archivo de utilidades dwr.jar que descargamos de la URL indicada anteriormente en el directorio *raiz_aplicacion/WEB-INF/lib* de la aplicación. Este fichero contiene tanto el servlet DWRServlet de la aplicación como los elementos necesarios para generar dinámicamente los archivos .js cliente.

Lo siguiente que debemos hacer es registrar el servlet en el archivo de configuración web.xml de la aplicación. Dicho archivo debe estar situado en el directorio r*aiz_aplicacion/WEB-INF*. Si estamos utilizando un entorno de desarrollo, seguramente existirá alguna opción que nos permita editar este archivo dentro del propio entorno, si no podemos emplear cualquier editor de textos como, por ejemplo, el notepad. Una vez editado el archivo, añadimos las etiquetas <servlet> y <servlet-mapping> para proceder al registro del servlet, tal y como se muestra en el siguiente listado:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
       xmlns="http://java.sun.com/xml/ns/Java EE"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/Java EE
http://java.sun.com/xml/ns/Java EE/web-app_2_4.xsd">
    <servlet>
        <servlet-name>dwr-invoker</servlet-name>
        <servlet-class>uk.ltd.getahead.dwr.DWRServlet
         </servlet-class>
        <init-param>
            <param-name>debug</param-name>
            <param-value>true</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>dwr-invoker</servlet-name>
        <url-pattern>/dwr/*</url-pattern>
    </servlet-mapping>
    :
</web-app>
```

3.3.4.2 IMPLEMENTACIÓN DE LA CLASE DEL SERVIDOR

Hasta aquí, tenemos ya todo preparado para comenzar a escribir el código de la aplicación DWR. Empezaremos por la clase que contiene los métodos que va a exponer el objeto del servidor.

En nuestro ejemplo, la clase se llamará Datos y dispondrá únicamente del método *getTitulos()*, el cual devolverá un array de cadenas de caracteres con los títulos de los libros. Éste es el código completo de la clase Datos:

```
package modelo;

public class Datos {
    public Datos() {
    }
    public String[] getTitulos(){
        String [] titulos={"Programacion con C#",
                           "ASP.NET",
                           "AJAX en un dia",
                           "JAVA EE"};
        return titulos;
    }
}
```

Dado que esta clase debe formar parte de la aplicación Web del servidor, el archivo .class con el código compilado deberá estar incluido el algún paquete dentro del directorio *raiz_aplicacion/WEB-INF/classes*. En nuestro ejemplo este paquete se llama “modelo” y “lista_titulos_dwr” es el nombre del directorio raíz de la aplicación Web.

Lógicamente, todo este proceso que acabamos de explicar para implementar una clase Java en el servidor no será necesario realizarlo si, en vez de hacer uso de una clase personalizada, queremos utilizar en el código JavaScript cliente alguna de las clases incluidas en las librerías de Java.

3.3.4.3 CREACIÓN DEL ARCHIVO DWR.XML

Además del archivo de configuración web.xml propio de cualquier aplicación Web JAVA EE, en el caso de aplicaciones DWR es necesario incluir un segundo archivo de estas características en el mismo directorio que el anterior, llamado **dwr.xml**, en el que se establezca la relación entre la clase de servidor y el objeto cliente que se utilizará para hacer las llamadas a los métodos de los objetos de ésta. En nuestro ejemplo el contenido de este archivo será:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web
      Remoting 1.0//EN"
      "http://www.getahead.ltd.uk/dwr/dwr10.dtd">
<dwr>
  <allow>
    <create creator="new" javascript="JDatos">
      <param name="class" value="modelo.Datos"/>
    </create>
```

```
  </allow>
</dwr>
```

Mediante el atributo *javascript* del elemento <create> se indica el nombre del objeto cliente, utilizándose el atributo *creator* para definir la acción a realizar en el servidor. A través del subelemento <param> se especifican los datos de la clase del servidor.

Todo lo anterior implica que **cada vez que se invoque a un método del objeto JavaScript JDatos, se creará una instancia de la clase modelo.Datos en el servidor y se ejecutará el método con el mismo nombre**. Más adelante veremos algunas otras opciones de configuración más que pueden ser incluidas en este archivo.

3.3.4.4 CREACIÓN DE LA PÁGINA CLIENTE

Para poder hacer uso de DWR desde la página cliente, debemos incluir las siguientes referencias a los archivos .js en la cabecera del archivo titulos.html:

```
<head>
    <title>Página de información sobre libros</title>
    <script type="text/javascript"
            src="/lista_titulos_dwr/dwr/engine.js">
    </script>
    <script type="text/javascript"
            src="/lista_titulos_dwr/dwr/util.js">
    </script>
    <script type="text/javascript"
            src="/lista_titulos_dwr/dwr/interface/JDatos.js">
    </script>
    :
```

Los tres serán generados dinámicamente por DWR dentro de los directorios indicados, siendo *lista_titulos_dwr* el directorio raíz de la aplicación Web. Como se puede apreciar, los archivos util.js y engine.js son creados dentro del subdirectorio *dwr* mientras que el objeto proxy se genera en dwr/interface.

En cuanto a la vista de la página, el siguiente listado muestra el contenido XHTML de la misma:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                                   Transitional//EN">
<html>
```

```
  <head>
       :
  </head>
  <body onload="buscarTitulos();">
  <img src="libros.jpg"/>
     <center>
      <table width="30%">
          <tr><td>Elija título de libro:</td></tr>
          <tr>
              <td>
                  <select id="titulo">
                  </select>
              </td>
          </tr>

      </table>
     </center>
  </body>
</html>
```

Como podemos comprobar, en el evento *onload* de la página se hace una llamada a la función *buscarTitulos()*. Ésta será la encargada de recuperar el array de títulos a través del objeto JDatos y generar a partir de él la lista HTML. He aquí el código de esta función:

```
function buscarTitulos(){
      //invoca al método getTitulos del objeto de servidor
      //a través del objeto cliente
      JDatos.getTitulos(generaLista);
}
```

Tal y como se indica en la última instrucción de la función, es necesario proporcionar como parámetro de llamada al método del objeto cliente la función de retrollamada donde se procesarán los resultados, en nuestro ejemplo esta función se llama *generaLista()*. Si hubiera que pasar parámetros al método del servidor para su ejecución, éstos deberían ser suministrados también a través del método cliente delante de la función de retrollamada:

objeto.metodo(argumento1, argumento2, .., funcion_retrollamada);

En caso de que el método de servidor devuelva algún resultado, la función de retrollamada deberá declarar un parámetro en el que se recoja dicho valor. La función *generaLista()* de nuestro ejemplo quedará, pues, de la siguiente forma:

```
function generaLista(titulos){
     //crea la primera opción de la lista
     //y la añade a ésta
     var opcionInicio=document.createElement("option");
     var textoOpcion=document.createTextNode(
                              "-Seleccione Título-");
     opcionInicio.appendChild(textoOpcion);
     var lista=document.getElementById("titulo");
     lista.appendChild(opcionInicio);
     //genera el resto de opciones de la lista a partir
     //de los datos recibidos en la respuesta
     for(var elm = 0;elm < titulos.length ;elm++){
           var item=document.createElement("option");
           var texto=document.createTextNode(titulos[elm]);
           item.appendChild(texto);
           document.getElementById("titulo").
                    appendChild(item);
     }
}
```

Como vemos, la variable *titulos* contiene el array de cadenas devuelto por el método *getTitulos()* del objeto Datos. Utilizando el DOM, la función *generaLista()* extrae la información de este array y genera los elementos <option> de la lista.

3.3.4.5 UTILIDADES DWR DE CLIENTE

Además del objeto proxy, las aplicaciones clientes DWR pueden hacer uso de otro objeto JavaScript incluido en el archivo util.js. Se trata del objeto DWRUtil y proporciona distintos métodos de utilidades que pueden resultar de gran ayuda al programador, puesto que muchos de ellos encapsulan gran parte de la funcionalidad DOM para manipular el contenido de la página. Entre estos métodos destacan:

- **getValue(identificador_elemento)**. Obtiene el valor del elemento cuyo identificador se le proporciona como parámetro.

- **setValue (identificador_elemento, cadena)**. Permite asignar al elemento cuyo identificador se indica en el primer parámetro el valor

indicado en el segundo, siendo éste una cadena de caracteres sin etiquetas de marcado.

Tanto este método como el anterior pueden ser utilizados tanto con controles HTML como con elementos de tipo <div> y <span>.

- **addOptions (identificador_lista, array)**. Se trata de un método muy potente; genera dinámicamente los elementos <option> dentro de la lista de selección cuyo identificador se pasa como primer parámetro, a partir del array de cadenas que se suministra como segundo parámetro.

  Con este formato de método, las cadenas de caracteres incluidas en cada una de las posiciones del array son utilizadas como texto delimitado por el elemento y como valor del atributo *value* de cada uno de ellos. Existe otra versión de *addOptions()* que permite generar una colección de elementos a partir de una array de objetos JSON, pudiendo especificar qué propiedad del objeto suministrará el valor del elemento y cuál el texto delimitado:

  addOptions(identif, array_objetos, prop_valor, prop_texto)

- **removeAllOptions (identificador_lista)**. Elimina todos los elementos <option> incluidos en la lista de selección cuyo identificador se especifica.

Aplicando a nuestro ejemplo el método *addOptions()*, que se acaba de explicar, podemos reducir considerablemente el código de la función *generaLista()*:

```
function generaLista(titulos){
    //crea la primera opción de la lista
    //y la añade a ésta
    var opcionInicio=document.createElement("option");
    var textoOpcion=document.createTextNode(
                                "-Seleccione Título-");
    opcionInicio.appendChild(textoOpcion);
    var lista=document.getElementById("titulo");
    lista.appendChild(opcionInicio);
    //genera el resto de opciones de la lista a partir
    //de los datos recibidos en la respuesta y haciendo
    //uso del objeto de utilidades
```

```
        DWRUtil.addOptions("titulo",titulos);
}
```

Suponiendo que el método *getTitulos()* del objeto de la clase Datos devolviera un array de objetos JSON, en vez de un array de cadenas, donde cada uno de éstos tuviera dos propiedades: *isbn* y *titulo*, para que el contenido de la propiedad *isbn* sirviera como valor del elemento <option> y *titulo* como texto contenido en el mismo, deberíamos invocar al método addOptions de la siguiente forma:

```
DWRUtil.addOptions("titulo",titulos, "isbn", "titulo");
```

A partir de aquí, ya tenemos la aplicación totalmente funcional; cada vez que se cargue la página titulos.html en el navegador, se producirá una llamada al método *getTitulos()* del objeto JDatos que hará que se lance una petición AJAX al servlet DWRServlet para que cree una instancia de la clase Datos y se invoque al método *getTitulos()* de este objeto, devolviéndose el resultado a la función JavaScript *generaLista()*.

3.3.5 Opciones de configuración de dwr.xml

En el ejemplo que acabamos de analizar se ha empleado únicamente el elemento <create> para el caso de que queramos asociar una instancia de una clase del servidor con el objeto cliente. Vamos a ver qué otras opciones más de configuración nos proporciona este archivo.

3.3.5.1 INSTANCIAS DE OBJETOS SIN CONSTRUCTORES PÚBLICOS

La configuración del elemento <create> presentada en el ejemplo anterior es una configuración básica que sólo nos permite crear instancias de clases que dispongan de constructores públicos.

En Java existen algunas clases, como es el caso de java.util.Calendar, cuyas instancias no pueden ser creadas mediante el operador *new*, sino que debe hacerse a través del algún método estático de la propia clase, en este caso *getInstance()*.

Cuando se trata de crear una instancia de una clase con estas características, el elemento <create> deberá utilizarse como se indica a continuación:

```
<create cretator="script" javascript="JCalendar">
    <param name="language" value="beanshell"/>
    <param name="script">
        import java.util.Calendar;
        return Calendar.getInstance();
    </param>
</create>
```

Como se puede observar, la creación de la instancia se debe realizar ejecutando un script, es por ello que el atributo *creator* de <create> tiene asignado el valor "script" y no "new".

Este script se encargará de realizar la llamada al método que crea la instancia, debiendo ser incluido dentro de un elemento <param>. Un anterior elemento <param> permite indicar el lenguaje de script que será utilizado, siendo BeanShell (Bean Scripted Framework) el empleado en este ejemplo.

3.3.5.2 INCLUSIÓN Y EXCLUSIÓN DE MÉTODOS

Por defecto, todos los métodos del objeto de servidor podrán ser invocados a través del objeto cliente. Si por motivos de seguridad queremos que alguno de estos métodos no pueda ser utilizado desde el cliente, debemos excluirlo a través del subelemento <exclude> de <create>. Por ejemplo, si no queremos que el cliente pueda invocar al método *clear()* de la clase Calendar utilizada como ejemplo anteriormente, debería configurarse <create> de la siguiente manera:

```
<create cretator="script" javascript="JCalendar">
    <param name="language" value="beanshell"/>
    <param name="script">
        import java.util.Calendar;
        return Calendar.getInstance();
    </param>
    <exclude method="clear"/>
</create>
```

Por otro lado, si lo que queremos es que no se puedan utilizar más que ciertos métodos de la clase, tenemos que habilitarlos a través del elemento <include>. En el siguiente ejemplo la aplicación cliente podrá utilizar únicamente los métodos *getInstance()* y *get()* de Calendar:

```
<create cretator="script" javascript="JCalendar">
    <param name="language" value="beanshell"/>
    <param name="script">
```

```
                import java.util.Calendar;
                return Calendar.getInstance();
          </param>
          <include method="getInstance"/>
          <include method="get"/>
    </create>
```

En el momento en que se hace uso de <include> dentro de <create>, se prohíbe el acceso a todos los métodos de la clase salvo a los especificados mediante ese elemento.

3.3.5.3 CONVERTIDORES

Cuando la llamada a un método de un objeto Java devuelve un resultado, DWR convierte automáticamente este dato a la representación JavaScript correspondiente. Sin embargo, cuando el tipo de devolución es un JavaBean, por motivos de seguridad se desactiva la conversión de este tipo de datos, evitando así la posibilidad de que pudiera ejecutarse algún código malicioso en el cliente.

Si queremos habilitar la conversión de un determinado tipo de JavaBean, es necesario habilitarlo explícitamente en el fichero dwr.xml a través del elemento <convert>. Por ejemplo, suponiendo que tenemos una clase en el servidor llamada Cesta, en la cual tenemos un método llamado *getItem()* que nos devuelve los datos de un determinado libro, encapsulados en un objeto JavaBean llamado "Libro", deberíamos configurar el archivo de configuración dwr.xml de la siguiente manera:

```
<dwr>
  <allow>
    <create creator="new" javascript="JCesta">
      <param name="class" value="modelo.Cesta"/>
    </create>
    <convert converter="bean" match="modelo.Libro"/>
  </allow>
</dwr>
```

PRÁCTICA 3.2. IMPLEMENTACIÓN DE UN CHAT

Descripción

En esta práctica vamos a realizar la implementación de un sencillo chat que permita enviar y recibir mensajes en tiempo real entre todos los usuarios conectados a la aplicación.

La vista de la aplicación estará formada por una página XHTML a través de la que se podrán enviar mensajes, visualizando al mismo tiempo la lista completa de todos los mensajes emitidos, junto con el nombre o nick de usuario que lo ha enviado. El aspecto de esta página se muestra en la figura 21.

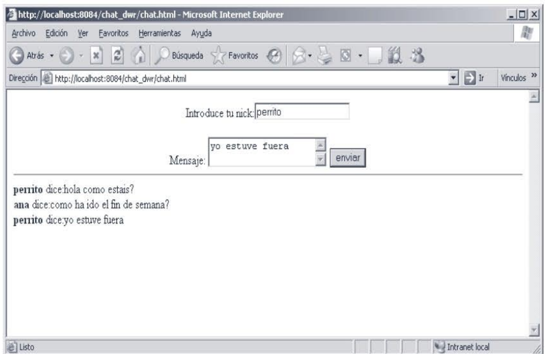


Fig. 21. *Aspecto de la página de la aplicación*

El funcionamiento de la aplicación es bastante sencillo: cuando un usuario desee enviar un mensaje al resto de miembros del chat, escribirá su nick y contenido del mensaje en los campos de texto correspondientes, pulsando a continuación el botón “enviar”. Cada vez que un usuario envíe un mensaje, se actualizará en su página la lista de mensajes enviados, por otro lado, mientras no realice el envío de mensajes esta lista se actualizará cada cinco segundos.

Desarrollo

El desarrollo de esta aplicación está basado en DWR. Para ello, crearemos una clase en el servidor llamada GestionMensajes que se encargue de almacenar en una lista todos los mensajes que se van enviando (método *agragarMensaje()*) y que, al mismo tiempo, proporcione algún método (*obtenerMensajes()*) para poder recuperar la lista de mensajes.

Los datos de un mensaje, nick y texto, se encapsularán en una clase de tipo JavaBean llamada Mensaje.

En lo que respecta al código de la capa cliente, se utilizará un objeto proxy para acceder a los métodos de GestionMensajes. El envío del mensaje se producirá

en el evento onclick del botón "enviar", mientras que la carga de mensajes en la página se realizará llamando al método *obtenerMensajes()* desde un función JavaScript que, a través de un temporizador, se ejecutará automáticamente cada cinco segundos.

Listado

A continuación, se muestra el código de los distintos elementos que forman la aplicación

Clase Mensaje

```
package modelo;

public class Mensaje {
    private String nick;
    private String texto;
    public Mensaje() {
    }
    public Mensaje(String nick, String texto){
        this.setNick(nick);
        this.setTexto(texto);
    }
    public String getNick() {
        return nick;
    }
    public void setNick(String nick) {
        this.nick = nick;
    }
    public String getTexto() {
        return texto;
    }
    public void setTexto(String texto) {
        this.texto = texto;
    }
}
```

Clase GestionMensajes

```
package modelo;

import java.util.*;
public class GestionMensajes {
```

```
    //variable que almacena la lista de mensajes
    private static ArrayList mensajes=new ArrayList();

    public void agregarMensaje(String nick, String texto){
        //crea un objeto Mensaje y lo
        //almacena en la lista
        Mensaje m=new Mensaje(nick,texto);
        mensajes.add(m);
    }
    public List obtenerMensajes(){
        return mensajes;
    }
}
```

Fichero de configuración web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/Java EE"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/Java EE
       http://java.sun.com/xml/ns/Java EE/web-app_2_4.xsd"
       version="2.4">
   <servlet>
        <servlet-name>dwr-invoker</servlet-name>
        <servlet-class>uk.ltd.getahead.dwr.DWRServlet
       </servlet-class>
        <init-param>
            <param-name>debug</param-name>
            <param-value>true</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>dwr-invoker</servlet-name>
        <url-pattern>/dwr/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Fichero de configuración dwr.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web
              Remoting 1.0//EN"
```

```
            "http://www.getahead.ltd.uk/dwr/dwr10.dtd">

<dwr>
  <allow>
    <create creator="new" javascript="JMensajes">
      <param name="class" value="modelo.GestionMensajes"/>
    </create>
    <convert converter="bean" match="modelo.Mensaje"/>
  </allow>
</dwr>
```

Página cliente chat.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                                 Transitional//EN">

<html>
  <head>
    <script type="text/javascript"
            src="/chat_dwr/dwr/engine.js">
    </script>
    <script type="text/javascript"
            src="/chat_dwr/dwr/util.js">
    </script>
    <script type="text/javascript"
            src="/chat_dwr/dwr/interface/JMensajes.js">
    </script>
    <script language="javascript">
        var temp=null;
        function crearTemporizador(){
            //crea un temporizador que actualice
            //periódicamente la tabla de mensajes
            window.setInterval("actualizarLista()",5000);
        }
        function enviar(){
            //recupera los datos de los controles de texto
            var nick=DWRUtil.getValue("nick");
            var mensaje=DWRUtil.getValue("mensaje");
            //añade el mensaje y después actualiza la
            //tabla de mensajes
            JMensajes.agregarMensaje(nick,mensaje);
            actualizarLista();
```

```
            }
            function actualizarLista(){
                JMensajes.obtenerMensajes(procesar);
            }
            //función de retrollamada que será invocada
            //tras la ejecución de obtenerMensajes() y cuya
            //misión es la de actualizar la tabla de mensajes
            function procesar(mensajes){
                var contenido="";
                for(var i=0;i<mensajes.length;i++){
                    contenido+="<b>"+mensajes[i].nick+
                                            "</b> dice:";
                    contenido+=mensajes[i].texto+"<br/>";
                }
                document.getElementById("mensajes").
                        innerHTML=contenido;
            }
        </script>
      </head>
      <body onload="crearTemporizador();">
          <center>
              Introduce tu nick:<input type="text" id="nick"/>
                 <br/><br/>
              Mensaje:
              <textarea id="mensaje" value=""></textarea>
              <input type="button" value="enviar"
                     onclick="enviar();"/>
          </center>
          <hr/>
          <div id="mensajes">
          </div>
      </body>
    </html>
```

3.4 HERRAMIENTAS AJAX PARA CLIENTE

El framework DWR, que acabamos de analizar en el apartado anterior, proporciona utilidades para el desarrollo, tanto para la capa cliente como para el servidor, de una aplicación AJAX.

No obstante, existen multitud de utilidades de libre uso en la Web orientadas a facilitar exclusivamente el desarrollo de la capa cliente. Algunas de estas utilidades son muy potentes y fáciles de utilizar, por lo que su uso puede resultar muy interesante en muchas aplicaciones.

En este apartado vamos a centrarnos en el estudio de dos de las más populares de estas utilidades: *prototype* y *script.aculo.es*, además de analizar el que es con toda seguridad el framework más importante para desarrollo de clientes AJAX: jQuery.

3.4.1 Prototype

Prototype consta, básicamente, de dos grupos de utilidades:

- **Objetos para el envío de peticiones al servidor**. Incluye una serie de objetos que encapsulan la funcionalidad del objeto XMLHttpRequest y facilitan enormemente la interacción con el servidor, simplificando tanto el proceso de envío de peticiones como la recepción y tratamiento de respuestas.

- **Funciones de utilidad**. Prototype proporciona también un serie de funciones auxiliares que simplifican la realización de tareas habituales dentro de una aplicación cliente AJAX, como, por ejemplo, el acceso a la funcionalidad del DOM.

Todas estas utilidades se incluyen en el archivo **prototype.js** que se puede descargar de la Web http://www.prototypejs.org.

Una vez descargado el archivo, para poder hacer uso de los objetos y funciones incluidos en el mismo habrá que incluir una referencia a éste dentro del código XHTML de la página a través de la etiqueta <script>.

3.4.1.1 EL NÚCLEO DE OBJETOS PROTOTYPE

3.4.1.1.1 El objeto Ajax.Request

Al igual que el objeto personalizado ObjetoAJAX, que creamos en el primer apartado de este capítulo, Ajax.Request de prototype permite realizar peticiones AJAX al servidor de una forma sencilla, abstrayendo al programador de los detalles de configuración del objeto XMLHttpRequest.

La sintaxis para crear una instancia de Ajax.Request es la siguiente:

var obj = new Ajax.Request(url, objeto_json)

siendo el significado de los parámetros el siguiente:

- **url**. Cadena de caracteres con la URL del proceso de servidor que va a atender la solicitud.
- **objeto_json**. Representa un objeto JSON, siguiendo la sintaxis estudiada en apartados anteriores de este capítulo, con el que se define una serie de propiedades adicionales que permiten completar la solicitud AJAX. El número de estas propiedades es variable, siendo las más utilizadas:
  - **method**: método utilizado para realizar la solicitud ("GET" o "POST").
  - **onComplete**: función de retrollamada que será ejecutada cuando se haya completado la recepción de la respuesta. Esta función recibirá como parámetro un objeto Ajax.Response que permitirá acceder a la información de la respuesta. Dado que la función de retrollamada es invocada únicamente cuando la respuesta ya ha sido recibida, **el programador no tendrá que preocuparse por comprobar el estado de la petición**.
  - **parameters**: cadena de caracteres con los parámetros que se van a enviar al servidor en la solicitud, siguiendo el formato: parametro=valor¶metro=valor...

Tan **sólo será necesario crear el objeto AJAX.Request** para que todo el proceso de apertura, configuración y envío de la petición se realice de forma automática, dedicando así todos los esfuerzos de programación al tratamiento de la respuesta.

Para poder acceder a los datos de la respuesta, el objeto Ajax.Response recibido por la función de retrollamada expone las siguientes propiedades de XMLHttpRequest:

- **responseText**
- **responseXML**
- **status**

Además se incluyen algunas propiedades y métodos nuevos, entre los que cabe destacar:

- **getHeader(encabezado)**. Método que devuelve, a partir de su nombre, el valor de uno de los encabezados recibidos en la respuesta.
- **responseJSON.** Propiedad que contiene la respuesta en formato JSON. Solamente tendrá un valor válido si el content-type de la petición se ha establecido a application/json.

Como ejemplo de uso de este objeto, el siguiente listado corresponde a un script de cliente en el que se realiza una petición de tipo "POST" a un servlet cuya url es "validar", pasándole como parámetro los contenidos de los controles de texto "txtusuario" y "txtpassword". La respuesta generada por el servlet es mostrada directamente en la página Web dentro de un elemento con identificador "resultado":

```
function validar(){
     //forma la cadena con los parámetros de la petición
     var parametros="txtusuario="+
             document.getElementById("txtusuario").value
       +"&";
     parametros+="txtpassword="+
             document.getElementById("txtpassword").value;
     //forma la cadena del objeto JSON
     var objson="{method:'POST',
             parameters:'"+parametros+"',";
     objson+="onComplete:muestraResultado}";
     //evalúa la cadena para poder tratarlo como objeto
     objson=eval("("+objson+")");
     //crea el objeto Ajax.Request
     var obj = new Ajax.Request("validar",objson);
}
function muestraResultado(objeto){
     document.getElementById("resultado").
             innerHTML=objeto.responseText;
}
```

3.4.1.1.2 El objeto Ajax.Updater

Trabaja de forma similar a Ajax.Request, realizando, además, la actualización automática de una parte de la página con la información recibida en la respuesta.

Así pues, si lo que queremos es mostrar directamente la cadena de texto de respuesta en algún punto de la interfaz, utilizando Ajax.Updater el programador no tendrá que definir ninguna función de retrollamada para realizar esta tarea, encargándose el objeto automáticamente de ello.

La sintaxis para crear un objeto Ajax.Updater es la siguiente:

var obj = Ajax.Updater(identificador, url, objeto_json);

donde el parámetro *identificador* representa el valor del atributo *id* del elemento HTML donde se debe mostrar el contenido de la respuesta.

El siguiente listado representa el mismo caso del ejemplo anterior, utilizando como solución el objeto Ajax.Updater:

```
function validar(){
     //forma la cadena con los parámetros de la petición
     var parametros="txtusuario="+
             document.getElementById("txtusuario").value
      +"&";
     parametros+="txtpassword="+
             document.getElementById("txtpassword").value;
     //forma la cadena del objeto JSON
     var objson= eval("("+"{method:'POST',
             parameters:'"+parametros+"'}"+")");
     var obj = new
              Ajax.Updater("resultado","validar",objson);
}
```

En cuanto al objeto JSON que se suministra como tercer parámetro, como vemos en el listado anterior dispondrá de las mismas propiedades que en el caso de Ajax.Request, salvo *onComplete* que para Ajax.Updater no tiene utilidad.

Por el contrario, el objeto JSON de Ajax.Updater incorpora una nueva propiedad llamada *Insertion*, mediante la que se puede indicar el modo de inserción de los datos de respuesta en la página. Por defecto, el contenido del elemento indicado como primer parámetro es reemplazado por el texto de respuesta, aunque

se puede indicar otra opción a través de esta propiedad, pasándole un objeto Insertion válido. Por ejemplo, la siguiente instrucción agregaría el texto de respuesta al contenido existente en el elemento, situándolo en la parte inferior del mismo:

```
function validar(){
     //forma la cadena con los parámetros de la petición
     var parametros="txtusuario="+
             document.getElementById("txtusuario").
                                   value +"&";
     parametros+="txtpassword="+
             document.getElementById("txtpassword").value;
     //forma la cadena del objeto JSON
     var objson="{method:'POST',
             parameters:'"+parametros+"',";
     objson+="insertion:Insertion.Bottom}";
     objson=eval("("+objson+")");
     var obj = new
              Ajax.Updater("resultado","validar",objson);
}
```

3.4.1.2 UTILIDADES PROTOTYPE

Como ya se ha indicado, además de los objetos anteriores, prototype proporciona también un conjunto de funciones y objetos auxiliares de apoyo. He aquí algunas de las funciones más importantes:

- **$(identificador)**. Esta función encapsula la funcionalidad del método *getElementById()* del DOM, devolviendo una referencia a uno de los elementos de la página a partir de su identificador. Admite un número variable de identificadores, en cuyo caso se devolverá un array con los elementos correspondientes.

- **$$(clase)**. A partir del nombre de clase definida en una hoja de estilo CSS, devuelve un array con los elementos que tengan aplicado dicho estilo.

- **$F(identificador)**. Devuelve el contenido de la propiedad *value* del control HTML cuyo identificador se proporciona como parámetro.

- **$w(cadena)**. Genera un array a partir de la cadena de caracteres indicada, utilizando el espacio en blanco como separador de elementos.

El siguiente listado corresponde a una tercera versión del ejemplo de validación de usuarios, en el que se hace uso de alguna de las funciones anteriores:

```
function validar(){
     //forma la cadena con los parámetros de la petición
     //y utiliza la función $F() para obtener los valores
     //de los campos de texto
     var parametros="txtusuario="+$F("txtusuario") +"&";
     parametros+="txtpassword="+$F("txtpassword");
     //forma la cadena del objeto JSON
     var objson="{method:'POST',
             parameters:'"+parametros+"',";
     objson+="onComplete:muestraResultado}";
     objson=eval("("+objson+")");
     var obj = new Ajax.Request("validar",objson);
}
function muestraResultado(objeto){
     //utiliza la función $() para acceder al elemento
     //div "resultado"
     $("resultado").innerHTML=objeto.responseText;
}
```

Además de estas funciones, prototype incorpora los siguientes objetos de utilidad:

- **Element**. Proporciona métodos que permiten modificar las características de la página, alterando las propiedades de sus elementos. Estos métodos pueden ser aplicados sobre cualquier referencia a un objeto HTML existente:

  - **hide()**. Oculta el elemento sobre el que se aplica.

  - **show()**. Muestra el elemento.

  - **update(nuevocontenido)**. Reemplaza el contenido del elemento sobre el que se aplica por un nuevo contenido XHTML. Equivale a asignar un nuevo valor a la propiedad *innerHTML* del elemento. Por ejemplo, la función *muestraResultado()* del ejemplo anterior podría escribirse de la siguiente manera:

```
function muestraResultado(objeto){
```

```
        $("resultado").update(objeto.responseText);
    }
```

- **Insertion**. Este objeto proporciona soporte para insertar la respuesta en una determinada zona de la página. Insertion incluye la definición de una serie de clases cuyos objetos se utilizan para especificar el punto de inserción de los datos. A continuación, se indica el formato de los constructores de estas clases:

  - **Insertion.Before(id_elemento, cadena)**. Inserta la cadena de caracteres especificada en el segundo parámetro delante del elemento cuyo identificador es indicado en el primer parámetro. La cadena de caracteres puede incluir caracteres de marcado.

    Por ejemplo, dado el siguiente bloque HTML:

    ```
    <div>
          <h1 id="bloque1">Párrafo inicial</h1>
    <div>
    ```

    tras ejecutar la siguiente instrucción:

    ```
    new Insertion.Before("bloque1",
          "<h1>Nuevo párrafo inicial</h1>");
    ```

    el bloque anterior quedaría:

    ```
    <div>
    <h1>Nuevo párrafo inicial</h1>
    <h1 id="bloque1">Párrafo inicial</h1>
    <div>
    ```

  - **Insertion.After(id_elemento, cadena)**. Inserta la cadena a continuación del elemento especificado.

  - **Insertion.Bottom(id_elemento, cadena)**. Inserta la cadena como último hijo del elemento especificado.

  - **Insertion.Top(id_elemento, cadena)**. Inserta la cadena como primer hijo del elemento especificado.

- **Form**. Proporciona métodos para trabajar con formularios y realizar útiles operaciones con los campos de éste. Entre los métodos más interesantes están:

  - **Form.disable(idform)**. Deshabilita el formulario cuyo identificador se pasa como parámetro, haciendo que todos los campos sean de sólo lectura.

  - **Form.enable(idform)**. Habilita el formulario indicado.

  - **Form.serialize(objetoform)**. Devuelve una cadena de caracteres formada por los nombres y valores de los controles del objeto formulario suministrado como parámetro, formateados de forma que puedan ser enviados en una petición HTTP:

    nombrecontrol=valor&nombrecontrol=valor.

PRÁCTICA 3.3. LISTAS DE SELECCIÓN ENLAZADAS

Descripción

Vamos a crear en esta práctica un tipo de aplicación que fue utilizada como ejemplo en el primer capítulo del libro cuando expusimos los escenarios de utilización de AJAX.

Se trata de una página Web con listas de selección enlazadas, donde la selección de un elemento en una de las listas determina los elementos a mostrar en la otra lista.

Este es el caso de la lista de localidades de provincia cuyo contenido dependerá de la provincia elegida en una lista previa, de modo que al seleccionar una provincia se cargarán en la lista de localidades únicamente aquellas que pertenezcan a la provincia seleccionada (figura 22).

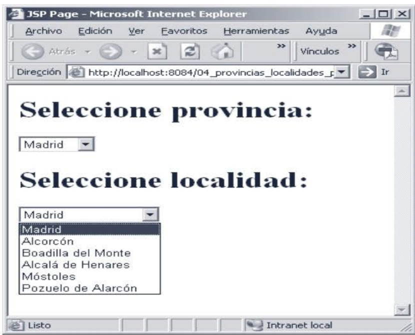


Fig. 22. *Página Web de la aplicación*

Desarrollo

Para implementar esta aplicación utilizaremos una base de datos donde se almacenarán en sendas tablas las provincias con sus respectivas localidades, tal y como se refleja en el diagrama de la figura 23.

Provincias

Campo	**Tipo**
idProvincia	numérico entero
provincia	cadena

Localidades

Campo	**Tipo**
idLocalidad	numérico entero
idProvincia	numérico entero
localidad	cadena

Fig. 23. *Tablas de la base de datos*

Para la implementación de la página Web utilizaremos una página JSP que incluirá cierta lógica de servidor, cuya finalidad será la creación de la lista de provincias a partir de la información almacenada en la base de datos. Por su parte, el código del servidor será implementado mediante un servlet que se apoyará en una clase llamada Gestion en la que se encapsulará la lógica de la aplicación.

El la figura 24 se muestra el diagrama de funcionamiento de la aplicación. En él podemos ver como la primera petición que se realiza directamente desde el navegador al servlet provoca que se envíe al cliente la página con los datos de

todas las provincias (método *getProvincias()*), manteniéndose vacía la lista de localidades.

Al seleccionar una provincia, se lanzará una petición AJAX al servlet enviándole como parámetro el código de la provincia seleccionada, éste utilizará ese dato para recuperar de la base de datos las localidades asociadas (método *getLocalidades()*) y devolvérselas como respuesta a la aplicación AJAX cliente que las cargará en la lista de localidades.

Los datos enviados en la respuesta a la aplicación AJAX cliente se formatearán como un array de objetos JSON, donde cada uno de estos objetos tendrá dos propiedades: una que contendrá el código de la localidad y otra el nombre de la misma.

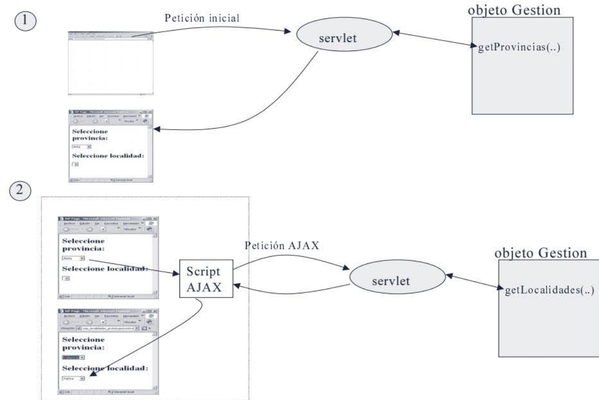


Fig. 24. *Diagrama de funcionamiento de la aplicación*

Por su parte, la aplicación cliente utilizará la librería prototype para la realización de las peticiones AJAX y la manipulación de las respuestas.

Listado

A continuación, se muestra el código de los distintos componentes de la aplicación.

Clase Gestion

```
package modelo;
import java.sql.*;
import beans.*;
import java.util.*;
public class Gestion {
    private Connection getConection(){
        Connection cn=null;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            cn=DriverManager.
                    getConnection("jdbc:odbc:localidades");
        }
        catch(Exception e){e.printStackTrace();}
        finally{
            return cn;
        }
    }
    public ArrayList getProvincias(){
        ArrayList<Provincia> provincias=
                    new ArrayList<Provincia>();
        try{
            String query = "select * from provincias";
            Connection cn=this.getConection();
            Statement st =cn.createStatement();
            ResultSet rs = st.executeQuery(query);
            //generación de un ArrayList con los datos de
            //las provincias
            while(rs.next()){
              Provincia p=new Provincia();
              p.setId(rs.getInt("idProvincia"));
              p.setProvincia(rs.getString("provincia"));
              provincias.add(p);
            }
        }
        catch(Exception e){e.printStackTrace();}
        finally{
            return provincias;
        }
    }
```

```
    public String getLocalidades(int prov){
        StringBuilder json=new StringBuilder("[");
        try{
            String query = "select * from localidades ";
            query+="where idProvincia="+prov;
            Connection cn=this.getConection();
            Statement st =cn.createStatement();
            ResultSet rs = st.executeQuery(query);
            //generación de un array de objetos JSON con
            //los datos de las localidades
            while(rs.next()){
              json.append("{codigo:"+
                    rs.getString("idLocalidad")+",");
              json.append("localidad:'"+
                    rs.getString("localidad")+"'},");
            }
            //eliminación de la última coma
            json.deleteCharAt(json.length()-1);
            json.append("]");
        }
        catch(Exception e){e.printStackTrace();}
        finally{
            return json.toString();
        }
    }
}
```

Clase Provincia

```
package beans;

public class Provincia {
    private int id;
    private String provincia;
    public Provincia() {
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
```

```
        }
        public String getProvincia() {
            return provincia;
        }
        public void setProvincia(String provincia) {
            this.provincia = provincia;
        }
}
```

Servlet control

```
package servlets;

import java.io.*;
import java.net.*;
import modelo.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Control extends HttpServlet {
    protected void service(HttpServletRequest request,
                           HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        String op=request.getParameter("operacion");
        //determina que debe generar la página inicial
        if(op==null){
            Gestion g=new Gestion();
            request.setAttribute("provincias",
                          g.getProvincias());
            RequestDispatcher rd=request.
                   getRequestDispatcher("/index.jsp");
            rd.forward(request,response);
        }
        //determina que debe generar la lista de localidades
        if(op.equals("localidades")){
            Gestion g=new Gestion();
            String prov=request.getParameter("provincia");
            out.println(g.getLocalidades(
                          Integer.parseInt(prov)));
        }
```

```
        out.close();
    }
}
```

Página formulario.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@page import="java.util.*,beans.*"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                     Transitional//EN"
                     "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type"
                   content="text/html; charset=UTF-8">
        <!--referencia a la librería prototype-->
        <script language="javascript"
                   src="prototype.js"></script>
        <script language="javascript">

            function peticion(){
                //configura y crea el objeto Ajax.Request
                var objson={method:"GET",
                          onComplete:cargaLocalidades};
                var lista=$("provincia");
                var id=lista.options[
                          lista.selectedIndex].value;
                var url="control?operacion=
                          localidades&provincia="+id;
                var obj=new Ajax.Request(url,objson);
            }
            function cargaLocalidades(objeto){
                borrarLista();
                var array=eval("("+objeto.responseText+")");
                for(var i=0;i<array.length;i++){
                    generaOption(array[i]);
                }
            }
```

```
        function generaOption(loc){
            //genera un elemento <option> a partir
            //del objeto JSON con los datos de la
            //localidad y lo añade a la lista
            var texto=document.
                      createTextNode(loc.localidad);
            var elemento=document.
                      createElement("option");
            elemento.setAttribute("value",loc.codigo);
            elemento.appendChild(texto);
            $("localidades").appendChild(elemento);
        }
        function borrarLista(){
           //elimina todos los elementos <option>
           //de la lista
            var lista=$("localidades");
            for(var i=lista.childNodes.length-1;
                                   i>=0;i--){
                lista.removeChild(
                      lista.childNodes.item(i));
            }
        }
    </script>
</head>
<body>
<h1>Seleccione provincia:</h1>
<select name="provincia" id="provincia"
                       onchange="peticion();">
    <%ArrayList provincias=(ArrayList)request.
                      getAttribute("provincias");%>
    <%//genera la lista de provincias
       for(int i=0;i<provincias.size();i++){
        Provincia p=(Provincia)provincias.get(i);%>
        <option value="<%=p.getId()%>">
            <%=p.getProvincia()%>
        </option>
    <%}%>
</select>
<br/><br/>
<h1>Seleccione localidad:</h1>
<select id="localidades">
```

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

3.4.2 La librería script.aculo.us

Script.aculo.us es una librería basada en prototype que proporciona un conjunto de objetos con los que se puede crear una amplia variedad de efectos visuales en las aplicaciones, además de controles gráficos avanzados para mejorar la interacción del usuario con la página.

En la dirección *http://script.aculo.us/download* podemos encontrar amplia información sobre todos estos objetos, así como las librerías de código .js para su descarga y utilización.

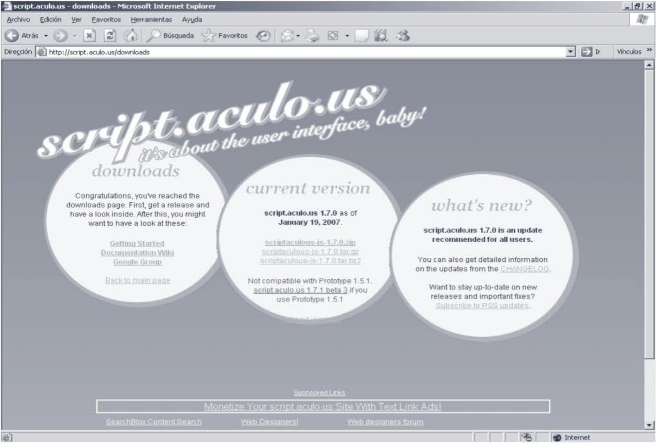


Fig. 25. *Página principal de script.aculo.us*

Todo el conjunto de librerías que forman scrip.aculo se encuentran en un archivo comprimido .zip que puede ser descargado desde la sección "current version" de la página anterior.

Cada conjunto de objetos se encuentra en una librería independiente, las más interesantes son:

- **builder.js**. Incluye objetos para la creación dinámica de nuevos elementos HTML en la página, encapsulando la funcionalidad DOM.
- **controls.js**. Esta librería contiene controles visuales avanzados, como por ejemplo Ajax.Autocompleter para la generación de cuadros de autorrelleno.
- **dragdrop.js**. Proporciona el soporte para arrastrar y soltar en la página.
- **effects.js**. Es quizás la librería más espectacular de script.aculo.us, pues es la que proporciona los objetos con los que se pueden generar el amplio abanico de efectos visuales.

Para poder utilizar estos objetos, debemos añadir en la página una referencia tanto al archivo prototype.js como a scriptaculo.js, esta última se encargará de **cargar dinámicamente el código de los archivos .js** mencionados anteriormente.

PRÁCTICA 3.4. EFECTOS VISUALES

Descripción

Se trata de probar alguno de los efectos visuales proporcionados por script.aculo.us. Para ello, crearemos una página Web con un texto de prueba y cuatro botones de pulsación que al ser pulsados aplicarán un determinado efecto sobre el texto (figura 26).

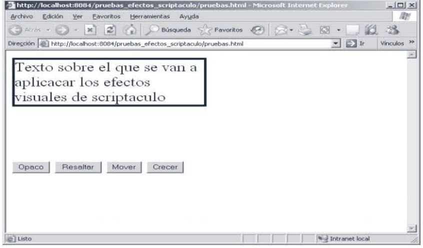


Fig. 26. *Página para prueba de efectos*

Desarrollo

Para la realización de esta práctica, utilizaremos los objetos Effect de script.aculo. Basta con instanciar el objeto para que se aplique el efecto correspondiente sobre el elemento indicado.

La sintaxis para crear un objeto Effect es la siguiente:

new Effect.NombreEfecto(elemento, parametros);

donde *NombreEfecto* es el nombre del efecto que aplicará el objeto, *elemento* es el identificador o referencia al elemento (depende del objeto utilizado) sobre el que se aplicará el efecto y *parametros* es un objeto JSON con la lista de parámetros requerida por el objeto.

En esta práctica probaremos los efectos *Opacity*, *Highlight*, *Move* y *Grow*.

A través del manejador del evento *onclick* de cada botón se invocará a una función JavaScript que aplique el efecto asociado a ese botón.

Listado

El siguiente listado corresponde a la vista de código de la página Web, resaltándose en fondo sombreado las instrucciones JavaScript asociadas a la aplicación de efectos script.aculo:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                           Transitional//EN">
<html>
  <head>
     <!-- referencia a archivos .js de script.aculo-->
     <script src="prototype.js" type="text/javascript">
     </script>
     <script src="scriptaculous.js" type="text/javascript">
     </script>
     <script>
         function efectoOpaco(){
            new Effect.Opacity('texto',
                { duration: 5.0,
                transition: Effect.Transitions.linear,
                from: 1.0, to: 0.1 });
         }
         function efectoResaltado(){
```

```
            new Effect.Highlight('texto',
                { startcolor:'#aacc11',endcolor:'#ffffff' });

         }
         function efectoMovimiento(){
            new Effect.Move ($('texto'),{ x: 100, y: 150,
                                            mode: 'relative'});
         }
         function efectoCrecer(){
            Effect.Grow('texto');
         }
      </script>
      <!--estilo inicial del bloque de texto-->
      <style>
          .inicial{font-size:20pt;width:50%;
                           border-style:solid}
      </style>
   </head>
   <body>
       <div id="texto" class="inicial">
           Texto sobre el que se van a aplicar
           los efectos visuales de script.aculo
       </div>
       <br/><br/><br/><br/><br/><br/>
       <input type="button" value="Opaco"
                     onclick="efectoOpaco();"/> 
       <input type="button" value="Resaltar"
                     onclick="efectoResaltado();"/> 
       <input type="button" value="Mover"
                     onclick="efectoMovimiento();"/> 
       <input type="button" value="Crecer"
                     onclick="efectoCrecer();"/>
   </body>
</html>
```

PRÁCTICA 3.5. CUADRO DE AUTORELLENADO

Descripción

Se trata de realizar una nueva versión mejorada del cuadro de autocompletado desarrollado en la práctica 2.2. En este caso, según vaya escribiendo caracteres el usuario en el cuadro de texto, se presentará una lista con

las opciones encontradas que se ajusten a los caracteres introducidos hasta el momento (figura 27).

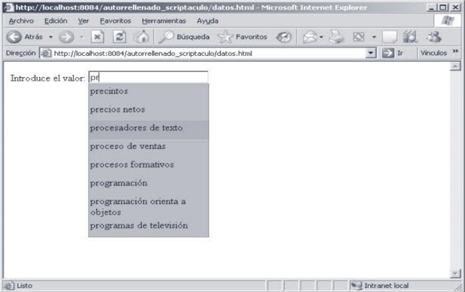


Fig. 27. *Cuadro de autorrellenado*

Sobre esta lista, el usuario podrá elegir la palabra o frase buscada para que quede escrita en el cuadro de texto.

Desarrollo

Para este desarrollo haremos uso del control Ajax.Autocompleter de la librería controls.js. Se trata de un potente control que, a través de AJAX, conecta con el proceso de servidor, para solicitarle la lista de palabras asociadas a los caracteres introducidos cada vez que el usuario escribe sobre el campo de texto. El propio control se encarga de rellenar la lista y, una vez seleccionada una palabra de la lista, volcarla en el campo de texto.

El formato para la creación de este objeto es el siguiente:

new Ajax.Autocompleter(id_text, id_div_lista, url, opciones)

donde *id_text* es el identificador del campo de texto en el que el usuario introduce los caracteres de búsqueda; *id_div_lista*, el identificador del elemento de tipo <div> donde se mostrará la lista; *url*, la URL del proceso de servidor, y *opciones*, un objeto JSON con opciones adicionales para el control, entre las que podemos destacar:

- **paramName**. Nombre del parámetro que se enviará en la petición AJAX con los caracteres introducidos por el usuario.

- **minChars**. Número de caracteres que debe introducir el usuario para que se produzca la petición.

Listado

A continuación, se muestra el listado correspondiente al servlet y a la página Web de la aplicación.

Servlet rellenado

```
package servlets;

import java.io.*;
import java.net.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;


  public class Rellenado extends HttpServlet {
    protected void service(HttpServletRequest request,
                    HttpServletResponse response)
                   throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        String dato=request.getParameter("texto");
        Connection cn=null;
        Statement st=null;
        ResultSet rs=null;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            cn=DriverManager.
                    getConnection("jdbc:odbc:palabras");
            st=cn.createStatement();
            //busca la palabra/frase que se ajuste
            //al texto recibido, ordenándose alfabéticamente
            //los resultados
            String sql="Select palabra from palabras where ";
            sql+="palabra like '"+dato;
            sql+="%' order by palabra";
            rs=st.executeQuery(sql);
            //los datos deben ser enviados en como una
```

```
            //lista no numerada
            String result="<ul>";
            while(rs.next()){
               result+=
                    "<li>"+rs.getString("palabra")+"</li>";
            }
            result+="</ul>";
             //envío de la respuesta
            out.println(result);
        }
        catch(Exception e){e.printStackTrace();}
        out.close();
    }
}
```

Página Web datos.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                                  Transitional//EN">

<html>
  <head>
      <!--referencia a las librerías de utilidades-->
      <script src="prototype.js" type="text/javascript">
      </script>
      <script src="scriptaculous.js" type="text/javascript">
      </script>
      <script language="javascript">
          function completar(){
            //crea y configura el objeto Autocompleter
            //estableciendo 'texto' como nombre del parámetro
            //que incluirá el dato introducido en la caja
            //de texto y una frecuencia de actualización
            //de la lista de un carácter
            new Ajax.Autocompleter("dato","lista",
               "rellenado",{paramName:"texto",minChars:1});
          }
      </script>

      <!--hoja de estilos para dar formato a la lista-->
      <style>
            div.lista {
```

```
                position:absolute;
                width:250px;
                background-color:aqua;
                border:1px solid #888;
                margin:0px;
                padding:0px;
              }
              div.lista ul {
                list-style-type:none;
                margin:0px;
                padding:0px;
              }


              div.lista ul li.selected {
                     background-color: #a5bf10;
              }
              div.lista ul li {
                list-style-type:none;
                display:block;
                margin:0;
                padding:2px;
                height:32px;
                cursor:pointer;
              }
        </style>
    </head>
    <body onload="completar();">
      Introduce el valor:
      <input type="text" id="dato" size="25"/>
      <!--lista con los datos-->
      <div id="lista" class="lista">
         
      </div>
    </body>
  </html>
```

3.4.3 El framework jQuery

jQuery es el framework para JavaScript más completo de los existentes actualmente en la red. Consta de un conjunto de utilidades orientadas a simplificar de manera considerable la manipulación del árbol de objetos DOM, la interacción con los componentes HTML de la página, así como el tratamiento de las solicitudes AJAX.

jQuery incluye numerosos selectores y funciones con los que podemos llevar a cabo las operaciones habituales en una aplicación AJAX de manera muy sencilla, como el acceso a las propiedades de un objeto HTML, el envío de una petición AJAX o la manipulación de los datos de respuesta.

3.4.3.1 UTILIZACIÓN DE JQUERY

En primer lugar, para poder utilizar jQuery en una aplicación, debemos descargarnos una librería JavaScript correspondiente desde la página oficial (*http://jquery.com/*).

En el momento de escribir este libro, la última versión existente de jQuery es la 1.5, mientras que el archivo a descargar con la librería JavaScript se llama jquery-1.5.min.js.

De cara a poder usar esta librería en la página, habría que incluir la correspondiente etiqueta <script> con la referencia al archivo .js. Si suponemos que el archivo y la página se encuentran en el mismo directorio sería:

```
<head>
      <script src="jquery-1.5.min.js"
            type="text/javascript">
      </script>
</head>
```

3.4.3.2 COMPONENTES JQUERY

A continuación, vamos a analizar los diferentes elementos que nos ofrece jQuery para ser utilizados en una aplicación JavaScript. Estos elementos o componentes podríamos dividirlos en las siguientes categorías:

- Selectores
- Manipulación DOM
- Eventos

- Efectos
- Funciones AJAX

3.4.3.2.1 Selectores

Como su nombre indica, un selector es un componente jQuery que permite seleccionar un conjunto de elementos de la página a partir de un determinado criterio. Para utilizar un selector empleamos la expresión $(...), donde el contenido indicado entre paréntesis dependerá del tipo de selector utilizado.

Todos los selectores jQuery están basados en la especificación CSS 1 del W3C.

Entre los selectores más utilizado de jQuery tenemos los siguientes:

- **$("*")**. Devuelve una referencia a un array con todos los objetos HTML de la página.
- **$("nombre_elemento")**. Devuelve una referencia a un array con todos los elementos del tipo indicado en "nombre_elemento". Por ejemplo, la siguiente instrucción nos indicaría en un cuadro de diálogo el total de elementos tipo párrafo existentes en la página:

```
alert($("p").length);
```

- **$("#id_elemento")**. Devuelve una referencia al elemento cuyo id se especifica. El siguiente código modifica el contenido HTML de un elemento con id llamado "datos":

```
$("#datos").html("<b>nuevo texto</b>");
```

Lo anterior es equivalente al siguiente código JavaScript:

```
document.getElementById("datos").innerHTML =
"<b>nuevo texto</b>";
```

- **$(".clase_estilo")**. Devuelve una referencia al array de elementos HTML que tienen aplicada una determinada clase de estilo. El siguiente listado de ejemplo modificaría el contenido del párrafo:

```
<div class="estilo1">datos en div</div>
<p class="estilo1">contenido parrafo</p>
<script>
```

```
$(".estilo1")[1].html("<b>nuevo contenido</b>");
</script>
```

- **$(selector1, selector2,..)**. Se trata de un selector múltiple con el que se pueden combinar cualquiera de las opciones de selector anteriores. La siguiente expresión haría referencia al array formado por todos los elementos de tipo párrafo más aquel que tenga como identificador "prueba":

```
$("p", "#prueba")
```

3.4.3.2.2 Manipulación DOM

Se trata de métodos que pueden ser aplicados sobre objetos HTML, por ejemplo, referenciados a través de selectores, con los que podemos manipular el árbol DOM de la página.

Entre los más destacados se encuentran:

- **attr("nombre_atributo")**. Devuelve el valor del atributo indicado que en ese momento tenga aplicado el elemento. El siguiente ejemplo muestra el valor del atributo *bgColor* del body:

```
alert($("body").attr("bgColor"));
```

- **attr("nombre_atributo", valor)**. Se trata de una variante del método anterior que permite establecer un determinado valor sobre el atributo del elemento. La siguiente instrucción establece el color de fondo de la página en amarillo:

```
$("body").attr("bgColor","yellow");
```

- **.css("propiedad_estilo")**. Devuelve el valor de la propiedad de estilo indicada que en ese momento tenga asignada el elemento. Por ejemplo, la siguiente expresión devuelve el color de fondo de un elemento cuyo id sea "texto":

```
$("#texto").css("background-color")
```

- **.css("propiedad_estilo",valor)**. Esta variante del anterior permite establecer un valor sobre la propiedad de estilo del elemento.

- **.text()**. Devuelve el contenido del elemento y todos sus descendientes. Por ejemplo, dado el siguiente bloque HTML:

```
<div id="capa">
        <h1>Titulo principal</h1>
        <p>parrafo1</p>
        <p>parrafo2</p>
</div>
```

  Al ejecutar la siguiente instrucción:

```
alert($("#capa").text());
```

  se mostraría el siguiente texto en el cuadro de diálogo:

  Titulo principal parrafo1 parrafo2

  También puede ser utilizado este atributo para modificar el texto de un elemento:

```
$("#capa").text("nuevo texto");
```

- **.html()**. Igual que la anterior, pero devolviendo todo el contenido HTML del elemento sobre el que se aplica. Como vimos en los ejemplos de los selectores, también puede utilizarse para modificar el contenido HTML de un elemento.

- **.append(contenido)**. Inserta el contenido especificado en el parámetro (texto, HTML u objeto jQuery) dentro del elemento, como último hijo del mismo.

  Dado el siguiente código:

```
<div class="estilo1"><b>primer dato</b> </div>
<script>
$(".estilo1").append("<b>segundo dato</b>");
</script>
```

  Tras la ejecución del script, el código fuente de la página sería equivalente a lo siguiente:

```
<div class="estilo1"><b>primer dato</b>
 <b>segundo dato</b></div>
```

Si hubiera más de un elemento que cumpliera la condición del selector, se aplicaría sobre todos ellos.

- **.appendTo(contenido)**. Similar al anterior, sólo que el contenido se añade como primer hijo del elemento.

- **.remove()**. Elimina un objeto del árbol DOM. Por ejemplo, dado el siguiente código:

```
<div>
<p id="parrafo1">valor parrafo 1 </p>
<p id="parrafo2"> valor parrafo 2 </p>
</div>
<script>
$("#parrafo1").remove();
</script>
```

  Tras la ejecución del script, la página quedaría:

```
<div>
<p id="parrafo2"> valor parrafo 2 </p>
</div>
```

  El método anterior admite como parámetro opcional un selector, a fin de establecer un filtro de borrado sobre un conjunto de elementos. Por ejemplo, lo anterior podría haberse realizado también con la instrucción:

```
$("div").remove("#parrafo1");
```

- **.val()**. Se emplea para obtener el valor de un control existente en un formulario HTML. La siguiente expresión de ejemplo recupera el valor del campo de texto con identificador “user”:

```
$("#user").val()
```

  También puede emplearse para modificar su contenido:

```
$("#user").val("usuario nuevo");
```

3.4.3.2.3 Eventos

Se trata de métodos que pueden ser aplicados sobre objetos HTML, normalmente obtenidos a través de un selector, para asociar un determinado evento sobre el objeto con su manejador. Entre los más destacados tenemos:

- **.bind("nombre_evento", funcion_manejadora)**. Asocia un determinado evento producido sobre el objeto con una función manejadora. El primer parámetro representa el nombre del evento sin el prefijo "on", como, por ejemplo, "click", "mouseover", etc. El segundo parámetro representa la función manejadora.

  El siguiente ejemplo asocia una función que muestra un mensaje de alerta al evento "click" sobre un párrafo de identificador "pr1":

  ```
  $("#pr1").bind("click",function(){

           alert("hizo click");});
  ```

- **.unbind("nombre_evento")**. Elimina las funciones asociadas al evento sobre el elemento.

3.4.3.2.4 Efectos

Son un grupo de métodos con los que podemos aplicar diferentes efectos visuales sobre los elementos de la página. He aquí algunos de los más destacados:

- **.hide()**. Oculta un elemento

- **.show()**. Muestra un elemento que estaba oculto.

- **.show(duracion)**. Es una variante del anterior en la que se puede especificar mediante un String la duración del efecto. Sus valores posibles son "fast" y "slow". El siguiente ejemplo muestra el elemento con una animación suave:

  ```
  $("#parrafo1").show("slow");
  ```

- **.toggle()**. Si el elemento estaba oculto lo muestra, y si estaba visible lo oculta.

- **.fadeOut()**. El elemento desaparece con un efecto de fundido suave.

3.4.3.2.5 Funciones AJAX

jQuery incluye una serie de funciones y métodos a aplicar sobre elementos que realizan las tareas habituales relacionadas con la funcionalidad AJAX, implicando una drástica reducción de código JavaScript.

Seguidamente, analizamos algunas de las más importantes:

- **load("Url_servidor")**. Se trata de uno de los métodos más potentes y prácticos de jQuery; realiza una petición AJAX a la URL indicada como parámetro e inserta en el interior del elemento el contenido obtenido en la respuesta.

  Por ejemplo, si tenemos un elemento <div> con identificador "resultado", en el que queremos mostrar el texto enviado por el servlet con URL "generador", todo lo que tendríamos que hacer para lanzar la petición AJAX e incluir la respuesta en el elemento sería:

  ```
  $("#resultado").load("generador");
  ```

- **get("url_servidor", lista_parametros_json,funcion_respuesta)**. Se trata de una función AJAX de jQuery que lanza una petición get al servidor, con la lista de parámetros de la cadena JSON especificada en el segundo parámetro, y que ejecutará como respuesta la función definida en el tercer parámetro. La función de retrollamada recibe como parámetro la cadena de texto enviada como respuesta por el servidor.

  La siguiente instrucción lanza una petición AJAX al servlet "login", pasando como parámetro el usuario y el password, y nos muestra el resultado en un cuadro de diálogo:

  ```
  $.get("login",
          {user:$("#user").val(), pwd:$("#pwd").val()},
          function(data){ alert(data);});
  ```

- **post("url_servidor", parametros_json, funcion_respuesta)**. Realiza la misma operación que la anterior función, pero utilizando método de envío POST.

- **getJSON("url_servidor", parametros_json, funcion_respuesta)**. Realiza una petición get, donde el dato recibido por la función de retrollamada es un objeto en formato JSON.

Por ejemplo, si el servlet login del ejemplo anterior devolviera un objeto JSON con el nombre, edad y teléfono del cliente, el siguiente script mostraría dichos datos tras lanzar la petición AJAX al servidor:

```
$.getJSON("login",
{user:$("#user").val(), pwd:$("#pwd").val()},
function(data){ alert("Nombre:"+data.nombre);
                alert("Edad:"+data.edad);
                alert("Teléfono:"+data.telefono);
});
```

PRÁCTICA 3.6. CATÁLOGO DE LIBROS

Descripción

Vamos a realizar una nueva versión de la aplicación del catálogo de libros desarrollada en la práctica 3.1. En esta ocasión, incluiremos la funcionalidad de autenticación de usuarios, de modo que sólo los usuarios registrados podrán consultar el catálogo de libros; si se suministran unos credenciales incorrectos, se mostrará un mensaje de advertencia en la página, mientras que si son correctos se visualizará la lista de temas para poder elegir aquel cuyos libros queramos visualizar (figura 28)

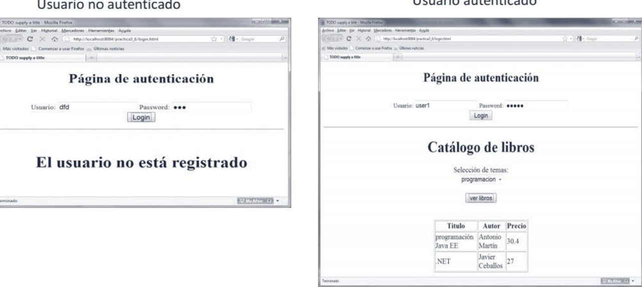


Fig. 28. *Página de la aplicación*

Desarrollo

Lo primero que necesitamos hacer para el desarrollo de esta práctica es incluir una nueva tabla en la base de datos que registre los credenciales de usuarios válidos. Esta tabla lo único que deberá tener son dos campos para almacenar los usuarios y password válidos.

Por otro lado, toda la funcionalidad AJAX de cliente será implementada mediante jQuery, tanto las peticiones de autenticación de usuarios como las de muestra de libros.

La parte de la página que está debajo de la zona de autenticación, se dividirá en dos capas, inicialmente ocultas. En una de ellas se incluye el mensaje de usuario no autenticado y, en la otra, la lista de temas, inicialmente vacía, más una subcapa donde se incluirá la tabla de libros. En función de la respuesta del servlet de login, que será llamado con la pulsación del botón "login", se ocultará/mostrará cada capa.

Código

En primer lugar, vamos a mostrar el código de los servlets login y libros, encargados de la autenticación de usuarios y recuperación de libros, respectivamente.

El servlet de login, además de comprobar si el usuario está registrado, recupera la lista de temas en caso de que lo esté, devolviendo dicha lista de temas en una cadena JSON.

Un detalle que hay que destacar en el código de ambos servlets, es la utilización de la secuencia de escape \" para incluir las comillas dobles dentro de las cadenas JSON, puesto que si **se utilizan comillas simples éstas no serían interpretadas correctamente por la función jQuery getJSON utilizada para lanzar la petición AJAX.**

Servlet login

```
package servlets;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
import java.sql.*;

public class Login extends HttpServlet {
   protected void service(HttpServletRequest request,
             HttpServletResponse response)
    throws ServletException, IOException {
       response.setContentType("text/plain;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            if(estaRegistrado(request.getParameter("user"),
                          request.getParameter("pwd"))){
               out.println(obtenerTemas());
            }
            else{
               String salida="{\"noautorizado\":\"true\"}";
               out.println(salida);
            }
        } finally {
            out.close();
        }
    }
    private boolean estaRegistrado(String user, String pwd){
        Connection cn=null;
        Statement st=null;
        ResultSet rs=null;
        boolean resultado=false;

        try{
            //en este ejemplo se utiliza un
            //driver nativo de MySQL
            Class.forName("com.mysql.jdbc.Driver");
            cn=DriverManager.getConnection(
      "jdbc:mysql://localhost:3306/bdlibros","root","root");
            st=cn.createStatement();
            String sql="select * from autorizados where
                   usuario='"+user+"'";
            sql+=" and password='"+pwd+"'";
            rs=st.executeQuery(sql);
            if(rs.next()){
                resultado=true;
```

```
                }
            }
            catch(Exception e){e.printStackTrace();}
            return resultado;

        }
        private String obtenerTemas(){
            Connection cn=null;
            Statement st=null;
            ResultSet rs=null;
            StringBuilder objSecciones=null;
            try{
                //en este ejemplo se utiliza un
                //driver nativo de MySQL
                Class.forName("com.mysql.jdbc.Driver");
                cn=DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/bdlibros","root","root");
                st=cn.createStatement();
                rs=st.executeQuery("select * from temas");

                //crea un array de objetos JSON con cada
                //tema de la tabla
                objSecciones=new StringBuilder("[");
                while(rs.next()){
                    objSecciones.append("{\"id\":");
                    objSecciones.append(rs.getString("idTema"));
                    objSecciones.append(",\"nombre\":\"");
                    objSecciones.append(rs.getString("tema"));
                    objSecciones.append("\"},");
                }
                //sustituye la última coma por el
                 //carácter de cierre del array
                objSecciones.replace(objSecciones.length()-1,
                              objSecciones.length(),"]");
            }
            catch(Exception e){e.printStackTrace();}
            return  objSecciones.toString();
        }
    }
```

Servlet libros

```
package servlets;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.sql.*;

public class Libros extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/plain;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            String tema=request.getParameter("idTema");
            out.println(obtenerLibros(tema));
        } finally {
            out.close();
        }
    }
    private String obtenerLibros(String tema){
        Connection cn=null;
        Statement st=null;
        ResultSet rs=null;
        StringBuilder objLibros=null;
        try{
            //en este ejemplo se utiliza un
            //driver nativo de MySQL
            Class.forName("com.mysql.jdbc.Driver");
            cn=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/bdlibros","root","root");
            st=cn.createStatement();
            //busca los libros asociados al tema elegido
            String tsql="select * from libros ";
            tsql+="where idTema="+tema;
            rs=st.executeQuery(tsql);
```

```
                //crea un array de objetos JSON con cada
                //libro que cumpla la condición
                objLibros=new StringBuilder("[");
                while(rs.next()){
                    objLibros.append("{\"titulo\":\"");
                    objLibros.append(rs.getString("titulo"));
                    objLibros.append("\",\"autor\":\"");
                    objLibros.append(rs.getString("autor"));
                    objLibros.append("\",\"precio\":");
                    objLibros.append(rs.getString("precio"));
                    objLibros.append("},");
                }
                //sustituye la última coma por el
                 //carácter de cierre del array
                objLibros.replace(objLibros.length()-1,
                        objLibros.length(),"]");
            }
            catch(Exception e){e.printStackTrace();}
            return  objLibros.toString();
        }
    }
```

Hay que tener en cuenta, además, que tanto el servlet login como libros deberán ser registrados en el archivo de configuración web.xml

Por su parte, la página Web de la aplicación será implementada como se indica en el siguiente listado.

Pagina login.html

```
<html>
    <head>
        <meta http-equiv="Content-Type"
             content="text/html; charset=UTF-8"/>
        <title>TODO supply a title</title>
         <script src="jquery-1.5.min.js"
             type="text/javascript">
        </script>
```

```
<script language="javascript">
    function login(){
      //realiza la petición de login

      $.getJSON("login",{user:$("#txtusuario").val(),
            pwd:$("#txtpassword").val()},procesa);
    }
    function procesa(data){
        if(data.noautorizado){
            //si el usuario no está registrado
            //muestra la capa con el mensaje de error
            $("#noLogin").css("display", "block");
            alert("no autorizado");
        }
        else{
            //si el usuario está registrado
            //muestra la capa de selección de temas
            $("#libros").css("display", "block");
            $("#noLogin").css("display", "none");
            //recorre los temas y construye un option
            //para cada uno
            $.each(data, function(i,item){
                var op=document.
                  createElement("option");
                op.setAttribute("value",item.id);
                var texto=document.
                  createTextNode(item.nombre);
                op.appendChild(texto);
                $("#temas").append(op)
            });
         }
    }

    function cargaLibros(){
        $.getJSON("libros",
            {idTema:$("#temas").val()},muestra);
    }

    function muestra(data){
      $("#listado").css("display", "block");
      var tabla=
```

```
                 "<table border='1'><tr><th>Titulo</th>";
           tabla+="<th>Autor</th>";
           tabla+="<th>Precio</th></tr>";
           //recorre los libros y genera
           //una fila para cada uno
             $.each(data, function(i,item){
          tabla+="<tr><td>"+item.titulo+"</td>";
          tabla+="<td>"+item.autor+"</td>";
          tabla+="<td>"+item.precio+"</td></tr>";
          if(i==data.length-1){
                         tabla+="</table>";
                         $("#listado").html(tabla);
                     }
                 });
   }
 </script>
</head>

<body>
    <div style="text-align:center;font-size: 10pt">
        <h1>Página de autenticación</h1>
        <br/>
        <form>
            Usuario: <input type="text"
                id="txtusuario" name="txtusuario"/>
            Password: <input type="password"
                 id="txtpassword" name="txtpassword"/>
            <br/>
            <input type="button" value="Login"
                      onclick="login();"/>
        </form>
    </div>
    <hr/>
    <div id="libros"
         style="text-align:center;display:none">

        <h1>Catálogo de libros</h1>
        Selección de temas:
        <form>
            <select id="temas" name="tema">
            </select>
```

```
                <br/><br/>
                <input type="button" onclick="cargaLibros();"
                       value="ver libros"/>
            </form>
        </div>
        <br/>
        <div id="noLogin" style="display:none;
                  text-align:center">
            <h1>El usuario no está registrado</h1>
        </div>
        <center>
            <div id="listado" style="display:none;
                                  width:30%" >

            </div>
        </center>
    </body>
</html>
```

Es de destacar en la página anterior la utilización de la función *each* de jQuery con la que se puede recorrer un array o colección de objetos, en este caso, recorre el array de objetos JSON generado desde el servlet. El elemento *item* indicado en la función, contiene una referencia al elemento apuntado en la iteración actual.

Capítulo 4

AJAX EN APLICACIONES JAVA EE

La plataforma Java EE proporciona el marco de trabajo necesario para la creación de aplicaciones Web en Java, basadas en una arquitectura multicapa.

Utilizando las librerías incluidas en esta plataforma, es posible construir robustas aplicaciones que puedan ser alojadas en una amplia gama de servidores de aplicaciones y que sean capaces de ser ejecutadas desde diferentes tipos de clientes.

Como veremos a lo largo de este capítulo, el modelo de desarrollo AJAX se integra perfectamente en esta arquitectura, contribuyendo a mejorar el rendimiento y la potencia de JAVA EE.

Inicialmente, y tras presentar las características del modelo de aplicación de tres capas, examinaremos la implementación mediante JAVA EE de uno de los esquemas de desarrollo multicapa más eficientes que existen: la arquitectura Modelo Vista Controlador (MVC), analizando el rol jugado por AJAX dentro de este esquema.

A continuación, estudiaremos uno de los frameworks desarrollados para trabajar en arquitectura MVC y que ha sido adoptado como parte de la especificación JAVA EE, se trata de Java Server Faces (JSF). Dentro de que está pensado para trabajar en un esquema Modelo Vista Controlador, JSF está especialmente orientado a mejorar las prestaciones de la vista y a facilitar su desarrollo, por lo que la utilización de AJAX dentro de este framework resultará de enorme interés para un programador.

4.1 ARQUITECTURA DE TRES CAPAS

Una aplicación Web es un programa informático que puede dar servicio simultáneamente a múltiples usuarios que lo ejecutan a través de Internet. Este tipo de aplicaciones se basa en lo que se conoce como una arquitectura de tres capas, donde los diferentes actores y elementos implicados en la misma se encuentran distribuidos en tres bloques o capas (figura 29).

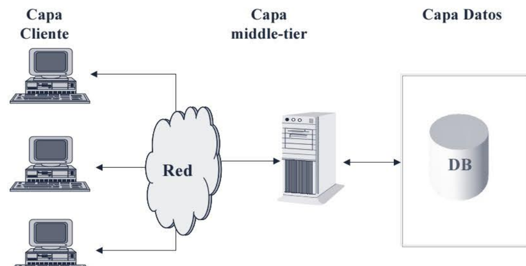


Fig. 29. *Arquitectura de tres capas*

Estas tres capas son:

- Capa cliente.
- Capa intermedia.
- Capa de datos.

4.1.1 Capa cliente

Se trata de la capa con la que interactúa el usuario de la aplicación, normalmente a través de un navegador Web. Realiza principalmente dos funciones: por un lado se encarga de capturar los datos de usuario con los que opera la capa intermedia y enviárselos a ésta, y por otro presentar al usuario los resultados generados por la aplicación.

Las páginas Web, construidas mediante XHTML/CSS, son las encargadas de implementar esta funcionalidad, ayudándose como hemos visto de código JavaScript/AJAX para mejorar la experiencia del usuario con la aplicación.

4.1.2 Capa intermedia

En una arquitectura de tres capas, la capa intermedia está constituida por la aplicación en sí. Ésta se encuentra instalada en una máquina independiente, conocida como servidor, a la que acceden los clientes a través de la red utilizando el protocolo HTTP.

La aplicación de la capa intermedia es ejecutada por un motor de aplicación especial capaz de permitir que una misma instancia de ella pueda dar servicio a múltiples clientes. Además de este motor, los servidores necesitan otro software, conocido como servidor Web, que sirva de interfaz entre la aplicación y el cliente, realizando el diálogo HTTP con éste.

Así pues, de forma resumida podríamos decir que las funciones de la capa intermedia consisten en:

- Recoger los datos enviados desde la capa cliente.
- Procesar la información y, en general, implementar la lógica de aplicación, incluyendo el acceso a los datos.
- Generar las respuestas para el cliente.

Es precisamente en el desarrollo de esta capa intermedia donde JAVA EE encuentra su aplicabilidad, pues las distintas librerías que la componen están orientadas a realizar estas funcionalidades.

4.1.3 Capa de datos

La capa de datos tiene como misión el almacenamiento permanente de la información manejada por la aplicación y la gestión de la seguridad de los mismos.

Para esta tarea se utiliza, en la mayoría de los casos, las llamadas “bases de datos relacionales”. Una base de datos relacional distribuye la información entre diferentes tablas, relacionándolas entre sí a través de un campo común que permita identificar los registros de una tabla que se corresponden con los de otra.

En la figura 30 aparecen algunos de los tipos de bases de datos relacionales más utilizados actualmente.

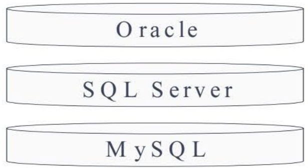


Fig. 30. *Principales tipos de bases de datos relacionales*

Cada una de las tecnologías de desarrollo de capa intermedia proporciona sus propias librerías para poder acceder a una base de datos desde la aplicación. En el caso de JAVA EE esta librería es JDBC.

4.2 ARQUITECTURA MODELO VISTA CONTROLADOR

De cara a afrontar con éxito el desarrollo de la capa intermedia de una aplicación Web, se hace necesario establecer un modelo o esquema que permita estructurar esta capa en una serie de bloques o componentes, de modo que cada uno de estos bloques tenga unas funciones definidas dentro de la aplicación y pueda desarrollarse de manera independiente.

Uno de estos esquemas, y, con toda seguridad, el más utilizado por los desarrolladores de aplicaciones JAVA EE, es la arquitectura Modelo Vista Controlador (MVC), la cual **proporciona una clara separación entre las distintas responsabilidades de la aplicación** (figura 31).

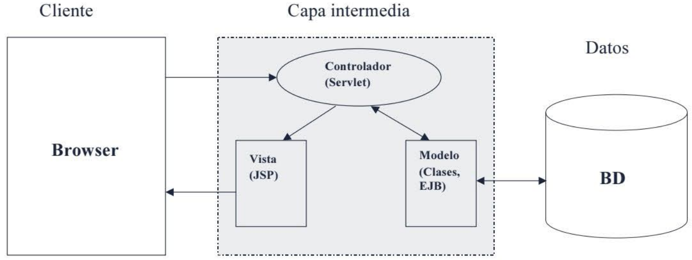


Fig. 31. *Esquema de una aplicación MVC*

Según esta arquitectura, la capa intermedia de una aplicación Web puede dividirse en tres grandes bloques funcionales:

- Controlador
- Vista
- Modelo

4.2.1 El controlador

Se puede decir que el controlador es el "cerebro" de la aplicación. **Todas las peticiones a la capa intermedia que se realicen desde el cliente son dirigidas al controlador**, cuya misión es determinar las acciones a realizar para cada una de estas peticiones e invocar al resto de los componentes de la aplicación (Modelo y Vista) para que realicen las acciones requeridas en cada caso, encargándose también de la coordinación de todo el proceso.

Por ejemplo, en el caso de que una petición requiera enviar como respuesta al cliente determinada información existente en una base de datos, el controlador solicitará los datos necesarios al modelo y, una vez recibidos, se los proporcionará a la vista para que ésta les aplique el formato de presentación correspondiente y le envíe la respuesta al cliente.

En aplicaciones JAVA EE el controlador es implementado mediante un servlet central que, dependiendo de la cantidad de tipos de peticiones que debe gestionar, puede apoyarse en otros servlets auxiliares para procesar cada petición.

4.2.2 La vista

Tal y como se puede deducir de su nombre, la vista es la encargada de generar las respuestas (habitualmente XHTML) que deben ser enviadas al cliente. Cuando esta respuesta tiene que incluir datos proporcionados por el controlador, el código XHTML de la página no será fijo, sino que deberá ser generado de forma dinámica, por lo que su implementación correrá a cargo de una página JSP. Cuando la información que se va a enviar es estática, es decir, no depende de datos extraídos de un almacenamiento externo, podrá ser implementada por una página o documento XHTML.

Es precisamente en la vista donde AJAX juega un papel importante. Tanto las páginas JSP como las XHTML pueden incluir código script de cliente que se comunique en segundo plano con el servidor para obtener datos de él.

En este caso, una vez que la vista ha generado la respuesta y la ha enviado al cliente, **las peticiones realizadas al servidor desde el código AJAX serán dirigidas al controlador**, de este modo, cuando el servlet recibe la petición desde una aplicación AJAX cliente, recuperará los datos necesarios a través del modelo y, en vez de encaminar la petición a la vista para que vuelva a generar la respuesta, lo que supondría una recarga de la página, aplicará a los datos el formato correspondiente (texto plano, XML, JSON) y los enviará directamente a la página cliente que hizo la petición (figura 32).

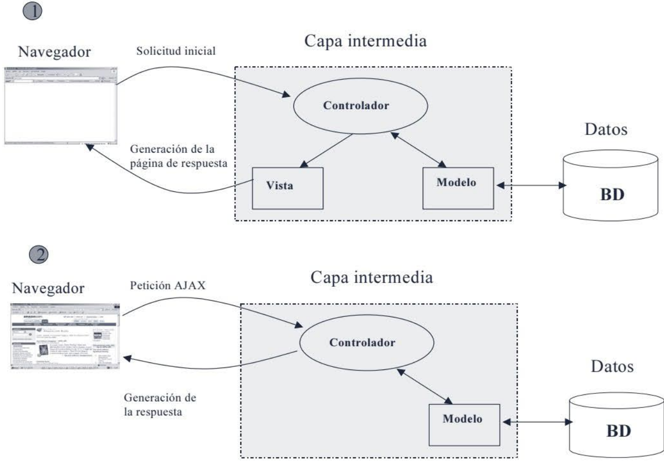


Fig. 32. *Tratamiento de una solicitud AJAX en una aplicación MVC*

4.2.3 El modelo

En la arquitectura MVC, la lógica de negocio de la aplicación, incluyendo el acceso a los datos y su manipulación, está encapsulada dentro del modelo. En una aplicación JAVA EE, el modelo puede ser implementado mediante clases estándar Java o a través de Enterprise JavaBeans.

4.2.4 Aplicación MVC básica

Como ejemplo práctico de utilización de AJAX en una aplicación MVC, vamos a presentar una sencilla aplicación para validación de usuarios contra una

base de datos a los que, en caso de estar registrados, se les mostrará una página donde aparecerá un listado de productos extraídos de la base de datos. Aquellos usuarios no registrados recibirán un mensaje de advertencia en la misma página de validación, con un enlace para acceder a la página de registro (figura 33).

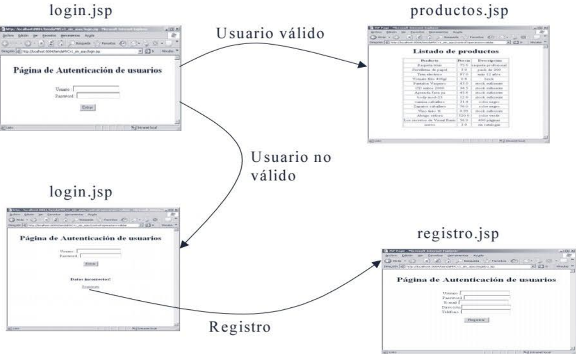


Fig. 33. *Esquema de la aplicación*

Siguiendo un esquema de desarrollo basado en la arquitectura MVC, determinamos que la capa intermedia de esta aplicación recibiría tres tipos de peticiones procedentes de la capa cliente:

- Petición para validación de usuarios.
- Petición para solicitud de registro de usuarios.
- Petición para registrar un usuario.

En el caso de la petición de validación, si el servlet controlador detecta que el usuario no está registrado en la base de datos, reenviará la petición de nuevo a la página login.jsp, produciéndose una recarga de la misma en el navegador. Es aquí donde AJAX puede mejorar el rendimiento de esta aplicación.

Seguidamente, analizaremos las dos posibles soluciones a la aplicación: una sin utilizar AJAX y otra utilizándolo, ambas siguiendo la arquitectura Modelo Vista Controlador.

4.2.4.1 SOLUCIÓN SIN AJAX

Como acabamos de indicar, en este caso cada petición recibida desde la capa cliente, una vez que ha sido tratada por el controlador, tiene como consecuencia el envío de una nueva página de respuesta al cliente.

Veremos a continuación la implementación de cada uno de los bloques funcionales de la aplicación.

4.2.4.1.1 Modelo

El modelo está constituido por una clase Java llamada GestionDatos en la que, a través de una serie de métodos que encapsulan el acceso a los datos, se proporciona al controlador las funcionalidades necesarias para la aplicación.

Los datos utilizados por ésta se encuentran en una base de datos cuya estructura se muestra en la figura 34. Dicha base de datos cuenta con tres tablas: "clientes", donde se almacenan los datos de los usuarios registrados; "secciones", que contiene los nombres de las secciones en las que se organizan los productos almacenados, y "productos", que es la tabla que contiene los datos de todos los productos.

Clientes

Campo	Tipo
pw	cadena
usuario	cadena
email	cadena
direccion	cadena
telefono	cadena

Productos

Campo	Tipo
id	autonumérico
id_secc	numérico entero
producto	cadena
precio	numérico decimal
descripcion	cadena

Secciones

Campo	Tipo
id	autonumérico
nombre	cadena

Fig. 34. *Tablas de la base de datos de ejemplo*

El siguiente listado corresponde al código de la clase GestionDatos:

```
package modelo;
import java.sql.*;
import beans.*;
import java.util.*;
```

```
public class GestionDatos {
    private String cadenaCon;
    public GestionDatos(String c) {
        cadenaCon=c;
    }
    //método común para obtener una conexión
    //con la base de datos
    private Connection getConection(){
        Connection cn=null;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            cn=DriverManager.getConnection(cadenaCon);
        }
        catch(Exception e){e.printStackTrace();}
        finally{
            return cn;
        }
    }
    public boolean validar(String user, String password){
        boolean result=false;
        try{
         Connection cn=this.getConection();
         //instrucción SQL para obtener los datos
         //del usuario indicado
         String query = "select * from clientes where pw ='"+
                    password+"' and usuario='"+user+"'";
         Statement st =cn.createStatement();
         ResultSet rs = st.executeQuery(query);
         result= rs.next();
         cn.close();
        }
         catch(Exception e){
             e.printStackTrace();
         }
        finally{
            return result;
        }
    }
    public void registrar(Usuario us){
         //genera la instrucción SQL de inserción a partir
         //de los datos almacenados en el JavaBean Usuario
```

```
            String query = "INSERT INTO clientes values('"+
                us.getPassword()+"','"+us.getUser()+"','"+
                us.getEmail()+"','"+us.getDireccion()+"','"+
                      us.getTelefono()+"')";
            try{
               Connection cn=this.getConection();
               Statement st =cn.createStatement();
               st.execute(query);
               st.close();
            }
            catch(Exception e){e.printStackTrace();}
       }
       public ArrayList<Producto> getProductos(){
           //colección del tipo de objetos que se
           //van a devolver
           ArrayList<Producto> productos=
                      new ArrayList<Producto>();
           try{
               String query = "select * from productos";
               Connection cn=this.getConection();
               Statement st =cn.createStatement();
               ResultSet rs = st.executeQuery(query);
               //para cada fila, construye un objeto Producto
               //y lo añade a la colección
               while(rs.next()){
                 Producto p=new Producto();
                 p.setProducto(rs.getString("producto"));
                 p.setPrecio(rs.getDouble("precio"));
                 p.setDescripcion(rs.getString("descripcion"));
                 productos.add(p);
               }
           }
           catch(Exception e){e.printStackTrace();}
           finally{
               return productos;
           }
       }
   }
```

Esta clase se apoya en dos clases de tipo JavaBean llamadas Usuario y Producto que encapsulan los datos de un usuario y un producto de la base de datos, respectivamente.

El código de estas clases se muestra a continuación:

```
package beans;
//clase Usuario

public class Usuario{
    private String user;
    private String password;
    private String email;
    private String direccion;
    private String telefono;

    public String getUser() {
        return user;
    }
    public void setUser(String user) {
        this.user = user;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getDireccion() {
        return direccion;
    }
    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }
    public String getTelefono() {
```

```
          String query = "INSERT INTO clientes values('"+
              us.getPassword()+"','"+us.getUser()+"','"+
              us.getEmail()+"','"+us.getDireccion()+"','"+
                    us.getTelefono()+"')";
          try{
             Connection cn=this.getConection();
             Statement st =cn.createStatement();
             st.execute(query);
             st.close();
          }
          catch(Exception e){e.printStackTrace();}
     }
     public ArrayList<Producto> getProductos(){
         //colección del tipo de objetos que se
         //van a devolver
         ArrayList<Producto> productos=
                    new ArrayList<Producto>();
         try{
             String query = "select * from productos";
             Connection cn=this.getConection();
             Statement st =cn.createStatement();
             ResultSet rs = st.executeQuery(query);
             //para cada fila, construye un objeto Producto
             //y lo añade a la colección
             while(rs.next()){
               Producto p=new Producto();
               p.setProducto(rs.getString("producto"));
               p.setPrecio(rs.getDouble("precio"));
               p.setDescripcion(rs.getString("descripcion"));
               productos.add(p);
             }
         }
         catch(Exception e){e.printStackTrace();}
         finally{
             return productos;
         }
     }
}
```

Esta clase se apoya en dos clases de tipo JavaBean llamadas Usuario y Producto que encapsulan los datos de un usuario y un producto de la base de datos, respectivamente.

El código de estas clases se muestra a continuación:

```
package beans;
//clase Usuario

public class Usuario{
    private String user;
    private String password;
    private String email;
    private String direccion;
    private String telefono;

    public String getUser() {
        return user;
    }
    public void setUser(String user) {
        this.user = user;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getDireccion() {
        return direccion;
    }
    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }
    public String getTelefono() {
```

```
            return telefono;
        }
        public void setTelefono(String telefono) {
            this.telefono = telefono;
        }
    }

    //clase Producto
    public class Producto{
        private int id;
        private String producto;
        private double precio;
        private String descripcion;

        public int getId() {
            return id;
        }
        public void setId(int id) {
            this.id = id;
        }
        public String getProducto() {
            return producto;
        }
        public void setProducto(String producto) {
            this.producto = producto;
        }
        public double getPrecio() {
            return precio;
        }
        public void setPrecio(double precio) {
            this.precio = precio;
        }
        public String getDescripcion() {
            return descripcion;
        }
        public void setDescripcion(String descripcion) {
            this.descripcion = descripcion;
        }
    }
```

4.2.4.1.2 Controlador

El controlador está implementado, como ya se ha indicado, mediante un servlet el cual, a través de una parámetro llamado "operacion" que es recibido en la petición, determinará la operación a realizar en cada momento.

El siguiente listado muestra el código del servlet controlador:

```
package servlets;

import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
import modelo.GestionDatos;
import beans.*;

public class Control extends HttpServlet {
    protected void service(HttpServletRequest request,
                              HttpServletResponse
                          response)
    throws ServletException, IOException {
        String op=request.getParameter("operacion");
        ServletContext ctx = this.getServletContext();
        //la cadena de conexión con la base de datos
        //se encuentra definida como parámetro en el
        //fichero de configuración web.xml
        String url = ctx.getInitParameter("url");
        GestionDatos dts=new GestionDatos(url);
        RequestDispatcher rd;
        //validación de usuarios
        if(op.equals("validar")){
            String user=request.getParameter("user");
            String password=request.getParameter("pass");
            if(dts.validar(user,password)){
                request.setAttribute("productos",
                                       dts.getProductos());
                rd=request.getRequestDispatcher(
                                       "/productos.jsp");
            }
            else{
                //se vuelve a cargar la página de login
```

```
                request.setAttribute("login",false);
                rd=request.getRequestDispatcher(
                                       "/login.jsp");
            }
            rd.forward(request,response);
        }
        //registro de un usuario
        if(op.equals("registrar")){
            Usuario us=
              (Usuario)request.getAttribute("datos");
            dts.registrar(us);
            response.sendRedirect("login.jsp");
        }
        //solicitud de registro de un usuario
        if(op.equals("solicitud")){
            response.sendRedirect("registro.jsp");
        }
    }
}
```

4.2.4.1.3 Vista

La vista de la aplicación está implementada mediante tres páginas JSP: login.jsp, producto.jsp y registro.jsp.

login.jsp

Ésta es la página de inicio donde el usuario introduce los credenciales para validarse en la base de datos, siendo recargada de nuevo en el navegador si el usuario no se encuentra registrado en la base de datos. En esta página es donde encontraremos los mayores cambios cuando veamos la versión equivalente en AJAX. El siguiente listado muestra el código de esta página:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                           Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
                                  charset=UTF-8">
```

```
    </head>
<body>
    <center>
        <br/>
        <h1>Página de Autenticación de usuarios</h1>
        <br/>
        <form action="control?operacion=validar"
                                    method="post">
            Usuario  : <input type="text" name="user"
                     value="" size="20" /><br>
            Password : <input type="password" name="pass"
                     value="" size="20" /><br>
            <br>
            <input type="submit" value="Entrar"
                              name="entrar" /><br><br>
            <%//si el usuario no está registrado se muestra
              //el enlace a la página de registro
              if(request.getAttribute("login")!=null&&
                    !(Boolean)request.getAttribute("login"))
             {%>
                <br/><br/>
                <h3>Datos incorrectos!</h3>
                <a href="control?operacion=solicitud">
                    Registrate</a>
            <%}%>
        </form>
    </center>
  </body>
</html>
```

producto.jsp

Es la página que muestra los datos de los productos obtenidos a partir de la base de datos, una vez que el usuario se ha validado. La información mostrada por esta página es obtenida por el servlet a partir del método *getProductos()* proporcionado por el modelo:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@page import="java.util.*,beans.*"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
```

```
                    Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
                                             charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
     <center>
         <h1>Listado de productos</h1>
            <table border="1">
                <tr>
                    <th>Producto</th>
                    <th>Precio</th>
                    <th>Descripción</th>
                </tr>
                <%ArrayList prods=
                    (ArrayList)request.getAttribute(
                                             "productos");
                  //genera la tabla de productos
                  for(int i=0;i<prods.size();i++){
                    Producto p=(Producto)prods.get(i);%>
                    <tr>
                        <td><%=p.getProducto()%></td>
                        <td><%=p.getPrecio()%></td>
                        <td><%=p.getDescripcion()%> 
                          </td>
                    </tr>
               <% }%>
            </table>
        </center>
    </body>
</html>
```

registro.jsp

Se trata de la página utilizada por el usuario para registrarse en la base de datos. A ella será enviado el usuario por el controlador cuando se active el enlace "Regístrate" dentro de la página login.jsp. Su código se muestra a continuación:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                                  Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
                                         charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <center>
             <br/>
            <h1>Página de Autenticación de usuarios</h1>
            <form action="registro.jsp" method="post">
              Usuario  :<input type="text" name="user"
                     value="" size="20" /><br>
              Password :<input type="password"
                    name="password" value="" size="20" /><br>
              E-mail   :<input type="text" name="email"
                    value="" size="20" /><br>
              Dirección:<input type="text" name="direccion"
                                 value=""size="20"/><br/>
              Teléfono :<input type="text" name="telefono"
                          value="" size="20" /><br/><br/>
              <input type="submit" value="Registrar" />
            </form>
        </center>
    <%if(request.getParameter("user")!=null){%>
        <!--crea un JavaBean Usuario y lo rellena con los
         datos procedentes del formulario XHTML, para después
         reenviar la petición al controlador-->
       <jsp:useBean id="datos" class="beans.Usuario"
                                         scope="request"/>
        <jsp:setProperty name="datos" property="*"/>
        <jsp:forward page="control?operacion=registrar"/>
    <%}%>
    </body>
</html>
```

4.2.4.2 SOLUCIÓN CON AJAX

La solución AJAX en esta aplicación consistirá en evitar la recarga de la página de login, en caso de que el usuario no esté registrado en la base de datos.

Para ello, cuando el usuario pulse el botón "Entrar" en la página login.jsp, en vez de realizar un *submit* de la página al controlador, se lanzará una petición asíncrona utilizando el objeto XMLHttpRequest. En función de la respuesta recibida, se optará por dejar al usuario en la página mostrándole además el enlace de registro, o por realizar una nueva petición al controlador, esta vez en modo estándar sin utilizar AJAX, para que se envíe al usuario la lista de productos.

Analicemos los cambios a realizar en los distintos bloques de la aplicación.

4.2.4.2.1 Modelo

En este caso, el modelo no sufrirá ningún tipo de modificación, manteniendo la misma clase GestionDatos con la misma interfaz de métodos que en el caso anterior.

4.2.4.2.2 Controlador

En cuanto al servlet control, la diferencia respecto a la versión sin AJAX la tenemos en la operación *validar*, puesto que ahora no se trata de redirigir al usuario a ninguna página de la aplicación, sino de devolver un resultado como respuesta, que será la cadena de texto "true" si el usuario está registrado o "false" si no lo está.

Así mismo, se deberá incluir una nueva operación para el caso de que desde la propia página login.jsp se quiera redirigir al usuario a la página de productos, una vez que desde el código AJAX cliente se determine que el usuario está validado.

El código completo del servlet controlador para esta versión se muestra en el siguiente listado, indicando en fondo sombreado las modificaciones respecto a la anterior versión:

```
package servlets;

import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
import modelo.GestionDatos;
```

```
import beans.*;

public class Control extends HttpServlet {
    protected void service(HttpServletRequest request,
                                   HttpServletResponse
                             response)
    throws ServletException, IOException {
        String op=request.getParameter("operacion");
        ServletContext ctx = this.getServletContext();
        String url = ctx.getInitParameter("url");
        GestionDatos dts=new GestionDatos(url);
        PrintWriter out = response.getWriter();
        RequestDispatcher rd;
        //operación de validar usuarios
        if(op.equals("validar")){
            String user=request.getParameter("user");
            String password=request.getParameter("pass");
            response.setContentType("text/plain;charset=
                                                 UTF-8");
            if(dts.validar(user,password)){
                out.println("true");
            }
            else{
                out.println("false");
            }
        }
        //operación que muestra la lista de productos
        if(op.equals("verproductos")){
            request.setAttribute("productos",
                                  dts.getProductos());
            rd=request.getRequestDispatcher(
                                  "/productos.jsp");
            rd.forward(request,response);
        }
        //operación de registro de usuarios
        if(op.equals("registrar")){
            Usuario us=
                    (Usuario)request.getAttribute("datos");
            dts.registrar(us);
            response.sendRedirect("login.jsp");
        }
```

```
            if(op.equals("solicitud")){
                response.sendRedirect("registro.jsp");
            }
        }
    }
```

4.2.4.2.3 Vista

Evidentemente, la vista que sufre variaciones en esta nueva versión es login.jsp, ya que en ella se incluirá todo el código JavaScript AJAX de la aplicación. Además, el botón *submit* de "Enviar" de la página será sustituido por un botón de pulsación estándar, capturándose el evento *onclick* para realizar la llamada a la función AJAX.

Otro de los cambios que tendrán lugar en esta página consistirá en sustituir el scriplet de código de servidor Java que se encargaba de determinar la necesidad de mostrar o no el enlace de registro por una capa HTML, que será mostrada u ocultada por el script de cliente en función de la respuesta recibida.

A continuación se muestra el código XHTML de la página, indicando en fondo sombreado los cambios realizados en esta nueva versión:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                                  Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
                             charset=UTF-8">
        <script language="javascript">
              :
        </script>
    </head>
    <body>
    <center>
        <form action="control?operacion=validar"
                    method="post">
            Usuario  : <input type="text" name="user"
                    value="" size="20" /><br>
            Password : <input type="password" name="pass"
```

```
                              value="" size="20" /><br>
              <br>
              <input type="button" value="Entrar"
                      onclick="validaUsuario();" /><br/><br/>
              <!--capa con enlace de registro
                          inicialmente oculta-->
             <div id="registro" style="display:none">
                  <br/><br/>
                  <h3>Datos incorrectos!</h3>
                  <a href="control?operacion=solicitud">
                                    Registrate
                  </a>
              </div>
          </form>
      </center>
    </body>
  </html>
```

En cuanto al código AJAX de la página, se muestra en el siguiente listado:

```
 <script language="javascript">
        var xhr;
        function validaUsuario(){
            if(window.ActiveXObject){
                xhr=new ActiveXObject("Microsoft.XMLHttp");
            }
            else if((window.XMLHttpRequest) ||
                          (typeof
                  XMLHttpRequest)!=undefined){
                xhr=new XMLHttpRequest();
            }
            else{
                alert("Su navegador no tiene
                                       soporte para AJAX");
                return;
            }
            enviaPeticion();
        }
        function enviaPeticion(){
            xhr.open("POST",document.forms[0].action,true);
            xhr.onreadystatechange=procesaResultado;
```

```
            //definición del tipo de contenido del cuerpo
            //para formularios HTML
            xhr.setRequestHeader("Content-Type",
                    "application/x-www-form-urlencoded");
            var datos=obtenerDatos();
            //se envían los datos en el cuerpo
            //de la petición HTTP
            xhr.send(datos);
        }
        function mostrarProductos(){
            //lanza la petición sin utilizar AJAX
            //para que se genere la vista con los productos
            window.document.location=
                    "control?operacion=verproductos";
        }
        function obtenerDatos(){
            //genera una cadena con los datos de los
            //controles del formulario para
            //enviarlos en la petición
            var controles=document.forms[0].elements;
            var datos=new Array();
            var cad="";
            for(var i=0;i<controles.length;i++){
                cad=encodeURIComponent(
                          controles[i].name)+"=";
                cad+=encodeURIComponent(controles[i].value);
                datos.push(cad);
            }
            //se forma la cadena con el método join() del
            //array para evitar múltiples concatenaciones,
            //mejorando el rendimiento de la aplicación
            cad=datos.join("&");
            return cad;
        }
        function procesaResultado(){
            if(xhr.readyState==4){
                var result=eval(xhr.responseText);
                //si el usuario no está registrado se muestra
                //la capa con el enlace de registro
```

```
                if(!result){
                    document.getElementById("registro").
                            style.display="block";
                }

                else{
                    mostrarProductos();
                }
            }
        }
    </script>
```

4.2.5 Implementación de un carrito de compra

Si hay un tipo de funcionalidad dentro de una aplicación Web donde AJAX pueda desplegar todo su potencial, ésa es la del carrito de compra.

Este tipo de módulos se caracterizan por disponer de una interfaz gráfica con alto nivel de interacción con el usuario, en donde la información visualizada debe ser constantemente actualizada ante las acciones de éste. Así, cada vez que el usuario decide comprar un producto, la página debe actualizar la lista del carrito con la información del nuevo producto elegido, de la misma forma, cuando se decide eliminar un producto del carrito, deberá desaparecer su información del mismo.

Estas acciones de compra y eliminación de productos implican solicitudes al servidor para realizar algún tipo de acción sobre la aplicación, acción que conllevará una actualización de la vista. Dada la elevada frecuencia con la que se realizarán estas acciones y la necesidad de que la actualización de la vista se produzca prácticamente en tiempo real, no resulta viable una solución tradicional basada en la petición-recarga de la página, siendo AJAX la opción a utilizar.

El ejemplo que vamos a exponer a continuación consiste en una mejora de la página de productos (productos.jsp) utilizada en el ejemplo de aplicación MVC básica analizado anteriormente.

La imagen mostrada en la figura 35 corresponde al nuevo aspecto de la página de productos.

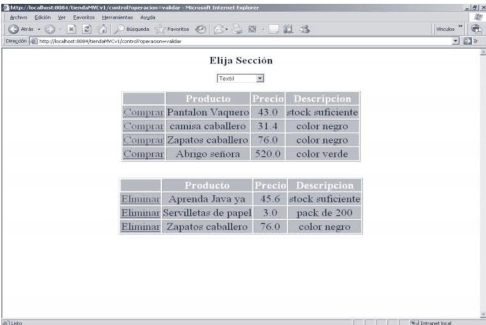


Fig. 35. *Nuevo aspecto de la página de productos*

Como se puede apreciar, la página contará con una lista desplegable en la parte superior de la misma donde el usuario podrá elegir la sección cuyos productos el usuario quiera visualizar. Una vez elegida la sección, se mostrará la tabla de productos correspondientes, incluyendo una nueva columna con el enlace "comprar" que realizará la operación de agregar el producto al carrito de la compra.

El carrito de la compra también será mostrado en esta misma página, incluyendo cada producto del mismo el enlace "eliminar" que permitirá descartar el producto con su correspondiente desaparición del carrito.

4.2.5.1 EL MODELO

Como en la versión anterior, el modelo será implementado mediante la clase GestionDatos.

Se mantienen intactos los métodos *registrar()* y *validar()*, ya que esta funcionalidad no sufre alteración alguna. En cuanto al método *getProductos()*, su implementación será diferente a la realizada en la versión anterior, pues, en vez de construirse una tabla HTML a partir de una colección de objetos Producto, la nueva aplicación deberá enviar al cliente los datos de los productos asociados a una determinada sección que han sido solicitados a través de una petición AJAX. Para ello, el modelo genera una cadena de caracteres con el documento XML que contiene los datos de los productos:

```
<productos>
       <producto id="identificador"
                   producto="nombre producto"
```

```
                        precio="precio producto"
                        descripcion="descripción producto"/>
            :
</productos>
```

Así mismo, se incorporan nuevos métodos a la clase para implementar las nuevas funcionalidades, como la gestión de la cesta o carrito de la compra y la generación de la lista de secciones.

A continuación, se muestra el código de la clase indicándose los cambios respecto a la versión anterior:

```
package modelo;

import java.sql.*;
import beans.*;
import java.util.*;
public class GestionDatos {
    private String cadenaCon;
    public GestionDatos(String c) {
        cadenaCon=c;
    }
    private Connection getConection(){
        Connection cn=null;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            cn=DriverManager.getConnection(cadenaCon);
        }
        catch(Exception e){e.printStackTrace();}
        finally{
            return cn;
        }
    }
    public boolean validar(String user, String password){
        boolean results=false;
        try{
         Connection cn=this.getConection();
         String query = "select * from clientes where pw
              ='"+password+"' and usuario='"+user+"'";
         Statement st =cn.createStatement();
         ResultSet res = st.executeQuery(query);
         results= res.next();
```

```
         cn.close();
        }
         catch(Exception e){
             e.printStackTrace();
         }
        finally{
            return results;
        }
    }
    public void registrar(Usuario us){
         //registra un usuario a partir del objeto
         String query = " INSERT INTO clientes values('"+
             us.getPassword()+"','"+us.getUser()+
             "','"+us.getEmail()+"','"+
             us.getDireccion()+"','"+us.getTelefono()+"')";
         try{
            Connection cn=this.getConection();
            Statement st =cn.createStatement();
            st.execute(query);
            st.close();
         }
         catch(Exception e){e.printStackTrace();}
    }
    public ArrayList<Seccion> getSecciones(){
         //devuelve una colección de objetos Seccion
         //con los datos de todas las secciones de la tabla
        ArrayList<Seccion> secciones=
                    new ArrayList<Seccion>();
        try{
            String query = "select * from secciones";
            Connection cn=this.getConection();
            Statement st =cn.createStatement();
            ResultSet rs = st.executeQuery(query);
            while(rs.next()){
              Seccion p=new Seccion();
              p.setId(rs.getInt("id"));
              p.setNombre(rs.getString("nombre"));
              secciones.add(p);
            }
        }
```

```
        catch(Exception e){e.printStackTrace();}
        finally{
            return secciones;
        }
    }
    private Producto getProducto(int codigo){
        //devuelve un objeto con todos los
        //datos del producto a partir de su identificador
        Producto pr=null;
        try{
            String query = "select * from productos
                                            where id="+codigo;
            Connection cn=this.getConection();
            Statement st =cn.createStatement();
            ResultSet rs = st.executeQuery(query);
            if(rs.next()){
                pr=new Producto();
                pr.setId(rs.getInt("id"));
                pr.setProducto(rs.getString("producto"));
                pr.setPrecio(rs.getDouble("precio"));
                pr.setDescripcion(
                    rs.getString("descripcion"));
            }
        }
        catch(Exception e){e.printStackTrace();}
        finally{
            return pr;
        }
    }
    public String getProductos(int idseccion){
        StringBuilder sb=null;
        try{
            String query = "select * from productos
                                where id_secc="+idseccion;
            Connection cn=this.getConection();
            Statement st =cn.createStatement();
            ResultSet rs = st.executeQuery(query);
            //genera un documento XML con los datos de los
            //productos de la seción elegida
            sb=new StringBuilder("<?xml version='1.0'?>");
            sb.append("<productos>");
```

```
            while(rs.next()){
              sb.append("<producto id='"+rs.getInt("id")+
                                            "' ");
              sb.append("producto='"+
                    rs.getString("producto")+"' ");
              sb.append("precio='"+rs.getDouble("precio")+
                          "' ");
              sb.append("descripcion='"+
                    rs.getString("descripcion")+"'/>");
            }
            sb.append("</productos>");
        }
        catch(Exception e){e.printStackTrace();}
        finally{
            return sb.toString();
        }
    }
    public void agregaCesta(ArrayList<Producto> cesta,
                                          int codigo){
        //añade un nuevo producto a la cesta a partir
        //del código de su código
        cesta.add(this.getProducto(codigo));
    }
    public void quitardeCesta(ArrayList<Producto> cesta,
                                          int pos){
        //elimina de la cesta el producto cuya posición
        //se indica
        cesta.remove(pos);
    }
    public String formatCesta(ArrayList<Producto> cesta){
        //genera un documento XML con los datos de la cesta
        StringBuilder sb=new StringBuilder("
                          <?xml version='1.0'?>");
        sb.append("<productos>");
        for(Producto pr:cesta){
              sb.append("<producto id='"+pr.getId()+"' ");
              sb.append("producto='"+pr.getProducto()+"' ");
              sb.append("precio='"+pr.getPrecio()+"' ");
              sb.append("descripcion='"+
                    pr.getDescripcion()+"'/>");
        }
```

```
            sb.append("</productos>");
            return sb.toString();
        }
    }
```

4.2.5.2 EL CONTROLADOR

El controlador debe ser capaz de gestionar las nuevas peticiones que le llegan desde la capa cliente. A continuación se muestra el listado del servlet control, indicándose los cambios respecto a la versión anterior:

```
package servlets;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import modelo.GestionDatos;
import beans.*;

public class Control extends HttpServlet {
    protected void service(HttpServletRequest request,
                           HttpServletResponse response)
    throws ServletException, IOException {
        String op=request.getParameter("operacion");
        ServletContext ctx = this.getServletContext();
        PrintWriter out = response.getWriter();
        String url = ctx.getInitParameter("url");
        GestionDatos dts=new GestionDatos(url);
        if(op.equals("validar")){
            //solicitud para validar un usuario
            String user=request.getParameter("user");
            String password=request.getParameter("pass");
            if(dts.validar(user,password)){
                out.println("true");
            }
            else{
                out.println("false");
            }
        }
```

```
if(op.equals("registrar")){
    //solicitud para registro de usuario
    Usuario us=
     (Usuario)request.getAttribute("datos");
    dts.registrar(us);
    response.sendRedirect("index.html");
}
if(op.equals("secciones")){
    //solicitud inicial para mostrar la lista
    //de secciones
    RequestDispatcher rd;
    request.setAttribute(
            "secciones",dts.getSecciones());
    rd=request.getRequestDispatcher(
            "/productos.jsp");
    rd.forward(request,response);
}
if(op.equals("solicitud")){
    //solicitud para mostrar la página de registro
    response.sendRedirect("registro.jsp");
}
if(op.equals("productos")){
    //solicitud para mostrar la lista de productos de
    //la sección elegida
    response.setContentType("text/xml;
                              charset=UTF-8");
    int id=Integer.parseInt(
            request.getParameter("idseccion"));
    out.println(dts.getProductos(id));
}
if(op.equals("agregarcesta")){
    //solicitud para añadir un producto al carrito
    int idproducto=Integer.parseInt(
            request.getParameter("idproducto"));
    ArrayList<Producto> cesta;
    HttpSession sesion=request.getSession();
     //obtiene la cesta. Si no existe la crea y la
     //almacena en la variable de sesión
    cesta=(ArrayList)sesion.getAttribute("cesta");
    if(cesta==null){
        cesta=new ArrayList<Producto>();
```

```
                    sesion.setAttribute("cesta",cesta);
                }
                dts.agregaCesta((ArrayList)cesta,idproducto);
                response.setContentType("text/xml;
                                    charset=UTF-8");
                out.println(dts.formatCesta((ArrayList)cesta));
            }
            if(op.equals("quitardecesta")){
                //solicitud para eliminar un producto del carrito
                int pos=Integer.parseInt(
                             request.getParameter("pos"));
                ArrayList<Producto> cesta;
                HttpSession sesion=request.getSession();
                cesta=(ArrayList)sesion.getAttribute("cesta");
                dts.quitardeCesta((ArrayList)cesta,pos);
                response.setContentType("text/xml;
                                    charset=UTF-8");
                out.println(dts.formatCesta((ArrayList)cesta));
            }
        }
    }
```

4.2.5.3 LA VISTA

De las vistas de la aplicación, solamente productos.jsp es la que tiene que ser modificada para soportar la nueva funcionalidad. Los cambios son bastante sustanciales, ya que debe incluir todo el código AJAX necesario para realizar la solicitud que permita obtener la tabla de productos cuando se elige una determinada sección, así como las solicitudes correspondientes a la compra de productos y la eliminación de los mismos del carrito.

A continuación, mostramos el código completo de productos.jsp:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@page import="java.util.*,beans.*" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                           Transitional//EN"
                     "http://www.w3.org/TR/html4/loose.dtd">

<html>
```

```
<head>
    <meta http-equiv="Content-Type"
                content="text/html; charset=UTF-8">
 <script language="javascript">
    var xhr;
    function buscarProductos(){
        crearObjeto();
        enviapeticion();
    }
    function crearObjeto(){
        //crea el objeto XMLHttpRequest
        if(window.ActiveXObject){
            xhr=new ActiveXObject("Microsoft.XMLHttp");
        }
        else if((window.XMLHttpRequest) ||
                (typeof XMLHttpRequest)!=undefined){
            xhr=new XMLHttpRequest();
        }
        else{
            alert("Su navegador no tiene
                      soporte para AJAX");
            return;
        }
    }
    function enviapeticion(){
        //lanza la petición para recuperar los productos
        //de la sección elegida
        var lista=document.getElementById("seccion")
        var s=lista.options[lista.selectedIndex].value;
        xhr.open("GET",
         "control?operacion=productos&idseccion="+s,
         true);
        xhr.onreadystatechange=procesaDatos;
        xhr.send(null);
    }
    function procesaDatos(){
        if(xhr.readyState==4){
            var textoHTML="<table border='1'>";
            textoHTML+="<tr>";
            textoHTML+="<th></th><th>Producto</th>";
            textoHTML+=
```

```
                "<th>Precio</th><th>Descripcion</th>";
            textoHTML+="</tr>";
            //construye la tabla HTML con los datos
            //de los productos
            var resp=xhr.responseXML;
            var productos=resp.getElementsByTagName(
                                   "producto");
            for(var i=0;i<productos.length;i++){
                var compra="control?operacion=
                            agregarcesta&idproducto=";
                var idproducto=productos.
                            item(i).getAttribute("id");
                textoHTML+="<tr><td><a href='javascript:
                      void(comprar("+idproducto+"))'>
                      Comprar</a></td>";
                textoHTML+="<td>"+productos.item(i).
                      getAttribute("producto")+"</td>";
                textoHTML+="<td>"+productos.item(i).
                      getAttribute("precio")+"</td>";
                textoHTML+="<td>"+productos.item(i).
                      getAttribute("descripcion")+
                                   "</td>";
            }
            textoHTML+="</tr></table>";
            document.getElementById(
                "productos").innerHTML=textoHTML;
        }
    }
    function comprar(idproducto){
        //lanza la petición para añadir el
        //producto al carrito
        crearObjeto();
        xhr.open("GET","control?operacion=
         agregarcesta&idproducto="+idproducto,true);
        xhr.onreadystatechange=procesaCesta;
        xhr.send(null);
    }
    function quitar(pos){
        //lanza la petición para eliminar
        //el objeto del carrito
        crearObjeto();
```

```
            xhr.open("GET","control?operacion=
                  quitardecesta&pos="+pos,true);
            xhr.onreadystatechange=procesaCesta;
            xhr.send(null);
        }
        function procesaCesta(){
             if(xhr.readyState==4){
               var textoHTML="<table border='1'>";
               textoHTML+="<tr>";
               textoHTML+="<th></th><th>Producto</th>";
               textoHTML+=
                   "<th>Precio</th><th>Descripcion</th>";
               textoHTML+="</tr>";
               //construye la tabla con el carrito de
               //de la compra
               var resp=xhr.responseXML;
               var productos=
                   resp.getElementsByTagName("producto");
               for(var i=0;i<productos.length;i++){
                   textoHTML+="<tr><td><a href='javascript:
                   void(quitar("+i+"))'>Eliminar</a></td>";
                   textoHTML+="<td>"+
                          productos.item(i).getAttribute(
                          "producto")+"</td>";
                   textoHTML+="<td>"+productos.
                          item(i).getAttribute("precio")+
                          "</td>";
                   textoHTML+="<td>"+
                          productos.item(i).getAttribute(
                          "descripcion")+"</td>";
               }
               textoHTML+="</tr></table>";
               document.getElementById("cesta").
                                 innerHTML=textoHTML;
            }
        }
    </script>
        <style>
            td{font-size:18pt;background-color:aqua}
            th{font-size:18pt;
                  background-color:orange;color:white}
```

```
        </style>
    </head>
    <body>
        <center>
            <h2>Elija Sección</h2>
            <select id="seccion" onchange=
                          "buscarProductos();">
            <%ArrayList seccs=
               (ArrayList)request.getAttribute("secciones");
               for(int i=0;i<seccs.size();i++){
                   Seccion p=(Seccion)seccs.get(i);%>
                   <option value="<%=p.getId()%>">
                       <%=p.getNombre()%>
                   </option>
               <% }%>
            </select>
            <br/>
            <br/>
            <div id="productos"></div>
            <br/>
            <br/>
            <div id="cesta"></div>
        </center>
    </body>
</html>
```

Como se puede observar, se hace uso directamente del objeto XMLHttpRequest dentro del código AJAX de la aplicación. Dejamos como ejercicio para el lector la realización de una nueva versión del código AJAX cliente en la que se utilice la clase ObjetoAJAX creada en el capítulo anterior.

4.3 JAVASERVER FACES

Java Server Faces (JSF) constituye un marco de trabajo para el desarrollo de aplicaciones Web en Java, basado en la arquitectura Modelo Vista Controlador.

A diferencia de otros framework MVC, JSF ha sido adoptado como parte de las especificaciones de Java Sun para la creación de aplicaciones Web, por lo que el conjunto de utilidades que proporciona se encuentran incluidas dentro de las librerías JAVA EE.

La principal característica de JSF es que, siendo un framework orientado a la creación de aplicaciones Web de tres capas, su filosofía de trabajo consiste en hacer que el desarrollo de este tipo de aplicaciones sea similar al de una aplicación de escritorio, para lo cual proporciona un amplio conjunto de componentes gráficos avanzados, más allá de los clásicos controles HTML, y basa la programación del código en la captura de eventos sobre la interfaz de usuario.

Si a lo anterior añadimos la posibilidad de utilizar AJAX dentro del código de cliente, concluimos que la creación de una aplicación Web basada en JSF resulta similar, tanto en rendimiento como en metodología de desarrollo, a una aplicación Java de escritorio basada en swing.

4.3.1 Componentes de la tecnología JSF

Los componentes de JSF están orientados a proporcionar soporte para el desarrollo del controlador y la vista de la aplicación. Entre los elementos principales destacamos:

- **El API JSF**. Lo constituyen un conjunto de clases e interfaces cuyas principales funcionalidades son:
  - Representación de los distintos componentes gráficos a utilizar.
  - Creación de JavaBeans.
  - Manejadores de eventos.
  - Control de navegación (servlet FacesServlet).
- **Librerías de acciones JSP personalizadas**. JSF proporciona dos librerías de acciones que permiten la construcción de los diferentes componentes gráficos incluidos en el API JSF, dentro de una página JSP.
- **Archivos de configuración**. Entre otros, las aplicaciones JSF incluyen un archivo de configuración en el que se definen los criterios de navegación entre páginas, evitando realizar estas operaciones desde código.

4.3.2 Arquitectura de una aplicación JSF

En la figura 36 se muestra la estructura de una aplicación JSF, organizada según los tres bloques de la arquitectura MVC.

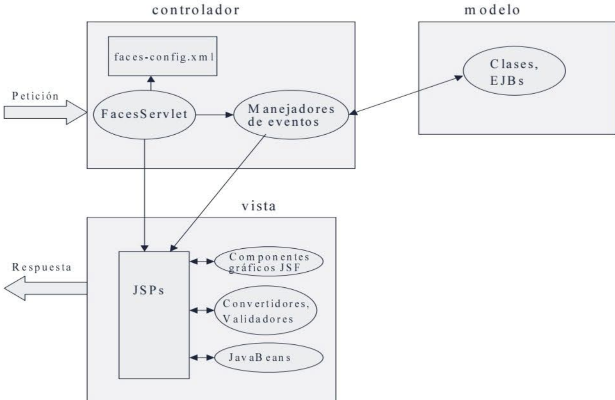


Fig. 36. *Arquitectura MCV con JavaServer Faces*

4.3.2.1 EL CONTROLADOR

Fundamentalmente, el controlador JSF está formado por:

- **Un servlet** que hace de Front Controller llamado **FacesServlet** . Este servlet recibe todas las peticiones procedentes de la capa cliente y, sirviéndose del resto de los componentes del controlador, despacha cada una de estas peticiones y prepara la respuesta para el cliente.

- **Archivo de configuración faces-config.xml**. Como ya se ha comentado anteriormente, en él se definen las reglas de navegación a través de la interfaz de usuario. Cada vez que FacesServlet despacha una petición, accede a este archivo para consultar qué tipo de vista debe ser enviada al cliente en función de la página que ha realizado la petición. La figura 37 muestra un extracto de un archivo de tipo faces-config.xml, en el que se indican las posibles vistas a generar cuando FacesServlet recibe una petición desde una página cliente llamada inicio.jsp.

```
:
<navigation-rule>
   <description>
      Ejemplo de navegación
   </description>
   <from-view-id>/inicio.jsp</from-view-id>
   <navigation-case>
      <from-outcome>correcto</from-outcome>
      <to-view-id>/ok.jsp</to-view-id>
   </navigation-case>
   <navigation-case>
      <from-outcome>fallo</from-outcome>
      <to-view-id>/error.jsp</to-view-id>
   </navigation-case>
</navigation-rule>
:
```

Fig. 37. *Regla de navegación entre páginas*

Además de las reglas de navegación, este archivo puede registrar otro tipo de información, como, por ejemplo, los JavaBean que van a ser gestionados por JSF.

- **Manejadores de eventos**. Son clases encargadas de responder a los distintos eventos que pueden producirse en la interfaz. Cuando se produce un evento en la interfaz que deseamos controlar desde la aplicación, se produce una petición a FacesServlet, el cual determina qué tipo de evento se ha producido, crea la instancia de la clase manejadora correspondiente e invoca al método encargado de su gestión.

4.3.2.2 LA VISTA

Habitualmente, la vista de una aplicación JSF está formada por páginas JSP en las que, utilizando una serie de acciones propias de JSF, se construye la interfaz de la aplicación.

La especificación JSF incluye dos tipos de librerías de acciones JSP:

- **La librería *html***. Esta librería proporciona, además de los controles básicos html con capacidades mejoradas, una serie de componentes gráficos complejos orientados a la presentación de datos en la

página. Todos los controles incluidos en esta librería se integran perfectamente con el modelo de JavaBean de Java, permitiendo vincular de manera automática las propiedades de un objeto con las de un control de la interfaz.

- **La librería *core*.** Además de incluir elementos básicos para la generación de una vista JSF, esta librería proporciona unos componentes especiales, conocidos como validadores y conversores, que permiten realizar validaciones y conversiones de datos, respectivamente, de la información suministrada por el usuario a través de la interfaz, todo ello de manera que el programador no tenga que incluir una sola línea de código para esta tarea.

Aunque habitualmente la utilización de JSF se centra en las páginas JSP, su uso no está limitado a éstas, ya que el árbol de componentes gráficos que constituyen el núcleo de JSF soporta diferentes tipos de interfaces de usuario.

Además de los componentes gráficos, otro elemento importante que forma parte de la vista son los JavaBean. Los JavaBean utilizados en JSF, conocidos también como **beans gestionados**, almacenan los datos de usuario suministrados a través de la interfaz.

Utilizando la sintaxis:

nombre_bean.propiedad

los componentes gráficos de la interfaz pueden acceder a los valores de las propiedades del bean.

Aunque según la figura 36 los JavaBean están incluidos dentro de la vista, también son accesibles desde el resto de los módulos. Algunos beans gestionados incluyen además de propiedades, métodos para gestión de eventos en la interfaz, actuando así como controlador.

4.3.2.3 EL MODELO

JavaServer Faces no aporta ningún elemento especial para la implementación del modelo, estando éste constituido con los elementos habituales para la encapsulación de la lógica de negocio de una aplicación Java: clases y Enterprise JavaBeans.

4.3.3 Proceso de construcción de una aplicación JSF

Seguidamente vamos a explicar el proceso a seguir para construir una aplicación basada en JavaServer Faces. A fin de comprender mejor el significado y utilidad de los distintos elementos que componen una aplicación de este tipo, desarrollaremos un ejemplo similar al creado en el apartado dedicado al estudio de la arquitectura MVC, consistente en una página de validación de usuarios contra una base de datos (figura 38).

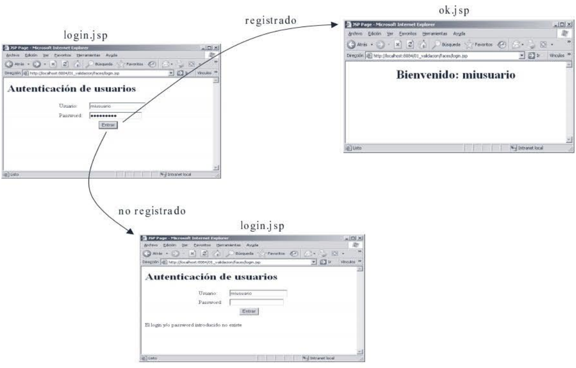


Fig. 38. *Páginas de la aplicación de ejemplo*

Si el usuario está registrado se le llevará a una página simple de bienvenida, mientras que si no lo está será devuelto a la página de login donde se le informará de que los datos introducidos no son correctos.

4.3.3.1 CONFIGURACIÓN DEL ENTORNO

Lo primero que habrá que hacer, una vez creada la estructura de aplicación Web, es configurar dicha aplicación para que pueda utilizar el framework JSF.

Para empezar habrá que hacerse con los archivos .jar que contienen el API JSF. Si dispusiéramos de un IDE compatible con JSF estos archivos se incorporarán a la aplicación al crear un proyecto que utilice este framework, en caso de no disponer de un IDE de esas características tendremos que descargar los archivos de librerías JSF desde la página Web de Sun (*http://java.sun.com/javaee/javaserverfaces/download.htm*l) y copiarlos en el

directorio *aplicacion_web\WEB-INF\lib*, siendo *aplicacion_web* el directorio raíz de la aplicación.

Seguidamente, crearemos el archivo faces-config.xml, inicialmente vacío, y lo situaremos en la misma carpeta donde se encuentra el archivo de configuración de la aplicación, web.xml. La estructura de faces-config.xml deberá quedar como se indica a continuación:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces
                                        Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config>

</faces-config>
```

Para finalizar con las tareas de configuración, debemos registrar el servlet FacesServlet en el archivo de configuración web.xml de la aplicación, para lo que debemos incluir en él los siguientes elementos:

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>
      javax.faces.webapp.FacesServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
     <servlet-name>Faces Servlet</servlet-name>
     <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

4.3.3.2 LÓGICA DE NEGOCIO DE LA APLICACIÓN

La lógica de negocio de una aplicación JSF es, como hemos indicado anteriormente, implementada a través de clases Java o EJB. En este ejemplo, la lógica de aplicación se encapsula en una clase llamada GestionClientes, en la que se define un único método llamado *valida()* que, a partir de los credenciales de usuario suministrados en un JavaBean, nos indica si dicho usuario existe o no dentro de la tabla clientes de la base de datos utilizada en el apartado anterior:

```
package logica;

import java.util.*;
import java.sql.*;
import beans.*;
public class GestionClientes {
    private Connection getConexion(){
        String driver="sun.jdbc.odbc.JdbcOdbcDriver";
        String con="jdbc:odbc:tienda";
        Connection cn=null;
        try{
            Class.forName(driver);
            cn=DriverManager.getConnection(con);
        }
        catch(Exception e){e.printStackTrace();}
        return cn;
    }
    public boolean valida(ValidacionBean c){
        Connection cn=null;
        ResultSet rs=null;
        boolean ret=false;
        try{
            cn=this.getConexion();
            Statement stm=cn.createStatement();
            //comprueba si el usaurio está registrado
            //a partir de los datos del bean
            String sql="Select * from clientes where ";
            sql+="usuario='"+c.getUsuario()+"' and pw='"+
                                     c.getPassword()+"'";
            rs=stm.executeQuery(sql);
            ret=rs.next();
            cn.close();
        }
        catch(Exception e){
            e.printStackTrace();
        }
        return ret;
    }
}
```

4.3.3.3 BEAN GESTIONADOS

Normalmente, una aplicación JSF contiene un bean por cada página JSP incluida en la misma. Estos beans definen propiedades y métodos asociados con los componentes gráficos utilizados en la página.

Se les llama "gestionados" a este tipo de JavaBean porque es **el framework el que se encarga de gestionar su ciclo de vida**, creando automáticamente las instancias, inicializándolas con los datos de usuario recogidos a través de los componentes de la interfaz y situándolas en el ámbito correspondiente. De lo único que tendrá que ocuparse el programador es de implementar la clase y registrarla en el archivo faces-config.xml.

Además de almacenar los datos de usuario, algunos bean gestionados suelen actuar como controladores de acciones, implementando los métodos de respuesta a los eventos sobre la interfaz que implican algún tipo de acción por parte del controlador y que afectan a la navegación entre páginas.

El siguiente listado corresponde al bean gestionado ValidacionBean, asociado a la página de login utilizada en la aplicación de ejemplo:

```
package beans;

import logica.*;
public class ValidacionBean {
    private String usuario;
    private String password;
    private String resultado;

    public ValidacionBean() {
    }
    public String getUsuario() {
        return usuario;
    }
    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
```

```
        }
        public String getResultado() {
            return resultado;
        }
        public void setResultado(String resultado) {
            this.resultado = resultado;
        }
        public String validar(){
            GestionClientes md=new GestionClientes();
            //invoca al método valida() con los datos almacenados
            //en las propiedades del bean
            if(md.valida(this)){
                return "validado";
            }
            else{
                resultado="El login y/o password introducido
                                                      no existe";
                return "novalidado";
            }
        }
    }
```

En esta clase podemos ver, por un lado, las propiedades que se encargan de mantener los valores suministrados en los dos campos de texto de la página (usuario y password) además de la propiedad *resultado*, encargada de almacenar el mensaje asociado al proceso de validación, y por otro lado el método *validar()*, cuya misión es invocar al método *valida()* del modelo a fin de determinar la existencia o no del usuario en la base de datos. El método *validar()* devuelve una cadena de caracteres que será utilizada por FacesServlet para determinar la página a la que deberá ser enviado el usuario.

Para que estos bean puedan ser gestionados por el framework, es necesario registrarlos en el archivo de configuración faces-config.xml. En el siguiente listado se indican los elementos que debemos añadir a este archivo para registrar la clase ValidacionBean como bean gestionado:

```
<managed-bean>
    <managed-bean-name>ValidacionBean</managed-bean-name>
    <managed-bean-class>
            beans.ValidacionBean
    </managed-bean-class>
```

```
        <managed-bean-scope>request</managed-bean-scope>
    </managed-bean>
```

A continuación, se explica el significado de estos elementos :

- **managed-bean**. Es el elemento principal en el que se incluyen los datos de registro del bean. Será necesario añadir un bloque <managed-bean> por cada bean que se quiera registrar.

- **managed-bean-name**. Es el nombre que permite referirse a la instancia del bean, utilizándose para acceder a las propiedades y métodos del mismo desde los componentes de la interfaz.

- **managed-bean-class**. Nombre cualificado de la clase a la que pertenece el bean.

- **managed-bean-scope**. Ámbito en el que será mantenida la instancia. Los posibles valores son *page*, *request*, *session* y *application*.

Además de los anteriores, managed-bean puede incluir también los siguientes subelementos:

- **managed-property**. Dado que al crear una instancia del bean FacesServlet invoca al constructor sin parámetros de la clase, si queremos que una propiedad pueda ser inicializada a un determinado valor durante este proceso, será necesario indicarlo a través de este elemento, suministrando los datos requeridos para la inicialización mediante los siguientes subelementos:

    - **property-name**: nombre de la propiedad que se quiere inicializar.

    - **property-class**: clase a la que pertenece el tipo de la propiedad.

    - **value**: valor al que se inicializará la propiedad cuando se cree la instancia.

4.3.3.4 COMPONENTES DE LA INTERFAZ DE USUARIO

De cara a crear la interfaz de usuario de una aplicación, JSF dispone de un amplio y variado conjunto de componentes gráficos, accesibles durante el desarrollo de una página JSP a través de las librerías de acciones *core* y *html*.

La funcionalidad de estos componentes se encuentra implementada a través de una serie de clases con capacidades para gestionar el estado del componente, mantener referencias a otros objetos, disparar eventos y generar la interfaz gráfica para el usuario.

En la figura 39 se muestra un esquema con algunas de las clases más importantes del modelo de componentes UI de JSF y la relación de herencia entre las mismas.

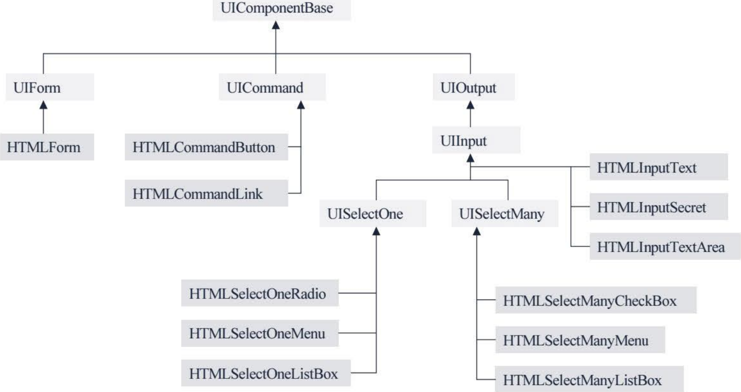


Fig. 39. *Modelo de componentes gráficos de JavaServer Faces*

Según se aprecia en el esquema, las clases del nivel inferior representan las implementaciones de componentes gráficos específicos para XHTML, proporcionados por la especificación JSF a través de la librería de acciones personalizadas *html*.

En la tabla de la figura 40 aparecen indicadas las etiquetas asociadas a las principales clases de componentes HTML, así como el tipo de control generado por cada una.

Componente	Tag	Tipo de control
HTMLCommandButton	CommandButton	Botón submit
HTMLCommandLink	CommandLink	Enlace HTML
HTMLInputText	inputText	Campo de texto
HTMLInputSecret	inputSecret	Campo de texto con caracteres ocultos
HTMLInputTextArea	inputTextarea	Caja de texto multilínea
HTMLSelectOneRadio	selectOneRadio	Conjunto de botones de radio
HTMLSelectOneMenu	selectOneMenu	Lista desplegable
HTMLSelectOneListBox	selectOneListbox	Lista normal
HTMLSelectManyCheckBox	selectManyCheckbox	Conjunto de casillas de verificación
HTMLSelectManyMenu	selectManyMenu	Lista desplegable de selección múltiple
HTMLSelectManyListBox	selectManyListbox	Lista normal de selección múltiple

Fig. 40. *Equivalencia entre componentes JSF y controles HTML*

El siguiente listado corresponde al código de la página login.jsp utilizada en la aplicación de ejemplo para validar al usuario:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                     Transitional//EN"
                     "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
                                          charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
```

```
        <h1>Autenticación de usuarios</h1>

        <f:view>
            <h:form>
                <table width="50%" align="center" border="0"
                               cellspacing="2" cellpadding="2">
                    <thead>
                        <tr>
                            <th></th>
                            <th></th>
                        </tr>
                    </thead>
                    <tbody>
                        <tr>
                            <td>Usuario:</td>
                            <td><h:inputText value=
                              "#{ValidacionBean.usuario}"/></td>
                        </tr>
                        <tr>
                            <td>Password:</td>
                            <td><h:inputSecret value=
                              "#{ValidacionBean.password}"/></td>
                        </tr>
                        <tr>
                            <td colspan="2" align="center">
                              <h:commandButton value="Entrar"
                              action=
                                  "#{ValidacionBean.validar}"/>
                            </td>
                        </tr>
                    </tbody>
                </table>
                <p>
                    <h:outputText value=
                              "#{ValidacionBean.resultado}" />
                </p>
            </h:form>
        </f:view>
        </body>
    </html>
```

Como vemos, inicialmente debemos declarar a través de la directiva *taglib* las dos librerías de acciones JSF. **Todas las acciones pertenecientes a estas librerías que vayan a ser utilizadas en la página JSP deberán estar incluidas dentro de la acción <f:view>** perteneciente al core.

Por otra parte, las acciones asociadas a controles HTML deben aparecer dentro del elemento <h:form> que, como se puede comprobar, no incluye ningún atributo de tipo *action* en el que se indique la URL a solicitar cuando se produzca el submit del formulario, dado que toda petición que se realice desde la capa cliente será dirigida a FacesServlet.

En cuanto a los componentes gráficos utilizados en la página, vemos que cada tipo de control de edición, cajas de texto estándar y tipo password, es generado mediante una acción diferente, haciendo su uso más intuitivo para el desarrollador de la página. Ambos tipos de control de edición permiten asociar el contenido del mismo con una determinada propiedad del bean a través del atributo *value*. En este caso estamos asociando los controles *usuario* y *password* con las propiedades del mismo nombre existentes en ValidacionBean; una vez que se produzca el submit del formulario, los valores introducidos por el usuario en estos controles serán almacenados automáticamente por FacesServlet en las propiedades de la instancia del bean, que será creada al recibir la petición.

Utilizando la vinculación de beans con los componentes de la interfaz gráfica JSF, es posible también mostrar el valor de alguna de las propiedades del objeto en la página de respuesta. Éste es el caso de la propiedad *resultado* de ValidacionBean vinculada al atributo *value* del componente <h:outputText>, cuya función es mostrar un texto sin formato en la página de usuario.

La notación utilizada para vincular un control con una propiedad de un bean es conocida como notación EL:

#{nombre_bean.propiedad}

Este simple mecanismo de vinculación entre los componentes de la interfaz de usuario y un bean gestionado permite, además de capturar los datos de usuario, implementar métodos en el bean que se ejecuten como respuesta a los eventos de acción provocados sobre los componentes de la interfaz. Éste es el caso del método *validar()* del bean, asociado con el evento de acción que se produce al pulsar el botón de comando de la página.

De esta forma, cuando se produzca el submit de la página, tras crear la instancia del bean y rellenar sus propiedades con los valores suministrados por el

usuario a través de los campos del formulario, el servlet **FacesServlet invocará al método *validar()* del bean, el cual devolverá una cadena de caracteres al servlet que representa la dirección lógica de la nueva vista que deberá ser enviada al cliente**.

4.3.3.5 NAVEGACIÓN ENTRE PÁGINAS

Partiendo de la dirección lógica devuelta por el método manejador del evento de acción, FacesServlet debe determinar la página que tiene que ser devuelta como respuesta al navegador cliente.

La información que relaciona direcciones virtuales con reales se suministra a través del archivo de configuración faces-config.xml, dentro de lo que se conocen como **reglas de navegación entre páginas**.

El siguiente listado corresponde al archivo faces-config.xml de la aplicación de validación, indicándose en fondo sombreado la sección correspondiente a las reglas de navegación:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config>
    <managed-bean>
        <managed-bean-name>ValidacionBean</managed-bean-name>
        <managed-bean-class>beans.ValidacionBean
             </managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
    </managed-bean>
    <navigation-rule>
        <from-view-id>/login.jsp</from-view-id>
        <navigation-case>
             <from-outcome>validado</from-outcome>
             <to-view-id>/ok.jsp</to-view-id>
        </navigation-case>
        <navigation-case>
             <from-outcome>novalidado</from-outcome>
             <to-view-id>/login.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

Seguidamente, describiremos el significado de estos elementos:

- **navigation-rule**. Define una regla de navegación. Cada regla de navegación vincula una página origen con los distintos destinos a los que desde ella se puede navegar.

- **from-view-id**. Contiene la URL de la página origen para la que se define la regla de navegación.

- **navigation-case**. Describe, para una determinada regla de navegación, un caso de posible destino al que puede ser enviado el usuario desde la página origen especificada mediante el elemento anterior. Cada navigation-case consta de los siguientes subelementos:

    - from-outcome: dirección virtual asociada al destino.

    - to-view-id: dirección real del destino.

4.3.4 AJAX en aplicaciones JSF

Como se desprende del proceso de funcionamiento de una aplicación JSF que se acaba de exponer, toda petición cliente que se realiza a través de los componentes gráficos JSF se dirige hacia el servlet FacesServlet, por lo que si quisiéramos añadir funcionalidad AJAX a este tipo de aplicaciones deberíamos crear un nuevo servlet adicional que se encargara de atender a las peticiones que se recibieran desde los scripts de cliente.

Esta aproximación, que es similar a la seguida en las aplicaciones MVC convencionales, no aprovecha las capacidades de la tecnología JSF, pues todas las peticiones asíncronas serían gestionadas fuera del ciclo de vida de la aplicación JSF (figura 41). No obstante, la versión JSF 2.0, incluida en la versión Java EE 6.0, permite incorporar funcionalidad AJAX dentro de una página Web con componentes JSF, manteniendo el ciclo de vida normal de la aplicación (figura 42). Este punto será analizado en el siguiente apartado.

Otra posibilidad, que permitiría aprovechar conjuntamente las características de AJAX y JSF, consiste en utilizar librarías de componentes JSF específicas que incorporen funcionalidad AJAX, como es el caso de la librería RichFaces, que comentaremos más adelante.

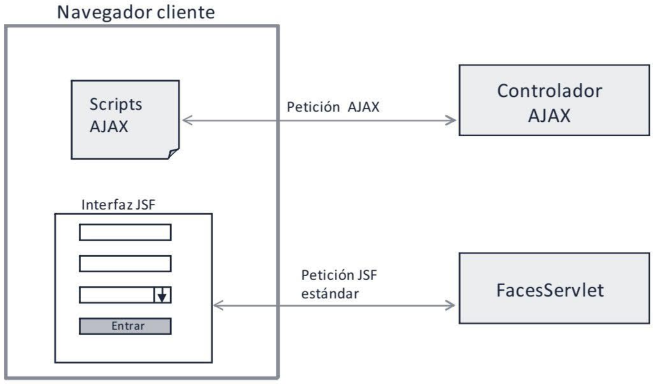


Fig. 41. *Desacoplamiento entre código AJAX y JSF*

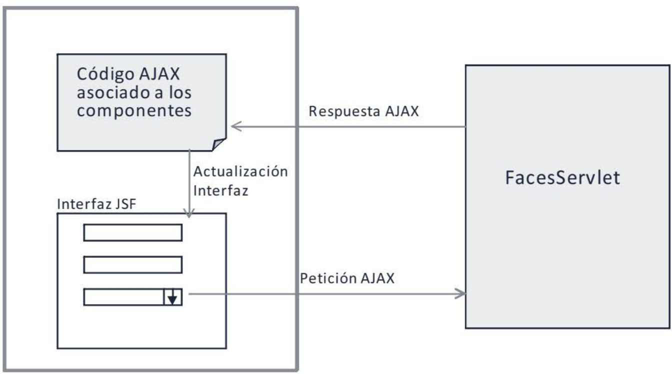


Fig. 42. *Utilización de funcionalidad AJAX JSF 2*

4.3.4.1 FUNCIONALIDAD AJAX EN JSF 2

La versión JSF 2 incorpora sustanciales mejoras respecto a las versiones anteriores. Entre estas mejoras destaca, sin duda alguna, la posibilidad de incorporar funcionalidad AJAX a los componentes JSF tradicionales sin necesidad de escribir una sola línea de código JavaScript.

Para ello, la librería core incorpora un nuevo elemento llamado precisamente ajax, el cual debe anidarse dentro del componente que tiene que provocar la petición AJAX. Este tag <f:ajax> incorpora todo el código JavaScript para llevar a cabo las tareas de petición asíncrona y procesamiento de respuestas.

El elemento <f:ajax> dispone de dos atributos con los que se configura su funcionalidad:

- **event**. Indica el nombre del evento del componente, dentro del que se encuentra anidado <f:ajax>, que provocará la petición AJAX al servidor. Este parámetro es opcional, de modo que, si no se indica, la petición AJAX se realizará cuando se produzca el evento predeterminado del componente. Todo componente tiene un evento predeterminado, en el caso, por ejemplo, de una lista de selección, sería *onChange*.

  Cuando se produce el evento indicado en este atributo, la petición AJAX provoca que se actualice solamente la propiedad del bean gestionado que esté asociada a este componente.

- **rendered**. Contendrá el identificador del componente (valor del atributo *id*) que será actualizado tras la petición. Tras actualizarse la propiedad del bean asociada al componente con la petición AJAX, la respuesta consistirá en refrescar los datos del componente indicado en rendered, actualizando éste con los nuevos valores de las propiedades del bean al que se encuentra vinculado. Se puede indicar el refresco de varios componentes, indicando la lista de sus identificadores separados por un espacio.

Por ejemplo, la siguiente definición del campo de texto provocaría una petición AJAX cada vez que se introduzca un carácter en el componente (evento "keyup"), actualizándose los datos mostrados por el componente de salida "resultado" con los nuevos valores de la propiedad "mensaje" del bean:

```
<h:outputText id="resultado" value="#{bean.mensaje}"/>
```

```
<h:inputText value=”#{bean.contenido}”>

        <f:ajax event=”keyup” render=”resultado”/>

</h:inputText>
```

4.3.4.1.1 Ejemplo práctico: generación dinámica de tabla de libros

A continuación, vamos a presentar un ejemplo práctico en el que emplearemos el elemento <f:ajax> para actualizar una tabla que muestra los libros asociados a un determinado tema, cada vez que un nuevo tema es seleccionado en la una lista desplegable (figura 43).

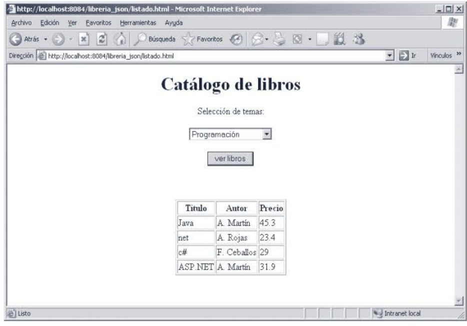


Fig. 43. *Página del catálogo de libros*

Realmente, como podemos deducir de la imagen anterior, se trata de una nueva implementación del ejercicio “Catálogo de libros” desarrollado en la práctica 3.1. Como podremos comprobar, en esta nueva versión del ejercicio se simplifica enormemente el código de la aplicación, pues, por una lado, toda la funcionalidad JavaScript de cliente es proporcionada por el elemento <f:ajax>, no siendo necesario desarrollar ningún script dentro de la página. Por otro lado, la utilización de componentes JSF para la presentación de la información, evita tener que recurrir a la generación “manual” del código JSON de las respuestas AJAX.

Es importante destacar que, utilizando JSF 2, **la programación de lado de servidor no se ve de ninguna manera alterada** para el caso de que decidamos utilizar AJAX en la aplicación.

El desarrollo de esta aplicación se realizará empleando el patrón MVC, por lo que lo primero que habrá que crear será la capa del Modelo. En nuestro caso, se trata de una clase que implementa un par de métodos para realizar la funcionalidad requerida por la aplicación, que es obtener la lista de temas disponibles y la lista de libros asociados al tema elegido:

```
package modelo;

import beans.*;
import java.util.*;
import java.sql.*;
public class GestionLibros {
    public List<Tema> getTemas(){
        Connection cn=null;
        Statement st=null;
        ResultSet rs=null;
        ArrayList<Tema> temas=null;
        try{
            //se utiliza un driver nativo de MySQL
            Class.forName("com.mysql.jdbc.Driver");
            cn=DriverManager.getConnection(
     "jdbc:mysql://localhost:3306/bdlibros","root","root");
            st=cn.createStatement();
            rs=st.executeQuery("select * from temas");
            //se crea el arraylist de temas
            temas=new ArrayList<Tema>();
            //rellena el ArrayList de objetos Tema
            while(rs.next()){
                temas.add(new Tema(rs.getInt("idTema"),
                                   rs.getString("tema")));
            }
        }
        catch(Exception e){e.printStackTrace();}
        return temas;
    }
    public List<Libro> getLibrosByTema(int idTema){
        Connection cn=null;
        Statement st=null;
        ResultSet rs=null;
        ArrayList<Libro> libros=null;
        try{
```

```
            //se utiliza un driver nativo de MySQL
            Class.forName("com.mysql.jdbc.Driver");
            cn=DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/bdlibros","root","root");
            st=cn.createStatement();
            rs=st.executeQuery("select * from libros
                     where idTema="+idTema);
            //se crea el arraylist de libros
            libros=new ArrayList<Libro>();
            //rellena el ArrayList de objetos Libro obtenidos
            while(rs.next()){
                Libro lb=new Libro(rs.getString("isbn"),
                        rs.getInt("idTema"),
                        rs.getString("titulo"),
                        rs.getString("autor"),
                        rs.getDouble("precio"));
                libros.add(lb);
            }
        }
        catch(Exception e){e.printStackTrace();}
        return libros;
    }
}
```

Según se desprende del listado anterior, nos debemos apoyar en dos clases de tipo JavaBean que encapsulen los datos de los registros de las tablas "temas" y "libros", por lo que llamaremos a estos JavaBeans Tema y Libro:

```
package beans;
public class Tema {
    private int idTema;
    private String tema;

    public Tema() {
    }
    public Tema(int idTema, String tema) {
        this.idTema = idTema;
        this.tema = tema;
    }
    public int getIdTema() {
        return idTema;
    }
```

```
    public void setIdTema(int idTema) {
        this.idTema = idTema;
    }
    public String getTema() {
        return tema;
    }
    public void setTema(String tema) {
        this.tema = tema;
    }
}

package beans;
public class Libro {
    private String isbn;
    private int idTema;
    private String titulo;
    private String autor;
    private double precio;

    public Libro(String isbn, int idTema,
           String titulo, String autor, double precio) {
        this.isbn = isbn;
        this.idTema = idTema;
        this.titulo = titulo;
        this.autor = autor;
        this.precio = precio;
    }
    public Libro() {
    }
    public String getAutor() {
        return autor;
    }
    public void setAutor(String autor) {
        this.autor = autor;
    }
    public int getIdTema() {
        return idTema;
    }
    public void setIdTema(int idTema) {
        this.idTema = idTema;
    }
```

```
        public String getIsbn() {
            return isbn;
        }
        public void setIsbn(String isbn) {
            this.isbn = isbn;
        }
        public double getPrecio() {
            return precio;
        }
        public void setPrecio(double precio) {
            this.precio = precio;
        }
        public String getTitulo() {
            return titulo;
        }
        public void setTitulo(String titulo) {
            this.titulo = titulo;
        }
    }
```

La parte del controlador será implementada mediante un bean gestionado. Este bean constará de dos tipos de propiedades:

- Las propiedades *temas* y *libros*. Tendrán como misión obtener del modelo la colección de temas y libros que serán utilizados para poblar los componentes de la página. Por tanto, sólo será necesario proporcionar la implementación de los métodos get de ambas propiedades.

- La propiedad *idTema*. Su objetivo será almacenar el identificador de tema seleccionado. Esta propiedad se actualizará con cada petición AJAX.

El código de este bean gestionado se muestra en el siguiente listado:

```
import beans.*;
import modelo.*;
import java.util.*;
import javax.faces.bean.*;
@ManagedBean(name="librosTemasBean")
@RequestScoped
public class LibrosTemasBean {
```

```
        private int idTema;

        public int getIdTema() {
            return idTema;
        }
        public void setIdTema(int idTema) {
            this.idTema = idTema;
        }
        public List<Tema> getTemas(){
            GestionLibros gestion=new GestionLibros();
            return gestion.getTemas();
        }
        public List<Libro> getLibros(){
            GestionLibros gestion=new GestionLibros();
            return gestion.getLibrosByTema(idTema);
        }
    }
```

Finalmente, la parte más interesante del ejercicio: la página JSP que contiene la interfaz gráfica de la aplicación. Esta interfaz está formada, por un lado, por un control de lista de selección que mostrará la lista de temas existentes y que, cada vez que se seleccione un tema, lanzará una petición AJAX al servidor para actualizar la tabla de libros. La tabla de libros es el otro componente de la interfaz, y será generado a partir de un componente JSF de tipo <h:dataTable>.

He aquí el listado completo de la página:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
    <h:head>
        <title>Facelet Title</title>
    </h:head>
    <h:body>
        <center>
        <h1>Catalogo de libros</h1>
        <h:form>
            Selección de temas:
            <h:selectOneMenu
```

```
                    value="#{librosTemasBean.idTema}">
                <f:selectItems
        value="#{librosTemasBean.temas}" var="tema"
        itemLabel="#{tema.tema}" itemValue="#{tema.idTema}"/>
                <f:ajax event="valueChange"
                           render="tablalibros"/>
            </h:selectOneMenu>
            <br/>
            <br/>
            <h:dataTable id="tablalibros" border="1"
                value="#{librosTemasBean.libros}" var="lib">
                <h:column>
                    <f:facet name="header">
                        <h:outputText value="Titulo"/>
                    </f:facet>
                    <h:outputText value="#{lib.titulo}"/>
                </h:column>
                 <h:column>
                    <f:facet name="header">
                        <h:outputText value="Autor"/>
                    </f:facet>
                    <h:outputText value="#{lib.autor}"/>
                </h:column>
                 <h:column>
                    <f:facet name="header">
                        <h:outputText value="Precio"/>
                    </f:facet>
                    <h:outputText value="#{lib.precio}"/>
                </h:column>
            </h:dataTable>
        </h:form>
        </center>
    </h:body>
</html>
```

Hay que destacar del listado anterior la manera en la que se genera una lista de selección en JSF; según el tipo de lista que queramos, desplegable, abierta o grupo de botones de radio, utilizaremos el componente <h:selectOneMenu>, <h:selectOneListBox> o <h:selectOneRadio>. En todos los casos, los datos que van a poblar la lista son obtenidos mediante un elemento <f:selectItems>.

La tabla, por su parte, se genera con un componente <h:dataTable>, en cuyo atributo *value* se deberá indicar la colección de elementos a mostrar.

4.3.4.1.2 La librería de componentes RichFaces

RichFaces es una librería de componentes avanzados para JSF que permiten integrar fácilmente las capacidades AJAX dentro de una aplicación.

A través de los componentes RichFaces podemos utilizar AJAX en páginas JSP creadas con componentes JSF 1.2, de forma similar a como hemos visto se trabaja en JSF 2.

RichFaces forma parte de la comunidad JBoss, así pues, para poder utilizar esta librería debemos descargar el archivo correspondiente desde la dirección *http://www.jboss.org/richfaces/download*. En el momento de escribir este libro, la última versión estable de RichFaces es la 3.3.3.

4.3.4.1.2.1 Utilización de RichFaces dentro de una aplicación Web

Una vez descargado el archivo .zip correspondiente, al descomprimirlo encontraremos una carpeta \lib en la que se encuentran los archivos .jar necesarios para utilizar los componentes. Estos archivos debemos incluirlos en el directorio \lib de nuestra aplicación. Si estamos empleando NetBeans, bastará con agregarlos a las librerías del proyecto.

Además de estos archivos, para poder utilizar los componentes RichFaces debemos incorporar las siguientes librerías de apache al proyecto:

commons-beanutils.jar

commons-collections.jar

commons-digester.jar

commons-logging.jar

En la figura 44 se indica el aspecto que debería tener la carpeta “Libraries” del proyecto NetBeans después de agregar los archivos .jar de RichFaces.

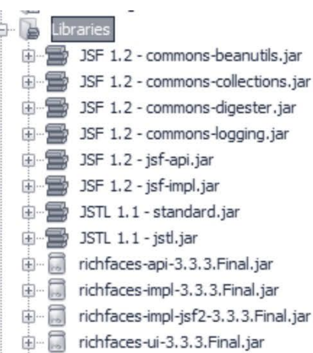


Fig. 44. *Librerías RichFaces añadidas a un proyecto NetBeans*

Además de incluir las librerías anteriores, para poder utilizar RichFaces dentro de la aplicación es necesario realizar **el registro del filtro org.ajax4jsf.Filter** en el archivo de configuración web.xml y asociarlo al servlet Faces Servlet:

```
<filter>
      <display-name>RichFaces Filter</display-name>
      <filter-name>richfaces</filter-name>
      <filter-class>org.ajax4jsf.Filter</filter-class>
  </filter>
  <filter-mapping>
      <filter-name>richfaces</filter-name>
      <servlet-name>Faces Servlet</servlet-name>
      <dispatcher>REQUEST</dispatcher>
      <dispatcher>FORWARD</dispatcher>
      <dispatcher>INCLUDE</dispatcher>
  </filter-mapping>
```

De esta manera, todas las peticiones dirigidas a Faces Servlet pasarán primero por el filtro.

De cara a poder hacer uso de los componentes dentro de una página JSP, será necesario incluir en la misma el correspondiente taglib que haga referencia a la librería específica del tipo de componente que queremos utilizar.

4.3.4.1.3 Tipos de componentes RichFaces

Los componentes proporcionados por RichFaces se organizan en dos grandes librerías:

- **La librería de componentes gráficos enriquecidos**. Se trata de componentes gráficos avanzados que, junto con los componentes JSF estándares, permiten crear interfaces gráficas enriquecidas con la misma calidad que pueden tener las aplicaciones de escritorio desarrolladas con swing. En este grupo de componentes, tenemos, por ejemplo, el control calendar para la entrada de fechas mediante un calendario, el control inputNumberSlider para selección de un valor numérico en una barra desplazable, o el dataDefinitionList para la presentación de datos en forma de lista de definición.

  Para poder utilizar estos componentes en una página se deberá incluir la siguiente referencia:

  <%@ taglib uri="http://richfaces.org/rich" prefix="rich"%>

- **La librería de componentes a4j**. Se trata del conjunto de componentes que incorporan la funcionalidad AJAX dentro de una página Web. Pueden utilizarse tanto con componentes rich como con los estándares JSF. En el siguiente apartado analizaremos con detalle algunos de los componentes más importantes de esta librería, de momento indicar que para poder hacer uso de los mismos dentro de una página tendremos que incluir la siguiente referencia:

  <%@ taglib uri="http://richfaces.org/a4j" prefix="a4j"%>

4.3.4.1.4 Principales componentes a4j

De cara a incorporar funcionalidad AJAX dentro de una página JSP basada en JSF sin necesidad de incluir ningún tipo de código JavaScript, podemos hacer uso de los siguientes tags de la librería de componentes a4j de RichFaces:

- **<a4j:commandButton>**. Se trata de un botón tipo submit que al ser pulsado provoca la ejecución de un método de acción asociado en el bean gestionado, provocando a continuación, la actualización de un elemento determinado de la página.

  Como vemos, su comportamiento es similar al commandButton estándar de JSF pero, a diferencia de éste, el a4j no provoca la

navegación a una determinada página, sino la actualización de otro componente de la misma página a través de AJAX.

Los principales atributos de esta acción son:

- **action**. Expresión EL, que hace referencia al método del bean gestionado que queremos ejecutar con la pulsación del botón.

- **value**. Texto a mostrar en el control.

- **reRender**. Identificador o identificadores del elemento o elementos que deben ser actualizados tras la petición. Puede indicarse más de un elemento a actualizar, indicando sus identificadores separados por un espacio.

El siguiente bloque de código correspondería a la página principal de un traductor español-inglés. Al pulsar el botón se produce una petición AJAX que ejecuta el método “traducir()” del bean gestionado, actualizando a continuación, el texto de salida con el resultado de la traducción:

```
Introduzca texto en español:
 <h:inputText value="#{bean.frase}"/>
 <br/>
 Inglés:<h:outputText id="traduccion"
 value="#{bean.traduccion}"/>
 <br/>
 <a4j:commandButton value="Traducir"
  action="#{bean.traducir}"
  reRender="traduccion"/>
```

El método de gestión de la acción definido en el bean tendrá que tener el mismo formato que en el caso de los commandButton estándares, es decir, no recibirá parámetros y devolverá un String. Sin embargo, dado que los botones de comandos AJAX no provocan navegación a otra página, **el String devuelto por el método será ignorado**.

Al igual que en la librería de componentes JSF estándares, tenemos una variante de este componente, que es el <a4j:commandLink>.

- **<a4j:support>**. Provoca una petición AJAX al servidor cuando se produce un evento sobre un determinado componente, actualizando a continuación uno o varios componentes de la interfaz.

  El elemento a4j:support debe anidarse dentro del elemento cuyo evento provocará la petición AJAX. Los principales atributos de esta acción son:

  - **event**. Nombre del evento que provocará la petición. Se utilizan los nombre de evento JavaScript (onchange, onkeypress, etc.).

  - **reRender**. Identificador o identificadores del elemento o elementos que deben ser actualizados tras la petición.

  El siguiente extracto de código corresponde a una página en la que se actualiza un campo de texto "texto2" cada vez que se introduce un carácter en otro campo de texto "texto1":

```
<h:inputText id="texto1" value="#{bean.dato1}">
 <aj4:support event="onkeypress" reRender="texto2"/>
</h:inputText>
<h:inputText id="texto2" value="#{bean.dato2}"/>
```

  Es importante destacar que cuando se produce la petición AJAX al servidor, la única propiedad del bean que se actualiza con los valores de la interfaz es aquella que se encuentra vinculada al componente donde se produce el evento, en nuestro ejemplo, sería *dato1*.

- **<a4j:poll>**. Este componente se encarga de lanzar peticiones AJAX al servidor de forma periódica, actualizando tras la petición uno o varios componentes de la interfaz. Entre sus atributos tenemos:

  - **intervall**. Intervalo de tiempo en milisegundos entre cada petición asíncrona al servidor.

  - **reRender**. Identificador o identificadores del elemento o elementos que deben ser actualizados tras la petición.

PRÁCTICA 4.1. CATÁLOGO DE LIBROS

Descripción

Vamos a realizar una nueva versión del catálogo de libros desarrollado en la práctica 3.1 del capítulo anterior, en la que incluiremos además una página de login para autenticación de usuarios, de modo que solamente los usuarios autenticados puedan consultar el catálogo, disponiendo también de una página de registro que permita añadir nuevos usuarios a la base de datos. La figura 45 muestra el diagrama de páginas de la aplicación.

La página de login mostrará, además, en la parte superior un texto que mostrará las ofertas destacadas del sitio, texto que se irá actualizando con una nueva oferta cada 3 segundos.

Por su parte, la página de registro permitirá a los usuarios comprobar si el usuario que han elegido está o no libre antes de proceder al registro de los datos.

Desarrollo

La práctica se desarrollará completamente en JSF y utilizaremos los componentes AJAX de RichFaces para la recuperación periódica de las noticias destacadas en la página de login, para la comprobación de la existencia del identificador de usuario elegido y durante la recuperación de los libros asociados al tema elegido.

La base de datos será similar a la empleada en la práctica 3.1, aunque se utilizarán además las tablas "clientes" y "ofertas" para el registro de los clientes y las ofertas destacadas, respectivamente (figura 46)

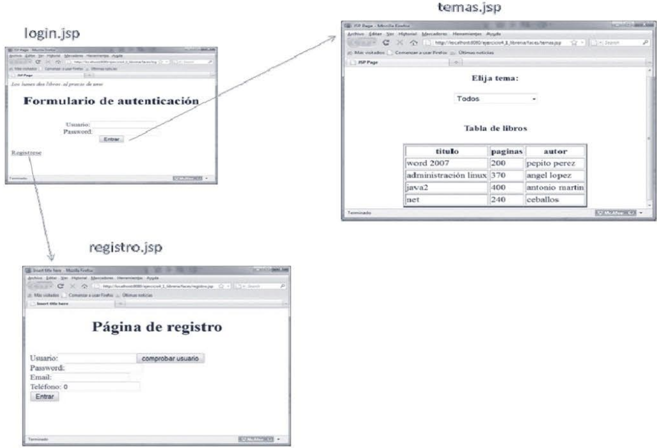
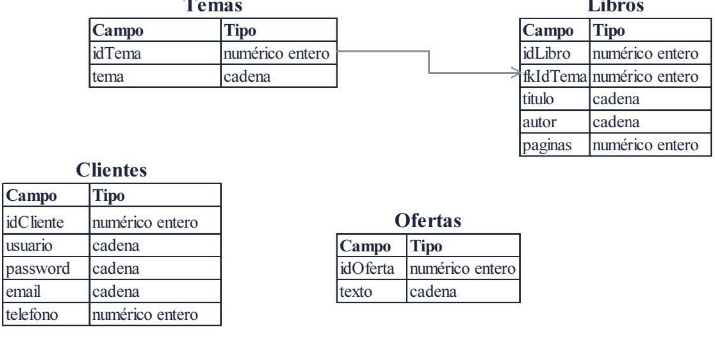


Fig. 45. *Diagrama de páginas de la aplicación*



Temas

Campo	Tipo
idTema	numérico entero
tema	cadena

Libros

Campo	Tipo
idLibro	numérico entero
fkIdTema	numérico entero
titulo	cadena
autor	cadena
paginas	numérico entero

Clientes

Campo	Tipo
idCliente	numérico entero
usuario	cadena
password	cadena
email	cadena
telefono	numérico entero

Ofertas

Campo	Tipo
idOferta	numérico entero
texto	cadena

Fig. 46. *Tablas de la base de datos de libros*

Código

Seguidamente, expondremos los listados de código de cada uno de los bloques de la aplicación.

Clase GestionDatos

En primer lugar, aquí tenemos el código de la clase GestionDatos, donde se encapsula toda la lógica de negocio de la aplicación. En este caso, hemos optado por utilizar una base de datos de MySQL:

```
package modelo;
import java.util.*;
import beans.*;

import java.sql.*;
public class GestionDatos {
        private Connection getConnection(){
            Connection cn=null;
            try{
               Class.forName("com.mysql.jdbc.Driver") ;
               cn=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/libros","root","root");

            }
            catch(Exception ex){
                ex.printStackTrace();
            }
            return cn;
        }
        public List<Tema> getTemas(){
            Connection cn=getConnection();
            Statement st=null;
            ResultSet rs=null;
            String sql="select * from temas";
            ArrayList<Tema> temas=new ArrayList<Tema>();
            try{
                st=cn.createStatement();
                rs=st.executeQuery(sql);
                while(rs.next()){
                    Tema t=new Tema(rs.getInt("idTema"),
                          rs.getString("tema"));
```

```
                temas.add(t);
            }
        }
        catch(Exception ex){
            ex.printStackTrace();
        }
        return temas;
    }
    //devuelve la colección de libros pertenecientes
   //a un determinado tema
    public List<Libro> getLibrosByTema(int idTema){
        Connection cn=getConnection();
        Statement st=null;
        ResultSet rs=null;
        String sql="select * from libros ";
        if(idTema!=-1)
            sql+="where fkidTema="+idTema;
        ArrayList<Libro> libros=new ArrayList<Libro>();
        try{
            st=cn.createStatement();
            rs=st.executeQuery(sql);
            while(rs.next()){
                Libro t=new Libro(rs.getInt("isbn"),
                            rs.getString("titulo"),
                            rs.getInt("fkidTema"),
                            rs.getInt("paginas"),
                            rs.getString("autor"));
                libros.add(t);
            }
        }
        catch(Exception ex){
            ex.printStackTrace();
        }
        return libros;
    }
   public boolean login(String user, String password) {
        boolean resultado=false;
        Connection cn=getConnection();
        Statement st=null;
        ResultSet rs=null;
        String sql="select * from clientes where
```

```
         usuario='"+user+"' and password='"+password+"'";
        try{
            st=cn.createStatement();
            rs=st.executeQuery(sql);
            if(rs.next()){
                resultado=true;
            }
        }
        catch(Exception ex){
            ex.printStackTrace();
        }
        return resultado;
}
 //devuelve una oferta aleatoriamente
public String getOferta(){
        String oferta="";
        Connection cn=getConnection();
        Statement st=null;
        ResultSet rs=null;
        String sql="select texto from ofertas";
        ArrayList<String> todas=new ArrayList<String>();
        try{
            st=cn.createStatement();
            rs=st.executeQuery(sql);
            while(rs.next()){
                todas.add(rs.getString("texto"));
            }
            int pos=(int)(Math.random()*todas.size());
            oferta=todas.get(pos);
        }
        catch(Exception ex){
            ex.printStackTrace();
        }
        return oferta;
 }
 public void registrar(Cliente c) {

        Connection cn=getConnection();
        Statement st=null;

        String sql="insert into
```

```
                clientes(usuario,password,email,telefono) ";
            sql+="values ('"+c.getUsuario()+"','"+
              c.getPassword()+"','"+c.getEmail()+
              "',"+c.getTelefono()+")";
            try{
                st=cn.createStatement();
                st.execute(sql);
            }
            catch(Exception ex){
                ex.printStackTrace();
            }
        }
        //comprueba si existe el usuario en la base de datos
        public boolean usuarioExiste(String usuario){
            boolean resultado=false;
            Connection cn=getConnection();
            Statement st=null;
            ResultSet rs=null;
            String sql="select * from clientes
                           where usuario='"+usuario+"'";
            try{
                st=cn.createStatement();
                rs=st.executeQuery(sql);
                if(rs.next()){
                    resultado=true;
                }
            }
            catch(Exception ex){
                ex.printStackTrace();
            }
            return resultado;
        }
}
```

JavaBeans

El intercambio de datos entre el modelo y el controlador se lleva a cabo utilizando tres clases de tipo JavaBeans que encapsulan los datos de los registros de las tablas implicadas en la aplicación: Tema, Libro y Cliente:

```
package beans;
public class Tema {
```

```
    private int idTema;
    private String tema;
    public Tema(){
    }
    public Tema(int idTema, String tema) {
        this.idTema = idTema;
        this.tema = tema;
    }
    public int getIdTema() {
        return idTema;
    }
    public void setIdTema(int idTema) {
        this.idTema = idTema;
    }
    public String getTema() {
        return tema;
    }
    public void setTema(String tema) {
        this.tema = tema;
    }
}
package beans;

public class Libro {
    private int idLibro;
    private String titulo;
    private int idTema;
    private int paginas;
    private String autor;
    public Libro(){}
    public Libro(int idLibro, String titulo,
             int idTema, int paginas, String autor) {
        this.idLibro = idLibro;
        this.titulo = titulo;
        this.idTema = idTema;
        this.paginas = paginas;
        this.autor = autor;
    }
    public String getAutor() {
        return autor;
    }
```

```
    public void setAutor(String autor) {
        this.autor = autor;
    }
    public int getIdLibro() {
        return idLibro;
    }
    public void setIdLibro(int idLibro) {
        this.idLibro = idLibro;
    }
    public int getIdTema() {
        return idTema;
    }
    public void setIdTema(int idTema) {
        this.idTema = idTema;
    }
    public int getPaginas() {
        return paginas;
    }
    public void setPaginas(int paginas) {
        this.paginas = paginas;
    }
    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
}
package beans;
public class Cliente {
        private int idCliente;
        private String usuario;
        private String password;
        private String email;
        private long telefono;
        public Cliente() {
                super();
                // TODO Auto-generated constructor stub
        }
        public Cliente(int idCliente,
                        String usuario, String password,
```

```
                    String email, long telefono) {
            super();
            this.idCliente = idCliente;
            this.usuario = usuario;
            this.password = password;
            this.email = email;
            this.telefono = telefono;
        }
        public int getIdCliente() {
            return idCliente;
        }
        public void setIdCliente(int idCliente) {
            this.idCliente = idCliente;
        }
        public String getUsuario() {
            return usuario;
        }
        public void setUsuario(String usuario) {
            this.usuario = usuario;
        }
        public String getPassword() {
            return password;
        }
        public void setPassword(String password) {
            this.password = password;
        }
        public String getEmail() {
            return email;
        }
        public void setEmail(String email) {
            this.email = email;
        }
        public long getTelefono() {
            return telefono;
        }
        public void setTelefono(long telefono) {
            this.telefono = telefono;
        }
    }
```

Clase LoginBean

Se trata del bean gestionado encargado de capturar los credenciales del usuario y realizar la validación. Dispone además del método *getOferta()*, que devuelve la oferta a mostrar:

```
package beans.gestionados;

import modelo.*;
public class LoginBean {
    private String usuario;
    private String password;

    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getUsuario() {
        return usuario;
    }
    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }
    public String getOferta(){
        GestionDatos gd=new GestionDatos();
        return gd.getOferta();
    }
    public String doLogin(){
      GestionDatos gd=new GestionDatos();
      if(gd.login(this.getUsuario(), this.getPassword())){
            return "success";
      }
      else{
            return "error";
      }
    }
}
```

Clase TemasLibrosBean

Bean gestionado que expone dos propiedades para la obtención de los temas y libros, así como una propiedad para capturar el tema elegido:

```
package beans.gestionados;
import beans.*;
import java.util.*;
import javax.faces.model.*;
import modelo.*;
public class TemasLibrosBean {
  private int idTema=-1;
  public void setIdTema(int t){
      idTema=t;
  }
  public int getIdTema(){
      return idTema;
  }
  public List<SelectItem> getTemas(){
      ArrayList<SelectItem> temas=
             new ArrayList<SelectItem>();
      GestionDatos libreria=new GestionDatos();
      for(Tema t:libreria.getTemas()){
          SelectItem item=
             new SelectItem(t.getIdTema(),t.getTema());
          temas.add(item);
      }
      return temas;
  }
  public List<Libro> getLibros(){
      GestionDatos libreria=new GestionDatos();
      return libreria.getLibrosByTema(idTema);
  }
}
```

Clase ClienteBean

Este bean gestionado está definido como una subclase de Cliente. En él se implementan los métodos para registro de un nuevo usuario y la comprobación de usuario existente.

```
package beans.gestionados;
import beans.Cliente;
import modelo.*;
public class ClienteBean extends Cliente {
    private String resultado;
    public ClienteBean() {
      super();

    }
    public ClienteBean(int idCliente, String usuario,
              String password, String email, long telefono) {
      super(idCliente, usuario, password, email, telefono);

    }
    public String getResultado() {
        return resultado;
    }
    public String doRegistrar(){
      GestionDatos gd=new GestionDatos();
      try{
            gd.registrar(this);
            return "success";
      }
      catch(Exception ex){
            ex.printStackTrace();
            return "error";
      }
    }
    public String comprobar(){
        GestionDatos gd=new GestionDatos();
        if(gd.usuarioExiste(this.getUsuario())){
          resultado="Ya existe un usuario
                   con ese identificador!" ;
        }
        else{
           resultado="El nombre de usuario está libre" ;
        }
        return null;
    }
}
```

Página login.jsp

La página de login se vincula con el bean gestionado LoginBean, utilizando el componente RichFaces de tipo <a4j:poll> para lanzar peticiones AJAX de forma periódica que actualicen el mensaje de la oferta:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@taglib            uri="https://ajax4jsf.dev.java.net/ajax"
prefix="a4j" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
      HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type"
             content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <f:view>
            <h:form>
                <a4j:poll interval="3000" reRender="oferta"/>
                <small><i><h:outputText id="oferta"
                    value="#{loginBean.oferta}"/></i></small>
                <center>
                    <h1>Formulario de autenticación</h1>
                    <br/>
                    Usuario:<h:inputText value=
                          "#{loginBean.usuario}"/><br/>
                    Password:<h:inputSecret value=
                          "#{loginBean.password}"/><br/>
                    <h:commandButton action=
                     "#{loginBean.doLogin}" value="Entrar"/>
                    <br/>
                    <br/>
                </center>
                    <h:commandLink value="Registrese"
                          action="paginaregistro"/>
            </h:form>
```

```
        </f:view>
    </body>
</html>
```

Página temas.jsp

En esta página se hace uso de <a4j:support> para actualizar la tabla de libros cada vez que un nuevo tema es seleccionado en la lista:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<%@taglib uri= "http://java.sun.com/jsf/core" prefix="f"%>
<%@taglib uri= "http://java.sun.com/jsf/html" prefix="h"%>
<%@taglib uri="https://ajax4jsf.dev.java.net/ajax"
 prefix="a4j" %>
<html>
    <head>
        <meta  http-equiv="Content-Type"  content="text/html;
charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <f:view>
            <h:form >
                <center>
                    <h4>Elija tema:</h4>
                    <h:selectOneMenu
                          value="#{temasLibrosBean.idTema}">
                            <f:selectItem itemLabel="Todos"
                                 itemValue="-1"/>
                            <f:selectItems
                          value="#{temasLibrosBean.temas}"/>
                        <a4j:support event="onchange"
                          reRender="tablalibros"/>
                    </h:selectOneMenu>
                    <br/>
                    <br/>
                    <h4>Tabla de libros</h4>
                    <h:dataTable id="tablalibros" border="2"
                          value="#{temasLibrosBean.libros}"
```

```
                                    var="lib">
                                  <h:column>
                                      <f:facet name="header">
                                        <h:outputText value="titulo"/>
                                      </f:facet>
                                      <h:outputText value=
                                           "#{lib.titulo}"/>
                                  </h:column>
                                  <h:column>
                                      <f:facet name="header">
                                       <h:outputText value="paginas"/>
                                      </f:facet>
                                      <h:outputText  value=
                                           "#{lib.paginas}"/>
                                  </h:column>
                                  <h:column>
                                      <f:facet name="header">
                                           <h:outputText
                                                  value="autor"/>
                                     </f:facet>
                                      <h:outputText value=
                                           "#{lib.autor}"/>
                                  </h:column>
                              </h:dataTable>
                          </center>
                      </h:form>
                  </f:view>
              </body>
          </html>
```

Página registro.jsp

La página de registro utiliza el componente <a4j:commandButton> para consultar si existe el identificador de usuario introducido:

```
<%@ page language="java" contentType="text/html; charset=ISO-
8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<%@taglib uri="https://ajax4jsf.dev.java.net/ajax"
       prefix="a4j" %>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
      <f:view>
            <h:form>
             <center>
                   <h1>Página de registro</h1>
                </center>
                <br/>
                Usuario:<h:inputText id="cajausuario"
                    value="#{clienteBean.usuario}"/>
                <a4j:commandButton value="comprobar usuario"
                    action="#{clienteBean.comprobar}"
                    reRender="resultado"/>
                <small><h:outputText id="resultado"
                    value="#{clienteBean.resultado}"/>
                           </small><br/>
                Password:<h:inputSecret  value=
                    "#{clienteBean.password}"/> <br/>
                Email:<h:inputText value=
                    "#{clienteBean.email}"/><br/>
                Teléfono:<h:inputText value=
                    "#{clienteBean.telefono}"/><br/>
                <h:commandButton action=
                    "#{clienteBean.doRegistrar}"
                           value="Entrar"/>
            </h:form>
        </f:view>
</body>
</html>
```

Apéndice A

EL ESTÁNDAR XHTML

El estándar XHTML se fundamenta en el lenguaje de marcado HTML. HTML es un sistema de codificación basado en texto que ha sido empleado tradicionalmente para dar formato a los datos, con el fin de que puedan ser visualizados por un navegador Web.

El rápido crecimiento de la Web en los últimos 10 años ha provocado una sobreutilización de este lenguaje, añadiendo cada fabricante de navegadores sus propias etiquetas al lenguaje a fin de adaptarlo a los nuevos requerimientos de las aplicaciones. Esto provoca problemas de compatibilidad de documentos HTML entre plataformas, comportándose un mismo documento de forma diferente en cada tipo de navegador.

Para solucionar estos problemas, el W3C desarrolló a mediados del año 2000 un nuevo estándar de presentación conocido como XHTML. XHTML está basado en HTML 4.0, pero utilizando las reglas de formación de XML, lo que permite a los navegadores compatibles con este estándar poder validar los documentos antes de ser procesados y rechazar todos aquellos que no cumplan exactamente con las reglas del estándar.

A lo largo de este apéndice conoceremos las etiquetas más importantes que incluye el estándar XHTML para la construcción de páginas Web.

CARACTERÍSTICAS BÁSICAS DE XHTML

Al conjunto formado por el texto y sus correspondientes etiquetas de marcado se le conoce como documento XHTML. Estos documentos pueden ser generados dinámicamente por la aplicación de servidor o, en el caso de los documentos estáticos, estarán almacenados en archivos de texto cuya extensión será .xhtml.

Independientemente de ello, cuando el navegador recibe el documento lo interpreta secuencialmente y muestra la información contenida en el mismo de acuerdo con el formato definido por las etiquetas.

Etiquetas HTML

XHTML es un lenguaje orientado a etiquetas. Las etiquetas son elementos especiales que indican al navegador las acciones a realizar. El nombre de la etiqueta se encierra entre los símbolos “<” y “>” y pueden anidarse unas dentro de otras:

```
<p>
        <h1>texto</h1>
</p>
```

XHTML hace distinción entre mayúsculas y minúsculas, por lo que deberá respetarse el formato exacto definido por el estándar. No obstante, es bueno saber que los nombres de la gran mayoría de etiquetas y atributos se escriben en minúsculas.

TIPOS DE ETIQUETAS

Las etiquetas pueden ser de dos tipos:

- **Abiertas**. Solamente precisan aparecer una vez para realizar la acción asociada y tienen el formato <etiqueta/>. Un ejemplo es la etiqueta de salto de línea <br/>.

- **Delimitadores**. Su acción se aplica sobre un bloque de texto. El comienzo del bloque se expresa con la etiqueta entre ángulos (<p>) y el final del bloque con la etiqueta de cierre (</p>). La gran mayoría de las etiquetas XHTML pertenecen a este grupo.

Atributos

Las etiquetas pueden tener atributos. Los atributos son propiedades adicionales de la etiqueta y deben aparecen dentro de la propia etiqueta, antes del símbolo “>”. Todos los atributos siguen la sintaxis:

nombreatributo=“valor”

Comentarios

Un documento XHTML puede contener comentarios que ocupan una o varias líneas. En cualquier caso, se delimitan por los símbolos <!-- y -->. El siguiente es un ejemplo de comentario:

```
<!--Esto es un comentario
    que ocupa
    varias líneas-->
```

Estructura de un documento XHTML

Un documento XHTML tiene la estructura indicada en la figura 47.

Fig. 47. *Estructura de un documento XHTML*

Esta estructura consta de cuatro elementos principales, cuya presencia es obligatoria en todos los documentos XHTML:

- **Declaración de tipo**. Declara la versión del estándar XHTML utilizada por el documento. Es empleado por el navegador para comprobar la buena formación del mismo.

- **Elemento raíz**. Indicado mediante la etiqueta <html>, es el elemento en el que están contenidos el resto de los elementos que componen el documento. A través del atributo xmlns declara que su contenido es compatible con XHTML.

- **Cabecera**. En ella se incluye información general referida al documento. La mayor parte de las etiquetas que se incluyen dentro de este bloque se emplean para almacenar datos que ayuden a interpretar y mantener el contenido del documento.

- **Cuerpo**. Delimitado mediante la etiqueta <body>, contiene los datos que serán visualizados en el navegador con sus respectivas etiquetas de formato. El elemento <body> dispone de una serie de atributos opcionales que permiten establecer ciertas características generales de la página, entre los más importantes están:

  - **bgcolor**. Contiene el color de fondo del documento. Se define con el nombre del color en inglés o como la combinación de colores primarios siguiendo el formato: #RRGGBB, siendo RR, GG y BB el valor hexadecimal de las tonalidades de rojo, verde y azul, respectivamente.

  - **background**. Permite establecer una imagen de fondo en la página. Su valor es la cadena de caracteres con la dirección, absoluta o relativa, del archivo de imagen utilizado.

  - **text**. Define el color del texto que está incluido directamente dentro del <body>. Se utiliza el mismo sistema de colores que en *bgcolor*.

  - **class**. Nombre de la clase de estilo que se le quiere aplicar al texto contenido en <body>.

PRINCIPALES ETIQUETAS DE XHTML

A continuación, vamos a presentar las principales etiquetas que incluye el estándar XHTML para dar formato a los datos que van a ser visualizados en la ventana del navegador. Por tanto, todas las etiquetas que aquí vamos a presentar deben aparecer en el interior de la etiqueta <body>.

Todo este conjunto de etiquetas recogidas por la recomendación W3C podemos dividirlo en los siguientes grupos:

- Organización de texto
- Formato de texto
- Cabeceras
- Separadores
- Hipertexto
- Listas
- Imágenes
- Tablas
- Marcos

Organización de texto

Como su nombre indica, mediante estas etiquetas podemos organizar la distribución del texto dentro del cuerpo de la página. Entre las más importantes están:

- **
**. Se trata de una etiqueta abierta que genera un salto de línea dentro de la página.

- **<p>**. Delimita un párrafo dentro del documento insertando un salto de línea al final del mismo. Permite establecer una serie de propiedades para el párrafo a través de sus atributos. El más importante de ellos es *align*, que define la alineación horizontal del párrafo respecto a la página. Los posibles valores de este atributo pueden ser “left”, “right”, “center” o “justified”.

- **<center>**. Permite centrar los contenidos delimitados por la etiqueta. Estos contenidos pueden ser cualquier elemento HTML válido dentro del <body>.

- **<div>**. Se utiliza para delimitar un determinado bloque de etiquetas que, al igual que en el caso de <center>, puede estar constituido por cualquier elemento válido dentro del <body>. Opcionalmente, se puede utilizar el atributo align para establecer la alineación del bloque.

Formato del texto

Con estas etiquetas podemos definir el estilo con el que el texto se va a presentar en el navegador. Entre las más utilizadas están:

- **<b>**. Aplica el estilo negrita al texto delimitado.

- **<i>**. Aplica el estilo cursiva al texto delimitado.

- **<u>**. Subraya el texto delimitado.

- **<strike>**. Aplica estilo tachado al texto.

- **<big>**. Aumenta el tamaño del texto sobre el que se aplica.

- **<small>**. Disminuye el tamaño del texto.

- **<sub>**. Convierte en un subíndice el texto delimitado.

- **<sup>**. Convierte en superíndice el texto delimitado.

Encabezados

Se utilizan para establecer un formato preestablecido adecuado para ser utilizado como título de un párrafo.

Existen seis etiquetas de tipo encabezado que van de la <h1> a la <h6>. En la figura 48 podemos ver los resultados de su aplicación en una página Web.

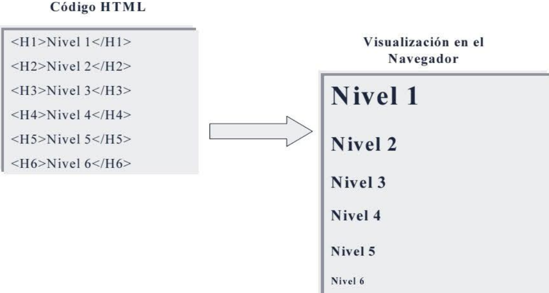


Fig. 48. *Formatos de encabezado*

Separadores

Mediante una etiqueta separador <hr/> podemos introducir líneas horizontales de separación dentro del documento. Utilizando los atributos adecuados, puede modificarse el ancho, grosor y aspecto de la línea. Dichos atributos son los siguientes:

- **size**. Define el grosor de la línea en número de píxeles.
- **width**. Define el ancho de la línea en número de píxeles.
- **align**. Establece la alineación horizontal de la línea respecto a la página. Sus valores posibles son “left”, “right” y “center”.
- **color**. Define el color de la línea.

Hipertexto

El hipertexto es la capacidad de enlazar documentos de texto diferentes o distintas partes de un mismo documento.

En un documento XHTML esta tarea la realizan los enlaces o hipervínculos. Un hipervínculo es un bloque de texto o imagen que está enlazado con otra parte del documento o con otro documento diferente, de manera que al pulsar sobre él, el navegador salta hacia ese punto.

El hipertexto se reconoce en la página porque aparece subrayado y con distinto color al resto del documento (figura 49). Situando el ratón sobre éste, el puntero se transforma en una mano.

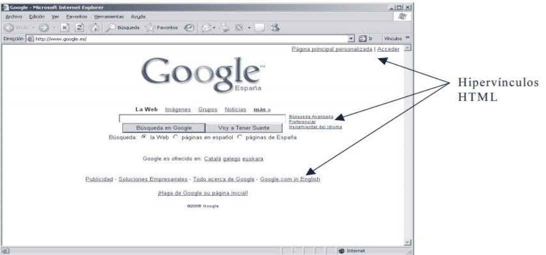


Fig. 49. *Hipervínculos en una página Web*

El bloque de texto o imagen que va a hacer de enlace se delimita con la etiqueta <A>. Esta etiqueta proporciona los siguientes atributos:

- **href**. Contiene la referencia al punto de enlace o destino. Ésta puede ser la URL relativa o absoluta de la página destino o, en el caso de un enlace local, el nombre de la marca asociada al destino:

```
<!--Enlace a la página destion.html-->
<a href="pagina1.html" >Pulsar aquí</a>
<!--Enlace a la zona del documento marca1-->
<a href="#marca1" >Pulsar aquí</a>
```

- **name**. Se utiliza para definir una marca dentro del documento. Así pues, el elemento <a> debería utilizarse también en la zona del documento donde se vaya a establecer una marca:

```
<a name="marca1">Comienzo del bloque</a>
```

Listas

Las listas facilitan la organización de la información, permitiendo su presentación como una serie de elementos que pueden venir precedidos por un número o una viñeta.

Las etiquetas utilizadas para la generación de listas son:

- **<ol>**. Define una lista numerada en donde cada elemento de la misma aparecerá precedido por un número. Dichos números se generan de forma automática y correlativa. Mediante el atributo *type* podemos definir el tipo de numeración utilizada. Entre sus posibles valores tenemos:
  - 1. Numeración estándar
  - I. Numeración romana
  - A. Numeración con letras mayúsculas
  - a. Numeración con letras minúsculas

- **<ul>**. Define una lista no numerada, donde cada elemento de la misma viene precedido por un símbolo o viñeta. Dicho símbolo puede definirse mediante el atributo *type*, cuyos posibles valores son: “disc”, “circle” y “square”.

- **<il>**. Delimita los elementos de una lista, tanto para listas numeradas como no numeradas.

La figura 50 muestra un ejemplo de utilización de listas en una página Web real, como es la del W3C.

Como se aprecia en dicha figura, los elementos de la lista pueden incluir también enlaces u otros tipos de elementos.

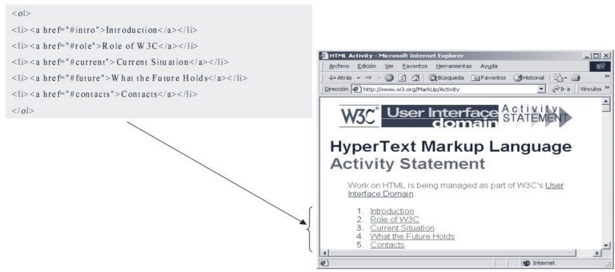
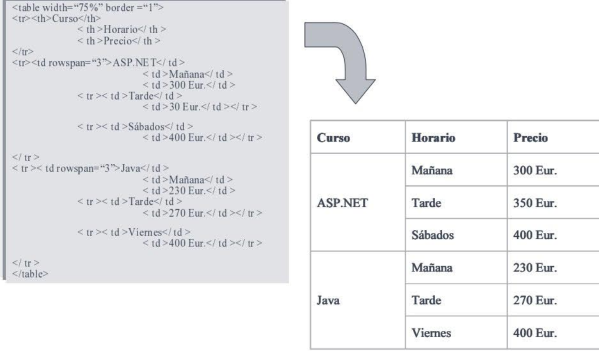


Fig. 50. *Utilización de una lista numerada*

Tablas

Las tablas es quizá el elemento más utilizado en una página Web, ya que gracias a ellas podemos dividir un área de la página en filas y columnas, cada celda resultante puede tener un ancho y un alto diferente, pudiendo contener texto, imágenes u otras tablas en su interior. La figura 51 muestra un ejemplo de utilización de tablas.



```
<table width="75%" border ="1">
<tr><th>Curso</th>
        < th >Horario</ th >
        < th >Precio</ th >
</tr>
<tr><td rowspan="3">ASP.NET</ td >
                < td >Mañana</ td >
                < td >300 Eur.</ td >
        < tr >< td >Tarde</ td >
                < td >30 Eur.</ td ></ tr >

        < tr >< td >Sábados</ td >
                < td >400 Eur.</ td ></ tr >

</ tr >
< tr >< td rowspan="3">Java</ td >
                < td >Mañana</ td >
                < td >230 Eur.</ td >
        < tr >< td >Tarde</ td >
                < td >270 Eur.</ td ></ tr >

        < tr >< td >Viernes</ td >
                < td >400 Eur.</ td ></ tr >

</ tr >
</table>
```

Curso	Horario	Precio
ASP.NET	Mañana	300 Eur.
	Tarde	350 Eur.
	Sábados	400 Eur.
Java	Mañana	230 Eur.
	Tarde	270 Eur.
	Viernes	400 Eur.

Fig. 51. *Utilización de tablas HTML*

A continuación presentamos las distintas etiquetas que se utilizan para la generación de tablas:

- **<table>**. Esta etiqueta sirve para definir una tabla XHTML. La organización de la misma vendrá determinada por otras etiquetas que deben estar incluidas dentro de <table>. Antes de analizar estas otras etiquetas, indicar que <table> proporciona una serie de atributos que permiten definir ciertos aspectos generales de la tabla. Entre éstos hay que destacar:

  - **width**. Determina el ancho de la tabla. Su valor puede ser absoluto en píxeles o relativo en porcentaje.

  - **bgcolor**. Color de fondo de la tabla.

  - **background**. Imagen de fondo de la tabla.

- **border**. Ancho de los bordes y líneas de separación de la tabla. Si no se especifica o su valor es 0 la tabla se mostrará sin bordes y sin líneas de separación entre celdas.

- **bordercolor**. Color de las líneas de separación de la tabla.

▪ **<tr>**. Delimitador de fila. Cada fila de la tabla vendrá delimitada mediante esta etiqueta.

▪ **<td>**. Cada celda dentro de una fila debe estar delimitada por la etiqueta <td>, dentro de la cual se incluirá el bloque de texto o imagen que estará contenida en ella. Una etiqueta <td> puede utilizar además los siguientes atributos:

- **colspan**. Indica el número de columnas de la tabla que debe ocupar la celda. De forma predeterminada una celda ocupa una única columna.

- **rowspan**. Indica el número de filas que debe ocupar la celda; al igual que colspan, su valor predeterminado es 1.

- **align**. Alineación horizontal del contenido de la celda.

▪ **<th>**. Al igual que <td>, delimita el contenido de una celda, aunque asignándole un formato predefinido. Se emplea para aquellas celdas que van a servir de título de una fila o columna de la tabla.

Imágenes

Una imagen puede aparecer en cualquier parte del cuerpo del documento, incluso alineada con un párrafo. La etiqueta <img> y sus atributos permiten insertar una imagen en un documento y configurar los distintos aspectos de la misma. Entre los principales atributos de esta etiqueta, tenemos:

▪ **src**. Dirección del archivo que contiene la imagen. Puede ser una dirección relativa o absoluta.

▪ **alt**. Texto alternativo que aparecerá en lugar de la imagen si ésta no se localiza.

- **height**. Alto de la imagen en píxeles.

- **width**. Ancho de la imagen en píxeles.

Frames

Los frames se utilizan para dividir la ventana del navegador en zonas o frames. Cada una de estas zonas albergará una página Web diferente.

Las etiquetas para la división de la ventana en frames deberán especificarse en el interior de la etiqueta <head> dentro de un documento XHTML sin cuerpo, es decir, en el que la etiqueta <body> estará vacía.

Estas etiquetas son:

- **<frameset>**. Permite establecer las áreas de división, el número y dimensión de éstas vienen especificados mediante los siguientes atributos:

    - **cols**. Indica el ancho de cada una de las columnas en que se dividirá la ventana, especificando los valores en píxeles o en porcentaje y separando unos de otros mediante una coma. Por ejemplo, la cadena "200,100,150", establece tres columnas de 200, 100 y 150 píxeles, respectivamente.

    - **rows**. Indica el alto de cada una de las filas en que se dividirá la ventana, siguiendo el mismo formato que *cols*.

    Los elementos <frameset> pueden anidarse a fin de conseguir una división irregular de la ventana.

- **<frame>**. Especifica el contenido de cada área o frame. A través de su atributo src indicamos la dirección de la página, que debe ser cargada en el frame correspondiente.

    La figura 52 nos muestra un ejemplo de una página para la división de una ventana en tres frames.

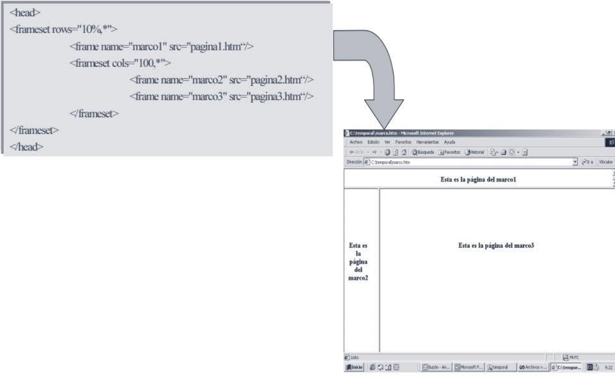


Fig. 52. *División irregular de una ventana*

FORMULARIOS HTML

El estándar XHTML proporciona también un conjunto de etiquetas para la generación de elementos gráficos que permitan capturar datos de usuario en una página Web y enviarlos a la capa intermedia para su procesamiento.

El formulario HTML

Para que la información proporcionada por el usuario a través de los controles gráficos pueda ser enviada a la aplicación que los va a procesar, es necesario que estos controles estén definidos dentro de lo que se conoce como un formulario HTML. Éste no ofrece ningún aspecto físico determinado en la página donde se encuentra; tan sólo tiene como misión la de englobar los controles encargados de recoger la información.

Un formulario se define mediante la etiqueta <form>. A través de los siguientes atributos se indican algunas características del mismo:

- **action**. Indica la URL de la aplicación o proceso que recogerá y procesará los datos. En el caso de que la página y el proceso formen parte de la misma aplicación Web, se deberá especificar en este atributo la URL relativa asociada al mismo.

- **method**. Indica el método de envío de los datos; su valor puede ser “GET” o “POST”. Si el valor es “GET”, los datos serán enviados en la cabecera de la petición como parte de la URL, mientras que si es “POST” la información viajará en el cuerpo de la petición.

Para que el formulario pueda hacer el envío de los datos, es necesario que cuente con un control tipo submit. Este control tiene la forma de un botón de pulsación. Cuando es activado, genera automáticamente la petición de la aplicación del servidor y le envía los datos de los controles incluidos en el mismo.

Los controles HTML

El lenguaje HTML proporciona una serie de etiquetas con sus respectivos atributos para la generación de controles gráficos. Vamos a exponer a continuación los más utilizados.

Antes de ello, hay que comentar que todas las etiquetas de controles disponen de un atributo llamado name. Este atributo permite asignar un nombre único al control de cara al envío de su contenido al servidor cuando se produzca la petición. Así, por cada control incluido en el formulario,, se enviará un dato con un nombre y un valor: el nombre será el indicado en la propiedad *name* del control correspondiente y el valor será el contenido del control.

CONTROL TEXT

Representa una caja de texto donde el usuario puede escribir cualquier cadena de caracteres en una línea. Este control se genera mediante la etiqueta <input>, pudiendo utilizar los siguientes atributos:

- **type**. Representa el tipo de control que tiene que generar la etiqueta. Como veremos, <input> puede utilizarse para generar otros controles, además de cajas de texto, por lo que deberá especificarse a través de este atributo el tipo de control deseado. Para una caja de texto, el valor de *type* debe ser “text”.

- **value**. Representa el texto contenido en el control. Si se especifica un valor, éste aparecerá inicialmente en el control.

CONTROL TEXTAREA

Consiste en una caja de texto multilínea. Para generar este control debemos utilizar la etiqueta delimitadora <textarea>. Si el control debe contar con un valor inicial, deberá especificarse entre <textarea> y </textarea>.

En cuanto a los principales atributos, tenemos:

- **cols**. Ancho del control en número de caracteres.
- **rows**. Alto del control en número de líneas.

CONTROL PASSWORD

Es similar a una caja de texto, con la diferencia de que los caracteres introducidos no se visualizan en la misma, mostrándose en su lugar un "*" por cada carácter. Es ideal para solicitud de contraseñas en una página de validación de usuarios.

Al igual que TextBox, se genera con la etiqueta <input>, especificando en el atributo *type* el valor "password". Dispone también del atributo *value* para establecer un valor inicial en el control.

CONTROL SUBMIT

Se trata de un control que tiene el aspecto de un botón de pulsación. Como ya se ha indicado anteriormente, cuando este botón es pulsado, se produce una solicitud HTTP de la página o aplicación indicada en el atributo *action* del formulario, enviándole los datos introducidos en los controles incluidos en él.

Este control se genera mediante la etiqueta <input>, especificando el valor "submit" en el atributo *type*.

El atributo *value* se utiliza en este control para especificar el texto mostrado en el interior del botón.

CONTROL BUTTON

Su aspecto es idéntico al control submit sólo que, a diferencia de éste, la pulsación del botón no genera ninguna acción predeterminada. Cualquier acción personalizada que queramos asociar al evento click del botón HTML deberá ser programada mediante un script de cliente.

Para generarlo utilizaremos la etiqueta <input> con el valor "button" en el atributo *type*.

CONTROL CHECKBOX

Representa una casilla de verificación. Estos controles se utilizan para que el usuario pueda marcar determinadas opciones que se le presentan en la página.

Para generarlo utilizaremos la etiqueta <input>, especificando en el atributo *type* el valor “checkbox”.

Entre los atributos de <input> asociados a este control tenemos:

- **value**. En este caso, *value* representa una cadena de texto asociada al control que no será mostrada en el formulario, sino que será enviada al servidor como valor del control cuando se produzca la petición. Sólo si la casilla de verificación se encuentra activada, su valor será enviado en la petición.

- **checked**. Permite establecer el estado inicial del control, si se asigna el valor “checked” a este atributo el control aparecerá activado, si el atributo no se utiliza el control permanecerá desactivado.

CONTROL RADIO

En este caso se trata de un tipo de control en forma circular que se utiliza para que el usuario pueda seleccionar entre opciones excluyentes entre sí, de modo que sólo pueda estar activada una en cada momento.

Para su generación, utilizaremos también la etiqueta <input> con el valor “radio” en el atributo *type*.

Para que las opciones representadas por un determinado grupo de botones de radio sean excluyentes entre sí, desactivándose las restantes opciones cada vez que una de ellas es activada, es necesario que todos los controles <input> asociados tengan el mismo valor de atributo *name*.

El significado de los atributos *value* y *checked* es exactamente el mismo que para el control CheckBox.

CONTROL SELECT

Este último control representa una lista de opciones, lista que puede ser desplegable o siempre abierta.

Para generarlo utilizaremos la etiqueta <select>, dentro de la cual deberán especificarse cada una de las opciones de la lista delimitadas por una nueva etiqueta llamada <option>.

En cuanto a los atributos de la etiqueta <select>, además de *name* tenemos:

- **size**. Determina el alto del control en número de filas. Si su valor es 1 (valor por defecto), el control se presentará como una lista desplegable, mientras que si es >1 aparecerá como una lista abierta en la que el número de filas visibles será el indicado en size.

- **multiple**. Si el atributo se establece al valor “multiple”, la lista permitirá la selección de más de un elemento. Si el atributo no se utiliza, sólo se podrá seleccionar un único elemento.

Por otro lado, cada elemento <option> debe incluir un atributo *value* que permita asociar una cadena de texto a cada opción (figura 53). Cuando se produzca el submit del formulario, en el caso de la lista se enviará un dato con el nombre especificado en el atributo name de <select>, cuyo valor será el contenido del atributo *value* del elemento <option> seleccionado.

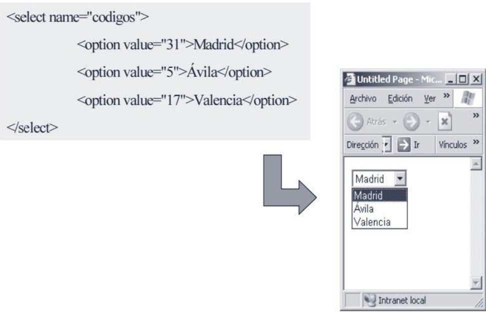


Fig. 53. *Generación de una lista HTML*

HOJAS DE ESTILO CSS

Aunque, como hemos visto, el estándar XHTML proporciona un gran número de etiquetas para dar formato a la información de una página, las opciones de estilo son bastante limitadas para los requerimientos de las aplicaciones actuales.

Como solución al problema del estilo, el W3C desarrolló un estándar bajo el nombre de *Cascading Style Sheet (CSS),* que tiene como objetivo poder definir

hojas de estilo para ser aplicadas sobre un documento HTML, permitiendo separar los datos de la información de formato y estilo de los mismos.

Las hojas de estilo CSS se basan en la definición de una serie de propiedades de estilo asociadas a determinados tipos de etiquetas e independientes de su contenido, de forma que, al ser utilizadas estas etiquetas en el documento, se apliquen automáticamente las diferentes propiedades y opciones de estilo definidas para las mismas.

Tipos de hojas de estilo

En función de cómo se defina una hoja de estilo y cómo se utilice en un documento, podemos distinguir tres tipos o formas de crearlas:

- Hojas de estilo externas
- Hojas de estilo internas
- Hojas de estilo en línea

HOJAS DE ESTILO EXTERNAS

Una hoja de estilo externa consiste en definir, en un archivo de texto con extensión .css, los valores de las propiedades de estilo que van a ser aplicados a las distintas etiquetas de un documento.

La definición de los valores de propiedades de estilo sigue el formato:

etiqueta {propiedad1:valor1;propiedad2:valor2...}

Para aplicar la hoja de estilo sobre un determinado documento, éste deberá vincularse al archivo .css utilizando la etiqueta <link> en su cabecera. Dicha etiqueta tiene el siguiente formato:

<link rel="stylesheet" href="fichero.css" type="text/css"/>

En la figura 54 se muestra un ejemplo de hoja de estilo definida en un archivo externo y su vinculación a un documento XHTML.

La principal ventaja que ofrecen las hojas de estilo externas es que una misma hoja de estilo puede ser utilizada por varios documentos.

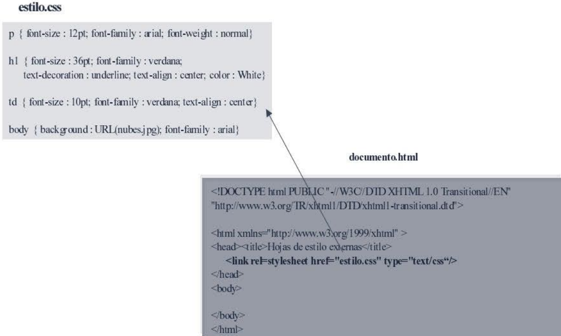


Fig. 54. *Hoja de estilo externa y aplicación en un documento*

HOJAS DE ESTILO INTERNAS

Las hojas de estilo internas se definen en el interior del documento donde van a ser utilizadas, concretamente, en la cabecera del mismo en el interior de la etiqueta <style>.

El formato utilizado en la definición de los valores de las propiedades de estilo es exactamente igual que en las hojas de estilo externas, tal y como queda ilustrado en el ejemplo que aparece en la figura 55.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

<head><title>Ejemplo 83: Hojas de estilo internas</title>
   <style>
      p { font-size : 12pt; font-family : arial; font-weight : normal}

      h1 { font-size : 36pt; font-family : verdana;
         text-decoration : underline; text-align : center; color : White}

      td { font-size : 10pt; font-family : verdana; text-align : center}

      body { background : URL(nubes.jpg); font-family : arial}
   </style>
</head>
<body>

</body>
</html>
```

Fig. 55. *Definición de una hoja de estilo interna*

HOJAS DE ESTILO EN LÍNEA

En muchas circunstancias nos interesará aplicar sobre una determinada etiqueta de la página una serie de propiedades de estilo, sin que esto afecte al resto de etiquetas de su mismo tipo que estén definidas en el documento.

Para ello, debemos definir los valores de estas propiedades de estilo dentro del atributo style de la etiqueta en la que se quieren aplicar (figura 56). La sintaxis para la definición de las propiedades será la misma que utilizamos en los casos anteriores.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

<head><title>Hojas de estilo en linea</title>
</head>
<body>
    <p style="font-size : 12pt; font-family : arial; font-weight : normal">
            Parrafo con estilo
    </p>
    <p>
            Parrafo sin estilo
    </p>
</body>
</html>
```

Fig. 56. *Definición de una hoja de estilo interna*

Las hojas de estilo internas representan también un mecanismo para alterar, en una etiqueta particular, determinadas propiedades definidas en una hoja de estilo interna o externa, pues los estilos que se definan en el interior de una etiqueta particular anularán a los definidos en hojas de estilo de ámbito superior.

Apéndice B

XML

El XML está cada vez más presente de una u otra manera en multitud de aplicaciones informáticas.

Para aquellos lectores no conocedores de este estándar, el presente apéndice pretende proporcionar una visión general sobre el mismo que permita afrontar sin dificultad aquellos capítulos donde se hace uso de XML, aclarando los conceptos más importantes en los que se basa.

FUNDAMENTOS SOBRE XML

¿Qué es XML?

Podríamos definir XML como un estándar para describir información de forma estructurada y jerárquica mediante la utilización de etiquetas personalizadas.

Estas etiquetas no están prefijadas por el estándar, sino que es el propio creador del documento el que las "inventa" en función del tipo de información que quiere describir.

En el ejemplo de la figura 57 podemos ver la información correspondiente a los datos de un curso y como el diseñador del documento utiliza las etiquetas <curso>, <titulo>, <duracion> y <comentarios> para su descripción.

```
<curso>
        <titulo>Programación con XML</titulo>
        <duracion>20 horas</duracion>
        <comentarios>Es necesario conocer XML</comentarios>
</curso>
```

Fig. 57. *Documento XML de ejemplo*

En XML las etiquetas no tienen un significado predefinido, sólo sirven para marcar los datos dejando "libertad" a las aplicaciones receptoras del documento para que interpreten y manipulen los datos según sus propias necesidades. Por ejemplo, una aplicación que lea este documento de ejemplo podría utilizar estos datos para almacenarlos en una base de datos de cursos, mientras que otra los utilizaría para generar un informe para un posible alumno.

Documentos XML

En el punto anterior apareció la expresión "documento XML". Pues bien, al conjunto formado por los datos referentes a la información que se quiere describir junto con sus respectivas etiquetas de marcado se le conoce como **documento XML**.

Un documento XML es un bloque de texto que puede ser tratado desde cualquier editor ASCII, de hecho, pueden ser almacenados en disco como archivos de texto con extensión .xml (figura 58) y puede incluir cualquier flujo de datos basado en texto: un artículo de una revista, un resumen de cotizaciones de Bolsa, un conjunto de registros de una base de datos, etc.

Cursos.xml

Fig. 58. *Documento XML en un archivo de texto .xml*

¿Por qué XML?

Hasta ahora, Internet ha sido utilizado como medio para acceso y presentación de información, basándose en un diálogo entre el servidor de información y el elemento de presentación (navegador).

Aunque este esquema ofrece muchas posibilidades (consulta de información, compra por Internet, etc.), presenta muchas limitaciones desde el punto de vista de los datos, ya que éstos están codificados en un formato (HTML) que sólo es entendible por un determinado tipo de aplicación que es el navegador.

Para que la Web pueda explotar todo su potencial, es necesario disponer de un estándar que permita representar información estructurada en la Web, de modo que pueda ser interpretada y manipulada por diversos tipos de aplicaciones y dispositivos sin la intervención de un usuario. Tal y como queda expresado en la figura 59, **XML es ese estándar**.

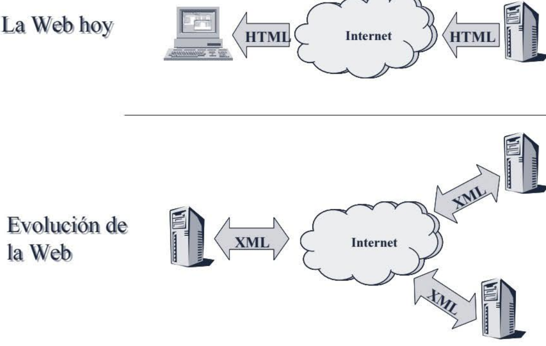


Fig. 59. *Intercambio universal de datos en la Web con XML*

XML vs. HTML

Cuando se habla de XML puede surgir una confusión al pensar que pueda tratarse de una especie de "HTML mejorado". Pues bien, no es así, aunque se parecen (ambos se basan en la utilización de etiquetas), cada tecnología ha sido desarrollada con un objetivo diferente.

Mientras que HTML está orientado a la presentación de datos en un navegador Web, XML es más genérico al utilizar las etiquetas simplemente para delimitar piezas de datos, dejando la interpretación de estos datos a la aplicación que lee el documento.

Fig. 60. *Diferencia de uso entre XML y HTML*

CARACTERÍSTICAS DE XML

Como se ha comentado anteriormente, XML se presenta como el estándar para el intercambio de información entre aplicaciones en la Web. Ello es posible gracias a las características que presenta esta tecnología, y que se pueden resumir en tres puntos clave:

- Comprensible
- Basado en texto
- Independiente

Comprensible

Un documento XML permite describir la información de manera que ésta sea fácilmente comprensible, no sólo por una persona que lea el documento, sino, lo que es más importante, por las posibles aplicaciones que lo van a procesar.

Para cualquier programa resultará más fácil extraer los datos del pedido en el documento XML mostrado en el ejemplo de la figura 61, que del otro documento en el que no se utilizan etiquetas de marcado para identificar los datos.

Documento en formato XML:

```
<pedido>
        <material>Monitor VGA</material>
        <codigo_pedido>A-45T</codigo_pedido>
        <tel_contacto>
                <fijo>916378901 </fijo>
                <movil>932456712 </movil>
        </tel_contacto>
</pedido>
```

```
Monitor VGA A-45T 916378901 932456712
```

Fig. 61. *Comparación entre documento XML y documento sin marcado*

Basado en texto

Los documentos XML están basados en texto, esto significa que no es necesario disponer de herramientas específicas de ningún fabricante para crear, interpretar y manipular la información.

Aquí podemos ver el documento XML del ejemplo anterior y la misma información en formato Microsoft Word (figura 62). La diferencia es clara; mientras que el documento XML podrá ser interpretado por cualquier aplicación, el documento Word sólo puede ser interpretado si se dispone de la aplicación específica Microsoft Word.

Documento en formato XML:

```
<pedido>
        <material>Monitor VGA</material>
        <codigo_pedido>A-45T</codigo_pedido>
        <contacto>
                <telefono>916378901 </telefono>
                <e-mail>futura@dta.com </e-mail>
        </contacto>
</pedido>
```


```
ÐÏ□à¡±□á > □ þÿ □ □ ! □ # □ þÿÿÿ □ □ +□ □ bjbjýÏýÏ □ □.□ Ÿ¥ Ÿ¥ ( □ ÿÿ□
ÿÿ□ ÿÿ□ l ² ² ² ² □□ □□ □□ □ □□ \□ \□ \□ \□ □ h□ □ □□ □ □ €□ €□ €□ €□
€□ _□ _□ _□ □□ □ □□ □□ □□ □□ □□ □□ $ □ □ □ N ¥□ □ □□ _□ [□ □ _□
_□ _□ ¥□ □□ ² ² €□ €□ Û º□ □ □□ □□ □□ _□ □ ² 8 €□ □□ €□ □□ □□
_□ □□ □□ □ □□ □□ ê □ □□ □□ ˆ 0 € €š 0 € €ˆ 0 € € □ +□ □ □
+□ □ □ *□ □ □ ð8 □ð□ □□ □ □ □ □ □ @□ñ□ ÿÿ ÿ €€€ ÷ □ □ □ð' □ □ð□ □ □□ □ □ð0 □ □ð( □ ð□
□ ð□ □ □ □ □ðB □ ð□ □□ □
```


Fig. 62. *Diferencia entre texto XML y otros formatos*

Independiente

XML es una tecnología desarrollada por un organismo internacional e independiente conocido como W3C (World Wide Web Consortium). Se puede encontrar más información sobre esta organización en la dirección *http://www.w3.org/*.

El W3C ha desarrollado una serie de estándares abiertos para ayudar a los programadores en el desarrollo de aplicaciones para XML.

Apoyándose en estos estándares, los desarrolladores pueden utilizar lenguajes y herramientas de desarrollo de diversos fabricantes, para implementar aplicaciones que generen, manipulen e interpreten documentos XML. Esto posibilita que un documento generado por una aplicación escrita con un determinado lenguaje pueda ser interpretado por otra aplicación basada en una tecnología diferente.

APLICACIONES DEL XML

XML encuentra multitud de aplicaciones en el mundo de la programación, especialmente en el entorno Web. El intercambio de datos entre aplicaciones es, sin duda alguna, la principal utilidad de XML, pero no la única. Entre las principales aplicaciones de XML podríamos destacar las siguientes:

- Intercambio de datos entre aplicaciones (B2B)
- Almacenamiento intermedio de datos en aplicaciones Web
- Presentación en la Web
- Utilización como base de datos

Intercambio de datos entre aplicaciones (B2B)

El comercio electrónico entre empresas o Business to Business es uno de los servicios más útiles y con más futuro que ofrece Internet, y donde XML encuentra su principal aplicación.

El Business to Business consiste en la posibilidad de realizar transacciones comerciales entre compañías a través de la Web, las cuales se llevan a cabo mediante el intercambio de mensajes entre aplicaciones. Estos mensajes son codificados mediante documentos XML.

La posibilidad de representar información estructurada en la Web y transmitirla entre aplicaciones, independientemente de la plataforma, hace de XML la tecnología ideal para codificar transacciones comerciales como órdenes de compra, petición de productos, etc.

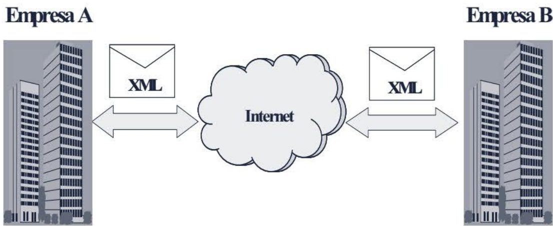


Fig. 63. *Intercambio de datos con XML*

Almacenamiento intermedio en aplicaciones Web

Los datos almacenados en una base de datos se encuentran en un formato específico del fabricante. XML proporciona un soporte neutro y único para los datos, pudiéndolo transformar después en diferentes formatos dependiendo del medio al que se destina la información.

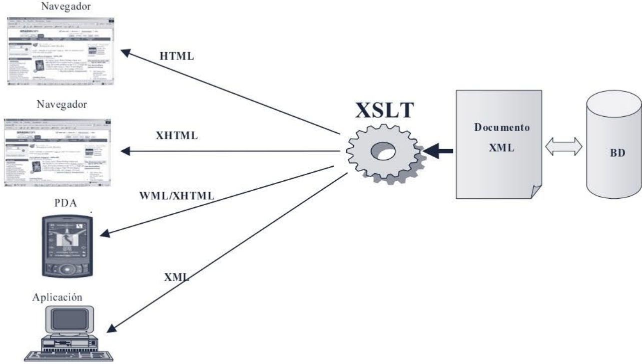


Fig. 64. *Aplicaciones multidispositivo*

Por ejemplo, es posible generar a partir de un único documento XML documentos HTML en diversas versiones, documentos en formato para terminales móviles o, simplemente, transformarlo en otro documento XML (figura 64).

Presentación en la Web

Como hemos comentado anteriormente, XML, a diferencia de HTML, no está orientado a la presentación.

No obstante, XML también puede aportar mucho en este campo, de hecho, se han desarrollado diversos estándares basados en XML encaminados a la presentación de información en diferentes tipos de terminales, uno de ellos, XHTML, ha sido presentado en el apéndice A de este libro.

Fig. 65. *Lenguajes de presentación XML*

Utilización como base de datos

XML no pretende ser una alternativa a los sistemas de almacenamiento de datos tradicionales, de hecho, XML nunca podrá competir con estos sistemas en aspectos tales como rendimiento, seguridad, integridad de los datos, etc.

Sin embargo, en ciertos tipos de información con varios niveles de anidamiento, su almacenamiento en una base de datos relacional puede ser bastante complicado, siendo necesaria la utilización de múltiples tablas relacionadas para almacenar una cantidad pequeña de datos. Es en estos casos donde suele ser más recomendable utilizar XML en vez de una BD (figura 66).

Para acceder a esta información, las aplicaciones no necesitarán utilizar gestores de datos propios de ningún fabricante, tan sólo los estándares definidos por el W3C.

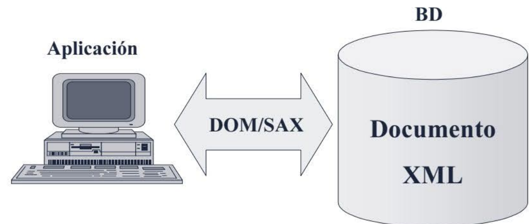


Fig. 66. *Almacenamiento de datos en documentos XML*

TECNOLOGÍAS BASADAS EN XML

De todo lo anterior se desprende que las utilidades que se le pueden dar al XML son prácticamente ilimitadas. Sin embargo, nada de ello sería posible sin la existencia de toda una serie de tecnologías sobre las que se apoya este estándar, estas tecnologías están representadas en la pirámide de la figura 67.

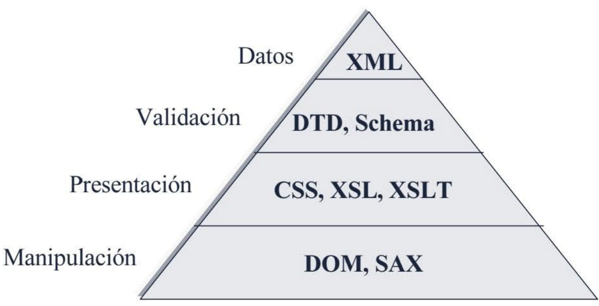


Fig. 67. *Tecnologías XML*

En las siguientes secciones comentaremos brevemente las características de estas tecnologías, comenzando por el estudio, desde el punto de vista sintáctico, del propio XML.

Construcción de documentos XML

Después de esta introducción al XML vamos a ver cómo se construye un documento XML y qué consideraciones hay que tener al respecto.

ESTRUCTURA DE UN DOCUMENTO XML

Anteriormente, hemos visto algunos ejemplos de documentos XML. A partir del documento de la figura 68, vamos a analizar con detalle la estructura de un documento XML y sus componentes fundamentales.

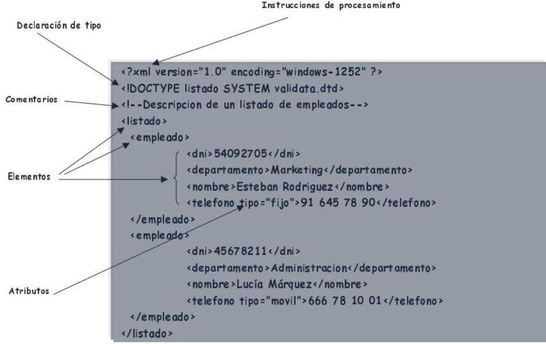


Fig. 68. *Estructura de un documento XML*

Éstos son:

- **Elementos**. Un elemento representa una pieza lógica del marcado, se representa con una cadena de texto (dato) encerrada entre etiquetas que pueden contener también atributos. Pueden existir elementos vacíos (<br/>).

- **Instrucciones de procesamiento**. Se trata de órdenes especiales para ser utilizadas por la aplicación que procesa el documento XML. Comienzan por <? y terminan por ?>. La instrucción de

procesamiento que aparece en este ejemplo se llama **Declaración XML** y, aunque no es obligatorio que aparezca, suele ser conveniente ponerla porque, además de identificar el documento como texto XML, aporta información relativa al mismo como versión de XML, codificación de caracteres, etc.

- **Comentarios**. Información que no forma parte del documento. Comienzan por <!-- y terminan por -->.

- **Declaraciones de tipo**. Especifican información acerca del vocabulario utilizado por el documento:

  <!DOCTYPE persona SYSTEM "persona.dtd">

- **Atributos**. Los atributos indican características adicionales de un elemento. Se escriben dentro de la etiqueta que define al elemento antes del carácter ">". Los valores de los atributos deben escribirse entre comillas.

REGLAS SINTÁCTICAS XML

Las normas sintácticas para la construcción de documentos XML están recogidas en la recomendación 1.0 del W3C. A continuación, presentamos algunos de los aspectos clave de esas normas:

- XML realiza distinción entre mayúsculas y minúsculas. Esto afecta a todos los componentes del documento, esto significa que no se puede escribir doctype, sino DOCTYPE, o que los elementos <persona> y <Persona> no se consideran iguales.

- Los nombres de elementos y atributos no pueden contener espacios ni signos de puntuación.

- Los elementos deben estar correctamente anidados.

- Todos los documentos deben tener un elemento principal que contenga a los demás.

- Los valores de los atributos deben estar escritos entre comillas.

- Un documento puede tener elementos vacíos, éstos se representan por: <elemento/>.

DOCUMENTOS BIEN FORMADOS

Un documento bien formado es aquel que cumple con la sintaxis de XML. Cuando una aplicación recibe un documento XML, es necesario que compruebe que el documento está bien formado antes de procesarlo.

En el ejemplo de la figura 69 tenemos un documento bien formado y el mismo documento con algunos errores, como la omisión de la etiqueta de cierre *</codigo_pedido>* y la anidación incorrecta de la etiqueta *</e-mail>*.

Bien formado

```
<?xml version="1.0"?>
<pedido>
   <material>Monitor VGA</material>
   <codigo_pedido>A-45T</codigo_pedido>
    <contacto>
            <telefono>916378901 </telefono>
            <e-mail>futura@dta.com </e-mail>
   </contacto>
</pedido>
```

Mal formado

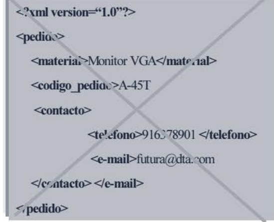


Fig. 69. *Documentos bien y mal formados*

Apéndice C

EL LENGUAJE JAVASCRIPT

Prácticamente desde los comienzos de la www, los creadores de páginas Web se dieron cuenta de las limitaciones que presentaba el lenguaje HTML para la creación de páginas interactivas que fueran atractivas para el usuario. En este contexto, surgieron los lenguajes de script como medio para añadir dinamismo a una página Web.

Un script es un bloque de código que se incluye directamente en el documento HTML y que puede ser interpretado por el navegador. Mediante estos scripts se pueden incluir instrucciones de código que respondan a acciones de usuario y que sean capaces de modificar dinámicamente el aspecto de las páginas.

Los lenguajes de script se caracterizan por poseer un reducido conjunto de instrucciones, a fin de reducir lo más posible la complejidad del intérprete de script del navegador y, de esta manera, ser accesible a un mayor número de clientes.

De todos los lenguajes de script existentes, es, sin duda alguna, JavaScript el que se ha ido imponiendo al resto, debido fundamentalmente a su soporte en prácticamente todos los modelos de navegadores que actualmente se utilizan para acceder a Internet.

Durante este apéndice presentaremos las características sintácticas de este lenguaje, así como los principales objetos en los que se apoya.

JAVASCRIPT EN DOCUMENTOS XHTML

Un script de JavaScript, o de cualquier otro lenguaje, se incluye dentro de la página HTML delimitándolo por la etiqueta <script>, la cual puede aparecer tantas veces como sea necesario en cualquier parte del documento (figura 70).

Para que el navegador reconozca que el script está escrito con JavaScript, el atributo language de <script> debe estar establecido al valor javascript.

```
<html>
<head>
            <script language="javascript">
            <!--
            //aquí código de javascript
            //-->
            </script>

</head>
<body>
            <script language="javascript">
            <!--
            //aquí código de javascript
            //-->
            </script>
</body>
</html>
```

Fig. 70. *Inclusión de bloques de script en XHTML*

Aunque el código puede situarse en cualquier parte dentro de la etiqueta, normalmente se agrupa en funciones. Gran parte de estas funciones se ejecutarán como respuesta a eventos sobre la interfaz.

SINTAXIS DEL LENGUAJE

A continuación, vamos a analizar las diferentes estructuras sintácticas del lenguaje JavaScript.

Sintaxis básica

Para empezar, debemos tener en cuenta las siguientes consideraciones:

- JavaScript hace distinción entre mayúsculas y minúsculas.

- Las sentencias terminan con ;
- Para comentarios de una línea se utiliza //
- Los comentarios de más de una línea se encierran entre /* y */.
- Los bloques de instrucciones se delimitan con { y }.

Tipos de datos y variables

JavaScript soporta los siguientes tipos de datos:

- **Cadena**. Conjunto de caracteres delimitado por " ".
- **Numérico**. Números enteros y decimales.
- **Boolean**. Representa dos valores posibles (*true* y *false*).
- **Objeto**. Representa cualquier objeto válido predefinido en JavaScript.

La figura 71 muestra un ejemplo de uso de cada uno de estos tipos.

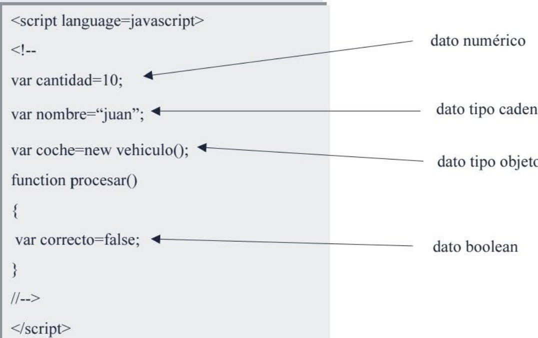


Fig. 71. *Tipos de datos JavaScript*

Como sucede con cualquier otro lenguaje de programación, los datos en JavaScript se manejan a través de variables.

Las variables se declaran con var seguido del nombre de la variable, cuyo primer carácter debe ser una letra o carácter de subrayado (_) y no debe contener espacios ni coincidir con palabras reservadas de JavaScript. Se hace distinción entre mayúsculas y minúsculas:

```
var mivariable;

var v1, variable_nueva;
```

Las variables no tienen tipo, pudiendo almacenar cualquiera de los tipos básicos manejados por JavaScript.

Cuando una variable se declara, adquiere por defecto el valor indefinido (*undefined*), aunque puede ser inicializada explícitamente a cualquier valor en la misma instrucción de declaración:

```
var contador=1;
```

En cuanto al ámbito de acceso a una variable, éste puede ser:

- **Global**: la variable se declara fuera de cualquier función, donde es accesible desde cualquier línea de código que exista en la página.
- **Local**: la variable se declara dentro de un bloque, sólo es accesible en el interior de ese bloque.

Operadores

La tabla de la figura 72 muestra la lista de los operadores que soporta el lenguaje JavaScript, clasificados según su tipo.

Además de éstos, JavaScript dispone de los siguientes operadores especiales:

- **new**. Se utiliza para crear objetos a partir de su clase.
- **typeof**. Al aplicarlo sobre un determinado dato devuelve como resultado el tipo JavaScript del mismo. Los posibles valores que se pueden obtener al aplicar este operador son: *number*, *object*, *boolean*

o *string*. Si se aplica sobre una variable que no haya sido inicializada devolverá como resultado undefined.

Tipo	Operador
Aritméticos	+, -, *, /, %, ++, --
Lógicos	&&, ||, !
Comparación	==, !=, >, <, >=, <=
Condicionales	(condicion) **?** valor1 **:** valor2
Asignación	=, +=, -=, *=, /=

Fig. 72. *Tabla de operadores JavaScript*

Instrucciones de control

Seguidamente, vamos a analizar las características sintácticas del juego de instrucciones de control de JavaScript.

IF

Comprueba una determinada condición, si ésta es cierta (*true*) se ejecutan las sentencias del bloque (las llaves sólo son obligatorias si hay más de una sentencia en el bloque), si la condición no se cumple, se ejecutan las sentencias indicadas en else, cuyo uso es opcional.

Su formato es el siguiente:

```
if(condicion){

    Instrucciones

}

else{

    Instrucciones

}
```

La condición puede ser una expresión que devuelva cualquier tipo válido JavaScript. En este sentido hay que tener en cuenta que cualquier valor distinto de 0 e indefinido (*undefined*) será evaluado como verdadero.

Por ejemplo, el siguiente script mostraría un cuadro de diálogo con el mensaje "variable no definida":

```
var c;
if(!c){
      alert("variable no definida");
}
```

SWITCH

Evalúa una expresión que puede dar lugar a múltiples resultados posibles:

```
switch (expresión){

  case valor1:

     sentencia(s);

     break;

  case valor2:

     sentencia(s);

     break;

    ……...

  [default:]

       sentencia(s);

 }
```

El valor de la expresión se compara con cada una de las constantes de la sentencia case, si coincide alguno, se ejecuta el código que le sigue, abandonando la instrucción *switch* en el momento en que se encuentre la sentencia *break*.

Si la expresión no coincide con ningún case se ejecutará el bloque *default* en caso de estar presente.

El siguiente ejemplo muestra el nombre del día laborable a partir de su posición.

```
var numero=prompt("Introduzca número de día");
switch (numero){
  case 1:
     alert("es lunes");
     break;
  case 2:
     alert("es martes");
     break;
  case 1:
     alert("es miércoles");
     break;
  case 1:
     alert("es jueves");
     break;
  case 1:
     alert("es viernes");
     break;
  default:
     alert("no laborable");
}
```

FOR

Se trata de una estructura repetitiva que permite ejecutar un conjunto de sentencias un número determinado de veces, mientras que se cumpla una determinada condición (normalmente numérica):

for(inicialización;comparación;incremento)

{

Instrucciones

}

Los pasos de ejecución de un bucle for son los siguientes:

1. Se ejecuta la instrucción de inicialización, inicializando todos los valores necesarios para el control del bucle.

2. Se realiza la comparación, si la condición es cierta se ejecutan las sentencias del bucle, en caso contrario se sale del bucle y se realizará la primera instrucción que aparezca a continuación.

3. Se ejecuta incremento y se vuelve al paso 2, realizándose de nuevo la comparación.

El siguiente bloque mostraría la suma de todos los números del 1 al 10:

```
var suma=0;
for(var i=0;i<10;i++)
{
        suma+=i;
}
alert ("La suma es: "+ suma);
```

WHILE

Se trata de una estructura repetitiva que ejecuta las instrucciones que se indican en el bloque un número indefinido de veces, mientras se cumpla una condición:

```
while(condicion)

{

        Instrucciones

}
```

La secuencia de ejecución de un bucle *while* es la siguiente:

1. Se evalúa la condición, si es cierta se entra en el bucle, en caso contrario se pasa el control a la instrucción situada a continuación.

2. Dentro del bucle se ejecutan las instrucciones que aparecen entre llaves.

3. Vuelve otra vez al paso 1 y se evalúa de nuevo la condición.

Existe otra versión que permite ejecutar primero el bloque de sentencias y después evaluar la condición:

do {

Instrucciones

}while(condicion);

Obsérvese el punto y coma (;) para finalización de la instrucción.

El siguiente ejemplo realiza la suma de números mientras ésta sea menor de 1000:

```
var suma=0, i;
while(suma<1000)
{
        suma+=i;
        i++;
}
```

LAS SENTENCIAS BREAK Y CONTINUE

Si en cualquier momento necesitamos interrumpir la ejecución de un bucle *for* o *while*, podemos utilizar las instrucciones de ruptura: *break* y *continue* para terminar anticipadamente el bucle.

- **break**. Interrumpe la ejecución de un bucle en cualquier momento, abandonando directamente el mismo aunque no se hayan finalizado todas las iteraciones. Se suele codificar dentro de una sentencia *if*, de forma que si se cumple una determinada condición se produzca la salida automática del bucle, independientemente de que la condición siga o no cumpliéndose.

- **continue**. Interrumpe la ejecución de la iteración actual pasando a ejecutar de nuevo el bloque de sentencias. En el caso concreto de la instrucción *for*, se evaluará la instrucción de incremento antes de entrar de nuevo en el bloque de sentencias.

Funciones

FUNCIONES DEL LENGUAJE

JavaScript dispone de una serie de funciones predefinidas que pueden utilizarse en cualquier parte del código. Para utilizar estas funciones, únicamente hay que invocarlas pasándole los parámetros adecuados.

La tabla de la figura 73 muestra el formato de las funciones predefinidas más importantes del lenguaje y la acción que realizan.

Nombre	Acción
eval (cadena)	Ejecuta la expresión o sentencia contenida en la cadena que recibe como parámetro.
escape (caracter)	Devuelve el código ASCII correspondiente al carácter.
unescape (cadena)	Devuelve el carácter cuya codificación coincide con el código indicado en cadena.
parseInt (cadena [,base])	Convierte la cadena de caracteres numéricos a un entero en una base numérica determinada. Por defecto en base 10.
parseFloat (cadena)	Convierte la cadena de caracteres numéricos a un real.
isNaN (valor)	Devuelve true si el valor evaluado es no numérico.

Fig. 73. *Funciones internas de JavaScript*

CUADROS DE DIÁLOGO

Se trata de unas funciones especiales cuya misión es la de generar una serie de cuadros de diálogo para presentar y solicitar información al usuario.

La figura 74 muestra los tres tipos de cuadros de diálogo que se pueden generar y la función encargada de ello.

Fig. 74. *Cuadros de diálogo*

La funcionalidad y sintaxis de estas funciones es la siguiente

- **alert**. Genera un cuadro de diálogo con un mensaje y botón de "Aceptar" para cerrar la ventana. Su formato de utilización es:

  alert("mensaje");

- **confirm**. Genera un cuadro de diálogo con un mensaje de pregunta y dos botones que cierran la ventana, "Aceptar", cuya pulsación devuelve el valor *true* y "Cancelar", cuya pulsación devuelve *false*:

  var variable = confirm ("mensaje pregunta");

- **prompt**. Genera un cuadro de diálogo para solicitar un dato a usuario. La pulsación de "Aceptar" cierra la ventana y devuelve el dato introducido:

  var variable = prompt("mensaje de solicitud");

DEFINICIÓN DE FUNCIONES

Además de utilizar las funciones propias del lenguaje, JavaScript permite definir funciones propias. Esto tiene una doble finalidad: por un lado, dividir un programa en "trocitos" pequeños, lo que facilita su seguimiento y búsqueda de errores y, por otro, reutilizar código, es decir, las instrucciones se escriben una vez dentro de la función y se ejecutan las veces que sea necesario.

Formalmente hablando, una función es un grupo de instrucciones que realizan una tarea determinada. Dentro de una función de JavaScript se pueden incluir llamadas a otras funciones de la aplicación, pero únicamente a aquellas que estén definidas en la página actual o en algún archivo .js externo referenciado desde la misma.

En la figura 75 se muestra cómo definir una función en JavaScript y la manera en que ésta debe ser invocada.

Los elementos que definen una función son:

- El nombre de la función (nombrefunción).

- Una lista de argumentos (argumento1, argumento2…). El número de argumentos puede variar desde ninguno hasta un número indeterminado.

- Las sentencias o instrucciones que se ejecutan dentro de la función.

Cuerpo de la función

```
function nombre_funcion (argumento1, argumento2, ...)
{
        sentencias
        [return valor;]
}
```

Llamada a la función

```
nombre_funcion(parametro1, parametro2, ...);
variable=nombre_funcion(parametro1, parametro2, ...);
 <elemento-HTML evento=nombre_funcion(valor1,valor2,valorn); >
```

Fig. 75. *Definición de una función en JavaScript*

Además, las funciones pueden devolver un valor de retorno, utilizando la instrucción: *return valor*, siendo *valor* una variable o expresión JavaScript que genera como resultado el dato a devolver por la función.

En el siguiente ejemplo tenemos una función que recibe dos números como parámetros y devuelve el mayor de ellos.

```
<html>
<head>
<script language="javascript">
      function mayor(x,y)
      {
            if(x>y)
                return x;
            else
                return y;
      }
</script>
</head>
<body>
<script language="javascript">
      var a=10,b=8,result;
      //obtiene el mayor de los números llamando
```

```
            //a la función definida
            result=mayor(a,b);
            alert("El número mayor es:\n" + result);
    </script>
    </body>
    </html>
```

Como se puede observar, la función está declarada en la cabecera de la página, mientras que la llamada se hace desde el cuerpo. En este caso, la instrucción de llamada aparece directamente en el script, aunque lo normal es que se encuentre dentro de otra función o en un manejador de evento.

EVENTOS

Un evento es una acción que el usuario puede realizar sobre un objeto XHTML (Controles, Tablas, Títulos...), tales como hacer un clic en un objeto del documento, arrastrar el ratón por encima de una imagen, pulsar un botón de comando, seleccionar un elemento en una lista, etc.

La mayoría de los eventos se producen por acciones del usuario, aunque existen otros que son generados por el propio navegador.

Manejadores de evento

Cuando el usuario realiza alguna acción sobre un elemento de la página, se produce el evento correspondiente que es capturado sobre la etiqueta correspondiente, por ejemplo, un clic en el documento provoca el evento *onclick* sobre la etiqueta <body>.

Cada evento tiene un manejador de evento, que se encarga de responder automáticamente cuando ocurre un evento. Estos manejadores están representados por atributos de la etiqueta y su nombre coincide con el del evento.

Habitualmente, las acciones de respuesta a un evento se agrupan en una función que es llamada desde el manejador.

Por ejemplo, la siguiente página incluye una función, llamada *procesa()*, que se ejecutará como respuesta al evento *onclick* sobre el documento:

```
<html>
<head>
<title>gestión de eventos</title>
```

```
<script language=javascript>
function procesar()
{
      window.alert("Ha hecho click en el documento");
}
</script>

</head>
<body onclick="procesar();">

</body>
</html>
```

Tipos de evento

La mayoría de los navegadores modernos soportan un gran número de eventos sobre los distintos componentes de la interfaz. La figura 76 muestra una tabla con los más destacados.

Nombre	Suceso
OnClick	Hacer un clic sobre el objeto.
onMousueOver	Sitúa el puntero del ratón encima del objeto.
onMouseOut	El puntero del ratón sale del objeto.
onMouseDown	Presionar un botón del ratón cuando el puntero está sobre el objeto.
onMouseUp	Liberar un botón del ratón cuando el puntero está sobre el objeto.
onFocus	El objeto recibe el foco.
onBlur	El objeto pierde el foco.
onKeyPress	Presionar una tecla cuando el foco está sobre el objeto.
onChange	Se modifica el valor de un select, text o Textarea.
OnLoad	Se produce la carga del documento.
OnUnload	El documento se descarga de la memoria.

Fig. 76. *Tipos de eventos reconocidos por JavaScript*

OBJETOS

Como sucede con la mayoría de los lenguajes modernos, JavaScript es un lenguaje basado en objetos. Esto significa que dentro de un script podemos hacer uso de objetos para aumentar la potencia y flexibilidad de nuestras aplicaciones.

Los objetos representan la evolución natural de las funciones, de hecho, persiguen su mismo fin: **reutilización de código y modularidad**.

Desde el punto de vista de la programación, un objeto es una "caja negra" que encapsula una serie de operaciones y de datos. El acceso al objeto desde el exterior se realiza mediante propiedades y métodos.

- **Propiedades**. Representan características del objeto (por ejemplo, el título de una ventana). Se les puede asignar un valor y leer su contenido. El acceso a las propiedades se realiza utilizando la notación *nombre_objeto.propiedad*. El siguiente ejemplo utiliza la propiedad location del objeto document para mostrar una nueva página en el navegador:

  document.location="http://www.google.es";

- **Métodos**. Son acciones asociadas al objeto, es decir, son las tareas que pueden hacer los objetos (por ejemplo, el método *open()* para abrir una ventana). Estas acciones están implementadas como funciones dentro del objeto. El acceso a los métodos se realiza utilizando la notación *nombre_objeto.metodo(parámetros)*. El siguiente ejemplo utiliza el método *write()* del objeto document para escribir información en la página de respuesta:

  document.write("<i>Mensaje de prueba</i>");

Tipos de objetos

Dentro de un bloque de código JavaScript de cliente podemos hacer uso de cuatro tipos de objetos:

- **Objetos del lenguaje**. Se trata de objetos incorporados por el propio lenguaje JavaScript que proporcionan propiedades y métodos de uso general en aplicaciones.

- **Etiquetas HTML**. Cualquier etiqueta HTML puede ser considerada como un objeto dentro de un script. El acceso a sus propiedades y

métodos se realiza utilizando el identificador (atributo *id*) de la etiqueta.

- **Controles**. Se trata de los objetos HTML utilizados para la generación de interfaces gráficas. Se crean mediante las etiquetas <input>, <select> y <textarea> y, al igual que sucede con el resto de etiquetas HTML, se utiliza el atributo id para acceder a sus propiedades y métodos.

- **Objetos del navegador**. Son objetos proporcionados por el navegador a los que se puede acceder directamente utilizando su nombre genérico (window, document.).

Objetos del lenguaje

JavaScript dispone de cuatro objetos que incorporan una serie de propiedades y métodos para realizar determinadas tareas habituales en los lenguajes de programación. Estas clases son:

- String
- Array
- Math
- Date

Algunos de estos objetos deben crearse a partir del operador new, seguido del nombre de clase correspondiente, mientras que otros son creados implícitamente por JavaScript.

A continuación, veremos las características de cada uno de ellos.

OBJETO STRING

Un objeto String es una cadena de caracteres. Cuando se asigna un dato de este tipo a una variable, ésta almacena una referencia al objeto, por lo que puede utilizarse para acceder a las propiedades y métodos del mismo.

Las propiedades y métodos de esta clase permiten la manipulación de cadenas desde código. Seguidamente, veremos algunos de los miembros más interesantes.

Propiedades

La única propiedad que proporciona este objeto es *length*, la cual contiene el número de caracteres de la cadena:

```
var k="hola";
alert(k.length); //muestra el valor 4
```

Métodos

- **charAt(posicion)**. Devuelve el carácter que ocupa la posición indicada, teniendo en cuenta que la posición del primero es 0.

  El siguiente script mostraría la cadena solicitada al usuario con los caracteres invertidos:

  ```
  var cadena=prompt("Introduce cadena");
  var aux="";
  for(var i=cadena.length-1;i>=0;i--){
          aux+=cadena.chartAt(i);
  }
  alert(aux);
  ```

- **indexOf(cadena)**. Este método devuelve la posición que ocupa en la cadena el texto proporcionado como parámetro. En caso de que no se encuentre, la llamada a *indexOf()* **devolverá –1**, de hecho, este método suele utilizarse en numerosas ocasiones para comprobar la existencia de un carácter o cadena dentro de otra.

- **substring(pos1, pos2)**. Devuelve el trozo de cadena que se encuentra entre las posiciones indicadas:

  ```
  var cad="Esto es una cadena";
  alert(cad.substring(12,17)); //muestra: cadena
  ```

- **toLowerCase()**. Devuelve la cadena en minúsculas.

- **toUppperCase()**. Devuelve la cadena en mayúsculas.

OBJETO ARRAY

Un array en JavaScript es un objeto de la clase Array. Para crear un objeto de este tipo hay que recurrir al operador new:

var obj = new Array();

Los arrays JavaScript no tienen un tamaño predefinido, redimensionándose dinámicamente cuando se añaden nuevos elementos al mismo y se eliminan los existentes.

Un array puede contener cualquier dato válido de JavaScript, incluso se pueden mezclar datos de diferente tipo.

Para acceder a cada elemento individual usaremos un índice, teniendo en cuenta que el primer elemento es el índice 0:

```
obj[0]= "hola"; //se asigna la cadena
                //a la primera posición
```

Es posible inicializar un array en el momento de su creación:

```
var datos =new Array("Pepe", "Juan", "Ana");
```

La anterior instrucción crearía un array con tres elementos.

La clase Array proporciona una serie de potentes métodos que facilitan la manipulación de arrays desde código.

Propiedades

La clase Array cuenta con una única propiedad. Se trata de la propiedad *length* y contiene el número de elementos que en ese momento tiene almacenados el array.

Métodos

- **join(separador)**. Devuelve una cadena de caracteres formada por la concatenación de todos los elementos del array, separándolos por el carácter indicado en el argumento de llamada al método:

  ```
  alert(datos.join("-"));
  ```

  La anterior instrucción devuelve la cadena: “Pepe-Juan-Ana”.

- **reverse()**. Invierte el orden de los elementos del array.

- **sort()**. Ordena alfabéticamente el contenido del array.

- **push(elementos)**. Añade el elemento indicado al array, situándolo en la última posición del mismo. Es posible añadir más de un elemento en la misma llamada al método:

```
datos.push("Laura", "Raúl"); //añade dos nuevos
                             //nombres al array
```

- **unshift(elemento)**. Añade un nuevo elemento al array, situándolo en la primera posición del mismo. Al igual que sucede con *push()*, permite añadir más de un elemento en la misma llamada.

- **pop()**. Elimina el último elemento del array.

- **shift()**. Elimina el primer elemento del array.

OBJETO MATH

Math es una clase que proporciona una serie de métodos para la realización de operaciones matemáticas dentro de un bloque de código JavaScript.

A diferencia de los anteriores, no es necesario crear objetos de esta clase para poder hacer uso de sus métodos y propiedades, utilizándose para ello la sintaxis:

Math.propiedad

o

Math.metodo(..)

Propiedades

- **PI**. Contiene la constante numérica que representa al número pi.

- **E**. Contiene la constante numérica que representa al número e.

Métodos

- **abs(numero)**. Devuelve el valor absoluto de un número.

- **ceil(numero)**. Devuelve el entero superior más cercano al número.

- **floor(numero)**. Devuelve el entero inferior más cercano al número.

- **round(numero).** Redondea el número, devolviendo el entero más cercano.

- **max(num1, num2)**. Devuelve el mayor de los dos números indicados.

- **min(num1, num2)**. Devuelve el menor de los dos números indicados.

- **pow(base, exponente)**. Calcula la potencia del número indicado en base, elevado a exponente.

- **random()**. Devuelve un número aleatorio entre 0.0 y 1.0, pudiendo llegar a alcanzar el extremo inferior pero no el superior.

El siguiente listado de ejemplo corresponde a una página HTML que, al ser cargada en el navegador, muestra una posible combinación de números para el juego de la primitiva (figura 77). Los números son generados aleatoriamente por un bloque de código JavaScript que se ejecuta cada vez que se carga la página:

```
<html>
<head>
    <style>
  td{font-size:30pt}
    </style>
</head>
<body>
<script language="javascript">
   var numerosvalidos,numerogenerado;
   var estado=false;
   //array donde se almacenarán los números correctos
   numerosvalidos= new Array();
   while(numerosvalidos.length<6){
     //genera un número aleatorio entre 0 y 49
      numerogenerado= Math.floor(Math.random()*49+1);
      for (var t=0; t<numerosvalidos.length; t++){
         if (numerosvalidos[t]==numerogenerado){
           estado=true;
         }
      }
      //si el número generado no está repetido lo
```

```
            //guarda
            if(estado==false){
               numerosvalidos.push(numerogenerado);
            }
            estado=false;
         }
         document.write("<table border='1' align='center'
                                        cellpadding='10'>");
         document.write("<tr>");
         //genera una tabla HTML con todos los números válidos
         for(var i=0;i<numerosvalidos.length;i++){
            document.write("<td>"+numerosvalidos[i]+"</td>");
         }
         document.write("</tr>");
         document.write("</table>");
      </script>
      </body>
      </html>
```

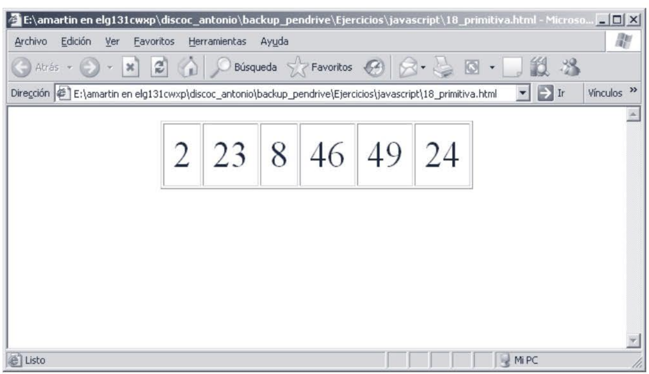


Fig. 77. *Página con la combinación aleatoria de la primitiva*

OBJETO DATE

Un objeto Date representa una fecha y hora concreta. Para poder crear un objeto de esta clase es necesario recurrir al operador new, pudiendo utilizar cualquiera de los siguientes formatos:

- new Date();
- new Date(año, mes, dia);
- new Date(año, mes, dia, hora, minuto, segundo);

En el primer caso se crea un objeto Date que almacena la fecha y hora actuales, mientras que en los otros dos casos se le asocian unos determinados datos.

Métodos

Seguidamente, exponemos algunos de los métodos más interesantes de este tipo de objeto.

- **getYear()**. Devuelve el año de la fecha que representa el objeto.
- **getMonth()**. Devuelve el mes como un número entre 0 y 11.
- **getDate()**. Devuelve el día del mes como un número entre 1 y 31.
- **getHours()**. Devuelve la hora como un número entre 0 y 23.
- **getMinutes()**. Devuelve los minutos como un número entre 0 y 59.
- **getSeconds()**. Devuelve los segundos como un número entre 0 y 59.
- **setYear(int año)**. Establece un nuevo valor para el año.
- **setMonth(int mes)**. Establece un nuevo valor para el mes.
- **setDate(int dia)**. Establece un nuevo valor para el día del mes.
- **setHour(int hora)**. Establece un nuevo valor para la hora.
- **setMinutes(int min)**. Establece un nuevo valor para los minutos.
- **setSeconds(int sec)**. Establece un nuevo valor para los segundos.

Objetos XHTML

Toda etiqueta HTML o XHTML existente en una página Web es un objeto que puede ser tratado desde JavaScript. Estos objetos exponen una serie de propiedades, relacionadas fundamentalmente con el estilo y formato de la etiquetas, que permiten modificar dinámicamente el aspecto de la página.

Esta modificación dinámica de la página cargada en el navegador es lo que se conoce como DHTML o HTML dinámico.

REFERENCIA A LOS OBJETOS ETIQUETA

El valor del atributo id de la etiqueta, utilizado dentro de uno de los métodos del DOM, permite obtener una referencia al objeto para así poder acceder a las propiedades del mismo:

var ref = document.getElementById(identificador);

PRINCIPALES PROPIEDADES

Aunque cada tipo de etiqueta puede tener algunas propiedades exclusivas propias de ese tipo, hay una serie de ellas que son comunes a la mayoría de las etiquetas. Vamos a analizar algunas de las más importantes.

- **style**. Es sin duda la propiedad más importante de cara a DHTML. Contiene una referencia al objeto Style, a través del cual podemos acceder a todas las propiedades de estilo CSS asociadas con la etiqueta. Por ejemplo, suponiendo que "parrafo" es el identificador asociado a una etiqueta tipo <p> existente en una página, el siguiente bloque de instrucciones permitiría modificar el tamaño del texto que se muestra en su interior:

  ```
  var obj = document.getElementById("parrafo");
  obj.style.fontSize = "20px";
  ```

- **className**. Como la anterior, esta propiedad permite modificar las propiedades de estilo de una determinada etiqueta. En este caso, className contiene el nombre de la clase de estilo asociada al objeto, y puede utilizarse para modificar simultáneamente un conjunto de propiedades de estilo.

  En la figura 78 se muestra una página de ejemplo consistente en tres párrafos que cambian de aspecto cada vez que pasamos el ratón por encima de ellos.

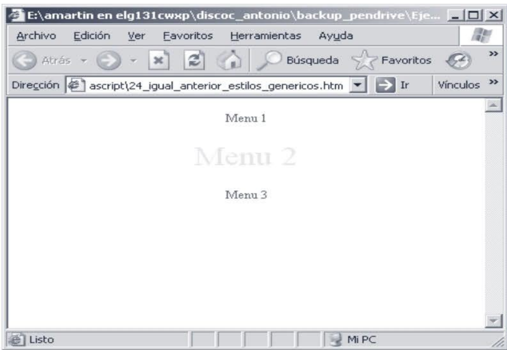


Fig. 78. *Modificación dinámica de estilos*

Para conseguir el efecto anterior, se debe implementar una función JavaScript que responda al evento *onmouseover* sobre la etiqueta y se encargue de modificar el estilo aplicado a la misma. El siguiente listado muestra el código completo de la página:

```
<html>
<head>
<style>
    .grande {font-size:30px;color:yellow;cursor:hand};
    .normal{font-size:12px;color:red;cursor:default};
</style>
<script>
    function aumenta(etiqueta){
 etiqueta.className="grande";
    }
    function estandar(etiqueta){
 etiqueta.className="normal";
    }
</script>
</head>
<body>
<center>
<p id="parrafo1" class="normal"
        onmouseover="aumenta(this);"
        onmouseout="estandar(this);">Menu 1</p>
```

```
<p id="parrafo2"  class="normal"
        onmouseover="aumenta(this);"
        onmouseout="estandar(this);">Menu 2</p>

<p id="parrafo2" class="normal"
        onmouseover="aumenta(this);"
        onmouseout="estandar(this);">Menu 3</p>
</center>
</body>
</html>
```

Obsérvese como se hace uso de la palabra *this* para hacer referencia al propio objeto etiqueta desde el manejador de evento.

- **innerHTML**. Permite modificar el contenido de una etiqueta HTML, admitiendo tanto texto plano como etiquetas HTML.
- **tagName**. Se trata de una propiedad de sólo lectura que contiene el nombre del tipo de etiqueta.
- **bgColor**. Representa el color de fondo de la etiqueta.
- **foreColor**. Representa el color del texto contenido en la etiqueta.

Objetos Controles XHTML

Los controles XHTML representan un tipo especial de etiqueta que se utiliza para solicitar datos al usuario a través de la página.

Estos objetos proporcionan propiedades y métodos que pueden utilizarse dentro de un script de cliente para realizar tareas diversas como la validación de datos o la personalización de la página o, incluso, como hemos visto en este libro, pueden ser utilizados como parte de una aplicación AJAX.

PROPIEDADES

Propiedades comunes

Entre las propiedades más importantes que podemos encontrar en todos los controles HTML, están:

- **value**. Representa el valor del control. En el caso de los controles de tipo caja de texto (text, textarea, password y hidden), esta propiedad

contiene el texto introducido en el mismo. En el caso de los controles checkbox y radio, la propiedad *value* contiene un valor asociado al control que no está visible en la página, mientras que en un control de tipo select esta propiedad está disponible para los subelementos de tipo option.

- **disabled**. Permiten habilitar (*false*) o deshabilitar (*true*) un control desde código.

- **checked**. Esta propiedad sólo está disponible para los controles de tipo checkbox y radio y contiene el estado del control: *true*, activado, y *false*, desactivado. Puede utilizarse también como escritura para establecer desde código un determinado estado en el control.

La figura 79 muestra una página con un formulario para la solicitud de datos al usuario.

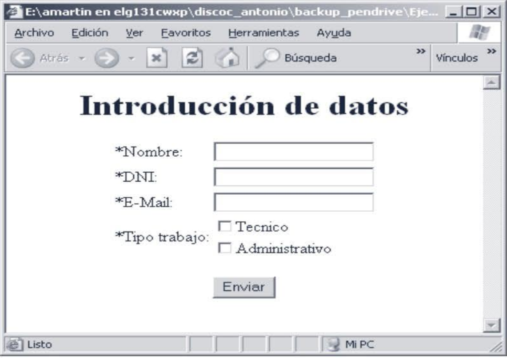


Fig. 79. *Formulario para solicitud de datos*

Todos los campos del formulario son de obligada cumplimentación, además, las cajas de texto DNI y E-Mail tienen que cumplir una condición adicional para que la información pueda ser enviada al servidor cuando se pulse el botón "enviar".

El siguiente listado muestra el código completo de la página:

```
<html>
<head>
<script language="javascript">

function valida(){
      if(cajas() && casillas() && doc() && correo())
            document.getElementById("datos").submit();
      else
            alert("¡Datos no válidos, compruébelos!");
}
function cajas(){
      //comprueba que se haya introducido algún valor
      //en las cajas de texto
      if(document.getElementById("nombre").value=="" ||
              document.getElementById("dni").value=="" ||
              document.getElementById("email").value=="")
            return false;
      else
            return true;
}
function casillas(){
      //comprueba que alguna casilla esté activada
      if(!document.getElementById("tec").checked &&
             !document.getElementById("adm").checked)
            return false;
      else
            return true;
}
function doc(){
      //comprueba que el DNI sera numérico y tenga
      //más de 8 caracteres
      if(isNaN(document.getElementById("dni").value) ||
             document.getElementById("dni").value.length>8)
            return false;
      else
            return true;
}
function correo(){
      //comprueba que la dirección de correo tenga
```

```
        //un formato válido
        var dir=document.getElementById("email").value;
        if(dir.indexOf("@")==-1 || dir.indexOf(".")==-1)
              return false;
        else
              return true;
  }

  </script>

  </head>
  <body>
        <form id="datos" action="" method="post">
              <center>
              <h1>Introducción de datos</h1>
              <table>
                    <tr>
                          <td>*Nombre:</td>
                          <td><input type="text"
                                id="nombre"></td>
                    </tr>
                    <tr>
                          <td>*DNI:</td>
                          <td><input type="text"
                                id="dni"></td>
                    </tr>
                    <tr>
                          <td>*E-Mail:</td>
                          <td><input type="text"
                                id="email"></td>
                    </tr>
                    <tr>
                          <td rowspan="2">*Tipo trabajo:</td>
                          <td><input type="checkbox"
                                id="tec">Tecnico</td>
                    </tr>
                    <tr>
                          <td><input type="checkbox"
                           id="adm">Administrativo</td>
                    </tr>
              </table>
```

```
                <br>
                <input type="button" value="Enviar"
                                     onclick="valida()">
                </center>
          </form>
    </body>
    </html>
```

Propiedades del control de tipo select

Dadas las características especiales de este control, <select> expone propiedades distintas a las de los anteriores:

- **selectedIndex**. Devuelve el índice del elemento seleccionado en la lista.
- **options**. Representa un array con los objetos de tipo option anidados en el control. Cada uno de estos objetos option expone, además, las siguientes propiedades:
- **value**. Valor asociado al elemento.
- **selected**. Indica si el elemento está (*true*) o no (*false*) seleccionado.
- **text**. Contiene el texto de la opción.

Por ejemplo, dado un control de tipo select cuyo identificador es “listado”, el siguiente bloque de instrucciones mostraría el texto de la opción seleccionada:

```
var obj = document.getElementById("listado");
var indice = obj.selectedIndex;
alert(obj.options[indice].text);
```

Objetos del navegador

Cuando se carga una página en un navegador se crean una serie de objetos característicos de éste, estos objetos disponen de una serie de propiedades y métodos que pueden ser utilizados en cualquier script del cliente sin tener que instanciar el objeto, simplemente se utilizará el nombre del objeto y el operador “.” para acceder a la propiedad o método.

Los objetos del navegador están organizados jerárquicamente (figura 80), lo que significa que para acceder a uno de estos objetos hay que indicar su referencia completa:

window.document.form

A continuación, veremos las propiedades y métodos más importantes de los principales objetos del navegador.

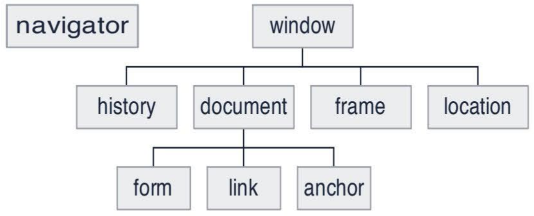


Fig. 80. *Jerarquía de objetos del navegador*

OBJETO WINDOW

Es el objeto principal de la jerarquía, representa la ventana del navegador y cada una de las áreas o marcos definidos.

Se trata del objeto que más propiedades y métodos tiene, con ellos se pueden mandar mensajes al usuario, solicitar datos, definir temporizadores, etc. Dado que es el objeto predeterminado del navegador, se puede acceder a sus propiedades y métodos escribiendo únicamente el nombre de éstos, sin tener que indicar la palabra window delante:

```
window.alert("hola");
```

equivale a:

```
alert("hola");
```

Propiedades

- **name**. Nombre del objeto window.
- **parent**. Referencia a la ventana padre del objeto. En caso de ser un frame, esta propiedad devuelve la ventana que contiene el elemento <Frameset>.

- **self**. Referencia a la ventana actual.

- **top**. Referencia a la ventana de nivel superior.

- **status**. Contiene el mensaje que aparece en la barra de estado del navegador. Puede ser modificado desde código.

Métodos

Algunos de los métodos de window, como *alert()*, *prompt()* y *confirm()*, han sido ya estudiados en el apartado dedicado a las funciones de JavaScript. Además de éstos, el objeto window proporciona los siguientes métodos:

- **setInterval(funcion, tiempo)**. Genera un temporizador. Un temporizador es un objeto que ejecuta una determinada tarea periódicamente cada cierto intervalo de tiempo.

  El primer parámetro del método es una cadena de caracteres que representa la llamada a la función que contiene el código de la tarea a ejecutar por el temporizador, mientras que "tiempo" es el intervalo de tiempo en milisegundos que trascurrirá entre una llamada a otra de la función.

  Por ejemplo, el siguiente código corresponde a una página XHTML con una imagen desplazándose de izquierda a derecha a través de la página. Este desplazamiento se consigue a través de un temporizador que cada décima de segundo modifica la propiedad de estilo *left* de la imagen:

```
<html>
<head>
<script language=javascript>
  var pos;
  function comenzar(){
      pos=0;
      bol.style.left=pos;
      //activa el temporizador
      window.setInterval("mover()",100);
  }
  function mover(){
      if(pos<400){
         pos+=10;
      }
```

```
        else{
          pos=0;
        }
        //actualiza la posición del objeto
        bol.style.left=pos;
    }
</script>
</head>
<body onload="comenzar();">
<h1>bola en movimiento</h1>
<img src="bolita.gif" id="bol"
                       style="position:relative">
</body>
</html>
```

La llamada al método *setInterval()* devuelve una referencia al objeto temporizador creado, dicha referencia puede ser almacenada en una variable de cara a poder utilizar el objeto en alguna otra parte del código.

- **clearInterval (objeto_temporizador)**. Este método elimina el temporizador cuya referencia se pasa como parámetro. Un vez eliminado el temporizador, dejará de ejecutarse la tarea periódica asociada al mismo.

- **setTimeout (funcion, tiempo)**. Genera un temporizador que, a diferencia del creado con *setInterval()*, realiza una única llamada a la función transcurrido el período de tiempo fijado en el segundo parámetro. Más que un temporizador, realiza una función de retardador.

  Por ejemplo, para conseguir el mismo efecto del programa anterior utilizando el método *setTimeout()*, deberíamos utilizar el siguiente script:

```
:
 <script language=javascript>
  var pos;
  function comenzar(){
        pos=0;
        bol.style.left=pos;
        window.setTimeout("mover()",100);
```

```
  }
  function mover(){
      if(pos<400){
         pos+=10;
      }
      else{
        pos=0;
      }
      bol.style.left=pos;
      //reactiva de nuevo el temporizador
      window.setTimeout("mover()",100);
  }
</script>
:
```

Como en el caso de *setInterval()*, la llamada a *setTimeout()* devuelve una referencia al objeto creado.

- **clearTimeout (objeto_temporizador)**. Elimina el temporizador que se le pasa como parámetro.

- **open(url, nombre, características)**. Otra de las funcionalidades que nos ofrece el objeto window es la apertura de nuevas ventanas del navegador desde código, tarea que es realizada con el método *open()*. Este método recibe tres parámetros:

  - **url**. URL del documento que se va a mostrar en la ventana. Si el contenido de la página a mostrar va a ser generado dinámicamente desde código, el valor del atributo se establecerá a cadena vacía (“”).

  - **nombre**. Nombre asociado a la ventana. Puede utilizarse como valor del atributo target de un enlace cuando se quiera cargar en la ventana el documento asociado al enlace.

  - **características**. Características de la ventana, como tamaño, posición, etc.

La llamada al método *open()* devuelve un objeto window que representa la nueva ventana creada.

La siguiente instrucción abriría una ventana vacía con las características indicadas:

```
var vent=window.open("","Ventana",
        "height=100,width=300,top=200,left=400");
```

- **close()**. Cierra el objeto window en el que se aplica el método.

OBJETO DOCUMENT

El objeto document representa el documento que se está visualizando en el objeto window. Equivale a la etiqueta <body> de XHTML, por lo que se puede acceder a las propiedades y métodos de este objeto utilizando document o el valor del atributo *id* de la etiqueta.

Propiedades

- **bgColor**. Representa el color de fondo del documento.

- **foreColor**. Color del texto incluido directamente en el documento.

- **location**. Contiene una cadena de caracteres con la URL de la página actual. Esta propiedad puede utilizarse para cambiar la página que en un determinado momento está siendo visualizada en la ventana del navegador. Por ejemplo, si queremos cargar en la ventana creada anteriormente con el método *open()* de window la página de inicio de google tendríamos que hacer lo siguiente:

```
vent.location="http://www.google.es";
```

Métodos

- **write(texto)**. Es un método muy utilizado en JavaScript y ha aparecido en alguno de los ejemplos presentados en este apéndice. Permite escribir un texto sobre el documento, pudiendo incluir etiquetas de marcado en el mismo.

  Por ejemplo, si quisiéramos generar en la ventana creada anteriormente una página como la que se muestra en la figura 81, deberíamos utilizar las siguientes instrucciones:

```
vent.document.write("<html><head><title>ventana abierta
                                    </title></head>");
vent.document.write("<body>");
vent.document.write("<h1>texto generado</h1>");
```

```
vent.document.write("<h2>desde la ventana principal
                                  </h2><br><br>");
vent.document.write("<center><input type='button'
             value='cerrar' onclick='self.close();'>");
vent.document.write("</center></body></html>");
```

Fig. 81. *Página generada dinámicamente*

OBJETO NAVIGATOR

Representa al propio navegador, y proporciona información sobre el mismo que puede ser utilizada desde código JavaScript a fin de personalizar determinadas instrucciones para cada tipo de navegador.

Propiedades

- **userAgent**. Contiene la cabecera con el mismo nombre que el navegador envía en las peticiones HTTP.

- **appName**. Contiene el nombre del navegador cliente.

- **appVersion**. Contiene la versión del navegador.

- **cookieEnabled**. Indica si el navegador admite (*true*) o no (*false*) el uso de cookies.

ÍNDICE ALFABÉTICO

A

a4j215
action, atributo 50, 247
ActiveXObject 29
addOptions, método 105
AJAX 13, 17, 37, 71, 153, 159, 170, 175, 183, 291
AJAX en JSF..203
AJAX en JSF 2.0..205
AJAX, acción JSF 205
AJAX, compatibilidad con navegadores.....30
Ajax.Autocompleter 133
Ajax.Request..114
Ajax.Updater..117
alert, método 277
API JSF 188
appendChild, método 69
appName, propiedad 301
appVersion, propiedad 301
Array, objeto JavaScript....................283, 284
ASP.NET...22
Attr, interfaz 61

B

Bean gestionado 191, 195
bgColor, propiedad 291, 300
break 272, 275
Business to Business..................................260

C

Cascading Style Sheet..............................251
checked, propiedad..................................292
childNodes, propiedad...............................64
className, propiedad 289
clearInterval, método...............................298
clearTimeout, método 299
close, método 300
COM 29
Componentes a4j.......................................215
Componentes gráficos JSF........................197
Componentes RichFaces...........................215
confirm, método 277
continue...275
Controlador 165, 181
Controlador JSF 189
Controles HTML
Button...249
CheckBox 249
Password...249
Radio 250
Select..250
Submit 249
Text...248
TextArea 248
Tipos 248
Convert..108
Convertidores..108
cookieEnabled, propiedad.........................301
Cookies...51
Create...102, 106
createElement, método...............................68

createTextNode, método 69
creator, método 102
CSS 20

D

Date, objeto JavaScript 288
Default 272
Direct Web Remoting 96, 97, 99, 101, 104, 106, 107, 108
do 275
DOCTYPE 265
Document, interfaz 61
Document, objeto 300
DOM 21, 32, 37, 59, 68, 72, 104
DSN..56
DWR ....*Véase Direct Web Remoting*
dwr.xml 101
DWRServlet 96, 98, 99
DWRUtil, objeto 104

E

Efectos jQuery 142
Element 119
Element, interfaz 61
Encabezados de petición 43
Encabezados de respuestas 50
encodeURIComponent, método 48
Engine.js 97
Enterprise JavaBeans 191
Etiquetas HTML
  Encabezados 240
  Formato del texto 240
  Frames 246
  Hipertexto 241
  Imágenes 245
  Listas 242
  Organización del texto 239
  Separadores 241
  Tablas 244
Eventos 279
Eventos jQuery 142
Eventos JSF 190
exclude 107

F

Faces-config.xml 189
FacesServlet 188, 189, 201
firstChild, propiedad 63, 66
for 273
foreColor, propiedad 291
Form 121
Formularios HTML 247
Funciones AJAX jQuery 143

G

get 42, 80
getAttribute, método 67
getElementById, método 63, 118
getElementsByTagName, método 63
getHeader, método 116
getResponseHeader, método 51
getValue, método 104

H

hasChildNodes, método 66
Hojas de estilo
  En línea 254
  Externa 252
  Interna 253
HTML 235
HTTP 21, 155

I

if 271
include 107
innerHTML, propiedad 28, 64, 291
insertBefore, método 69
Insertion 120
Insertion, propiedad 117
item, método 63

J

J2EE 14, 15, 22, 25, 70, 99, 101, 153, 155, 156, 157, 158
Java 96, 97, 99, 153
Java EE 187
Java Server Faces 153, 187, 192, 203
JavaBean 108, 193
JavaScript 20, 26, 267
  funciones 275
  instrucciones de control 271
  objetos controles XHTML 291
  objetos del lenguaje 282
  objetos del navegador 295

objetos XHTML 289
operadores 270
sisntaxis básica 268
tipos de datos..269
tipos de objetos..281
variables 270
JBoss 213
jQuery 137
jQuery, componentes..................................137
JSF, configuración del entorno..................192
JSF, registro de beans................................197
JSON 158
arrays 85
características 83
concepto 83
ejemplo..88
interpretación..86
objetos 84
utilidades 88
JSP..76, 188

L

lastChild, propiedad 66
length, propiedad..63
librerías JSF..190
location, propiedad.....................................300

M

Manejadores de evento........................37, 279
Manipulación DOM 139
Math, objeto JavaScript..............................285
method, atributo...248
minChars..134
Modelo 160, 176, 191
Modelo Vista Controlador........153, 156, 158, 175, 192

N

Navegación en JSF 202
Navigator, objeto...301
nextSibling, propiedad....................................66
Node, interfaz 61
NodeList, interfaz 61
nodeName, propiedad......................................65
nodeType, propiedad 64
nodeValue, propiedad..............................64, 65

O

onclick, evento 279
onkeyup, evento...56
onreadystatechange, propiedad 27, 44, 80
open, método.........................27, 41, 299, 300

P

param 102
paramName...133
parentNode, propiedad....................................66
parseJSON, método 88
PHP 22
POST..45, 48, 80
previousSibling, propiedad..............................66
prompt, método 277
Prototype...................................114, 118, 129

R

readyState, propiedad 28, 44
Reglas de navegación....................................189
removeOptions, método 105
responseText, propiedad........................32, 52
responseXML, propiedad 32, 52, 62
return 278
RichFaces..213

S

Script..267
Script.aculo.us.....................................129, 131
select, control 25
selected, propiedad..295
selectedIndex, propiedad................................295
Selectores...138
self, propiedad 297
send, método 45
setInterval, método 297
setRequestHeader, propiedad...........................43
setTimeout, método 298
setValue, método..104
status, propiedad 53
statusText, propiedad 53
String, objeto JavaScript 282
style, atributo..254
style, propiedad 289
switch 272

T

tagName, propiedad 291
text, interfaz..61
text, propiedad 295
toJSONString, método...............................89
typeof..31

U

undefined...................................30, 270, 272
USER-AGENT...43
userAgent, propiedad................................301
util.js...97

V

value, atributo...68
value, propiedad..............................291, 295
Vista 166, 183

W

W3C 20, 30, 41, 52, 59, 239, 260
While 274
window, objeto 296
write, método 300

X

XHTML 20, 24, 68, 235, 262
estructura de un documento 237
principales etiquetas............................239
tipos de etiquetas.................................236
XML 21, 33, 66, 68
aplicaciones...260
características......................................258
definición 255
estructura...264
reglas sintácticas 265
tecnologías base 263
XMLHttp 41
XMLHttpRequest...........21, 30, 37, 114, 170
acceso a respuesta 52
configuración 41
encabezados 43
encabezados de la respuesta..................51
envío de la petición 45
estado de la petición..............................44
estado de la respuesta............................53
función de retrollamada.........................43
origen 40
XMLHttpRequest.......................................27
XMLHttpRequest, encapsulación 75

AJAX en J2EE

2ª EDICIÓN ACTUALIZADA

La combinación de tecnologías Web existentes está provocando un importante cambio de filosofía en la construcción de aplicaciones para la Web, asemejándose cada vez más este tipo de desarrollos, tanto por su potencia como por su rapidez de respuesta al usuario, a la creación de aplicaciones de escritorio.

En este libro el lector encontrará todos los elementos necesarios para aprender a desarrollar auténticas aplicaciones Web interactivas con AJAX, incluyendo numerosos casos prácticos que le pueden resultar de gran ayuda.

Además de los elementos esenciales que lo componen, este libro también nos ilustra sobre la gran variedad de herramientas y utilidades de libre distribución existentes en la Red para trabajar con AJAX, analizando con gran detalle las más extendidas dentro de la comunidad de desarrolladores.

Aunque el libro está orientado a la utilización de Java/JAVA EE como tecnología de desarrollo de lado del servidor, cualquier persona con conocimientos de programación será capaz de comprender la mecánica de funcionamiento de AJAX y de adaptarla a su entorno particular.

Desde **www.ra-ma.es** podrá descargarse material adicional con los ejercicios del libro.