USERS

INCLUYE VERSIÓN DIGITAL GRATIS

Desarrollo web con Java desde cero

Programación web + Uso de Servlets + JavaServer Pages + Utilización de Struts y ActionForms + Programación de vistas + Validación de una web

RU

DESARROLLO WEB CON JAVA DESDE CERO

RedUSERS

TÍTULO: Desarrollo web con Java desde cero

COLECCIÓN: Desde Cero

FORMATO: 19 x 15 cm

PÁGINAS: 192

ISBN 978-987-1949-74-8

Martínez Quijano, Andrés

Desarrollo web con Java desde cero / Andrés Martínez Quijano ; coordinado por Gustavo Carballeiro.

1a ed. - Ciudad Autónoma de Buenos Aires : Fox Andina; Buenos Aires: Dalaga, 2014.

192 p. ; 19x15 cm. - (Desde cero; 39)

ISBN 978-987-1949-74-8

1. Informática. 2. Diseño de Aplicaciones. I. Carballeiro, Gustavo, coord. II. Título

CDD 005.3

Prólogo al contenido

A fines de 1994, la web dio un gran salto de la mano de Netscape 0.8: con este browser se podía acceder a fascinantes sitios con fondo gris y algunas imágenes.

Recién a mediados del año siguiente aparecía la aplicación web interactiva TeleGarden, que estaba armada con una cámara de video y un brazo robot que podía ser controlado desde la Web para hacer tareas básicas de jardinería. Nada más impresionante que eso.

Internet ya no genera tanta sorpresa, si se tienen en cuenta las experiencias con CGI, las soluciones más elegantes de ASP, PHP o JSP.

Hoy en día, parece natural asumir –erróneamente– que cualquier persona que posea cierto conocimiento de algún lenguaje puede construir algo que funcione en la Web.

¿El resultado?

Aplicaciones que agonizan cuando ingresan más de dos usuarios, reportes que tardan en aparecer y mega-objetos serializados que se transmiten en cada request.

La comunidad de desarrolladores continúa en expansión. Las nuevas caras que hoy empiezan a descubrir Java necesitan de un buen punto de partida para entender el paradigma web.

Este libro se propone ejercer un rol de maestre al evitar especificaciones rígidas, estableciendo una línea muy clara con ejemplos sencillos y concretos.

Explota el potencial de Java sin dejar de lado las buenas prácticas, incluso para quienes ya vienen programando.

Que lo disfruten mucho.

El libro de un vistazo

A lo largo del presente libro veremos todas las herramientas necesarias para convertirnos en verdaderos expertos en el desarrollo de aplicaciones web con Java. En cada capítulo, veremos un ejemplo integrador, real y concreto, desarrollado con las herramientas open source Eclipse y Tomcat.

*01 LA PROGRAMACIÓN WEB

En este capítulo, que tiene un carácter introductorio, veremos qué es la programación web y algunas de sus características clave.

*02 SERVLETS

¿Cómo es, en la práctica, una aplicación web Java? Conoceremos qué es un servlet, configuraremos un servidor Tomcat para realizar pruebas y codificaremos algunos ejemplos.

*03 JAVASERVER PAGES

El trabajo puede tornarse un poco difícil con grandes aplicaciones. Estamos en condiciones de conocer una alternativa más sencilla, denominada JavaServer Pages (JSP).

*04 STRUTS

Si utilizamos solo servlets o JSP tendremos complicaciones en los desarrollos complejos. Por eso, en este capítulo haremos una introducción a Struts, una potente herramienta de desarrollo web.

*05 ACTIONFORMS

En este capítulo conoceremos los ActionForms: qué son, cómo programarlos y cuál es su importancia en el desarrollo de aplicaciones. Luego del enfoque tradicional, contemplaremos otras alternativas.

*06 ACTIONS EN STRUTS

En este capítulo veremos los Action en Struts: para qué sirven,

cómo programarlos, sus diversas alternativas y cómo interactúan con la vista usando tags propios de Struts.

el envío duplicado. Además, cómo enviar mensajes y errores desde la acción para validaciones complejas.

*07 VISTA

Es momento de programar la vista. Es por eso que, en este capítulo, veremos cómo generar páginas JSP con tags de Struts y sus utilidades.

*ApB ON WEB EJEMPLO INTEGRADOR DEL CAPÍTULO 7

Conoceremos cómo crear una página de ingreso de usuarios y cuáles son las combinaciones de datos a ingresar.

*ApA ON WEB ENVÍOS DUPLICADOS

En esta sección online, complementaria al capítulo 6, aprenderemos a prevenir un problema recurrente:

*ApC ON WEB STRUTS VALIDATOR

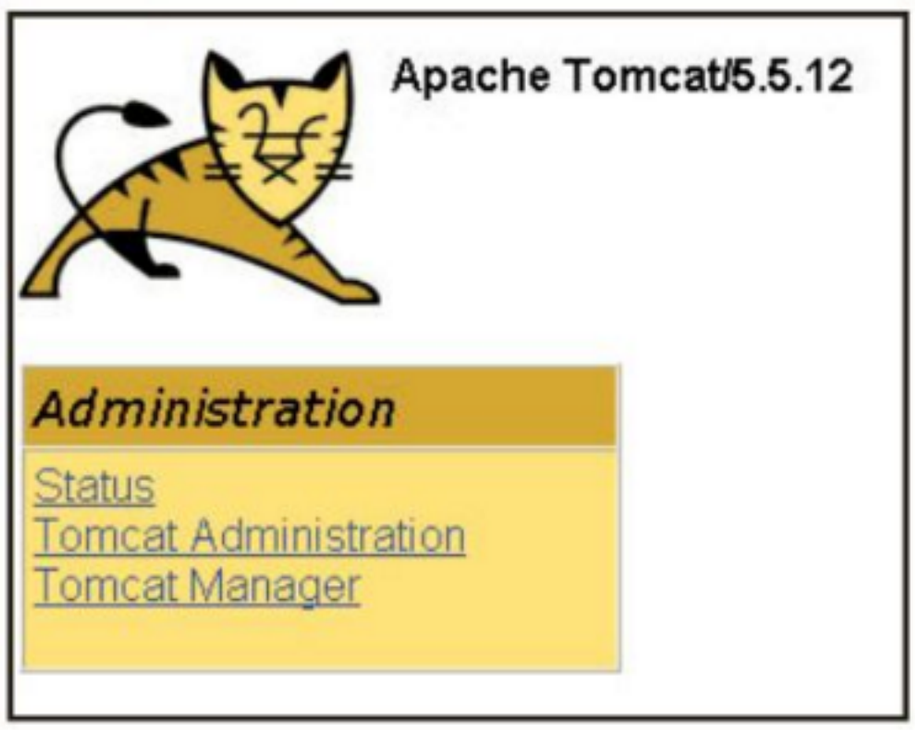
Veremos un componente que valida automáticamente nuestros formularios. Conoceremos sus validaciones básicas y crearemos nuevas.

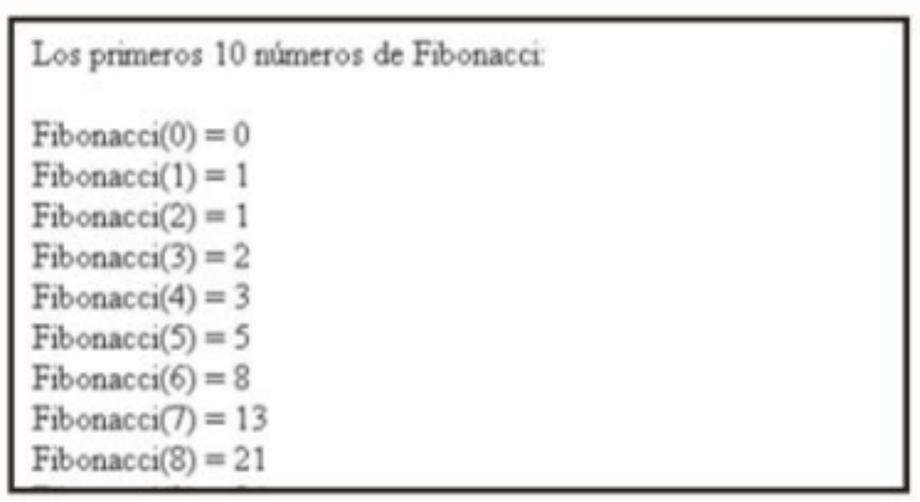
INFORMACIÓN COMPLEMENTARIA

A lo largo de este manual, podrá encontrar una serie de recuadros que le brindarán información complementaria: curiosidades, trucos, ideas y consejos sobre los temas tratados. Para que pueda distinguirlos en forma más sencilla, cada recuadro está identificado con diferentes iconos:

CURIOSIDADES E IDEAS | ATENCIÓN | DATOS ÚTILES Y NOVEDADES | SITIOS WEB

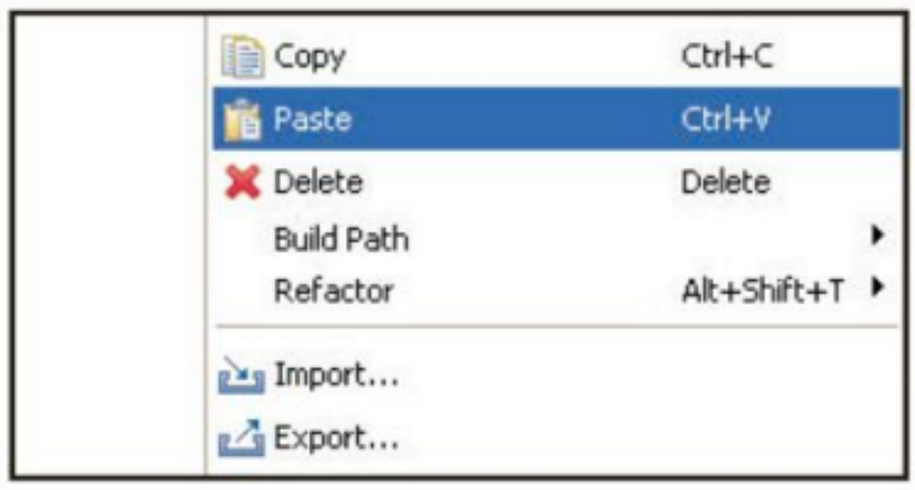
Contenido del libro





*04

Struts

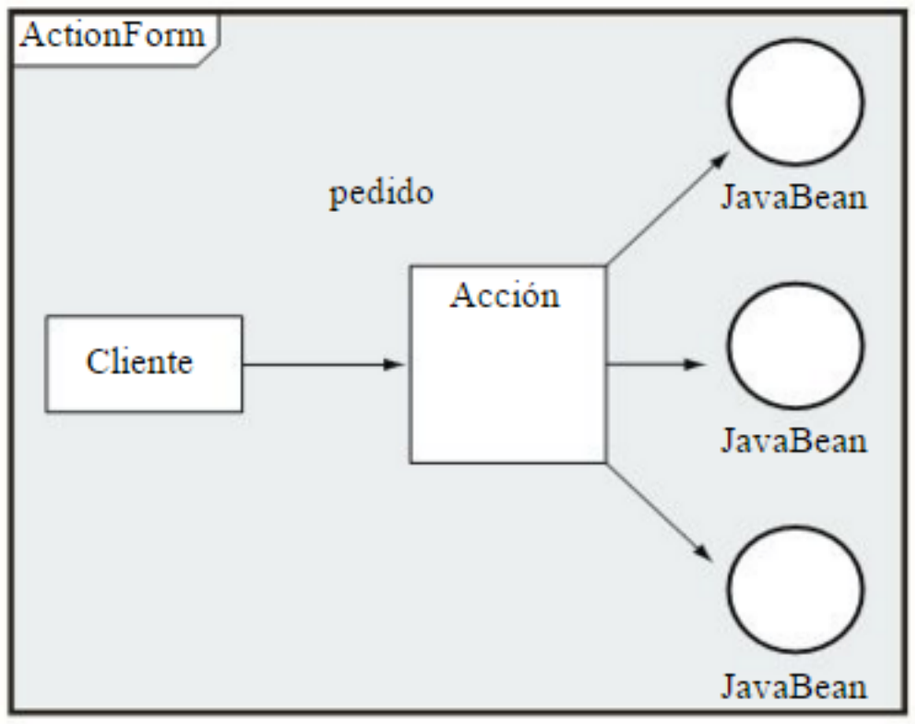


*05

ActionForms

*06

Actions en Struts



*07 Vista

*Ap A ON WEB Envíos duplicados

*Ap B ON WEB Ejemplo integrador del capítulo 7

*Ap C ON WEB Struts Validator

Introducción

El primer contacto con la programación web puede ser a veces difícil de implementar, si no es el ámbito habitual de nuestros desarrollos. Con el tiempo, practicando e investigando, haciendo algunos proyectos de sitios web dinámicos y, sobre todo, equivocándonos en el proceso –lo que conduce al aprendizaje–, podemos comenzar a entender de qué trata el desarrollo web. Con el recuerdo de cuáles son las complicaciones que nos podemos encontrar y las dudas que se nos pueden presentar en los primeros pasos del desarrollo, este libro está planteado en un orden cuasi-cronológico que sigue, de alguna forma, la evolución que sufrió el desarrollo web con Java. Empezaremos con simples servlets –esa promesa salvadora que intentó pero no pudo ser JSP– y con la inclusión del paradigma MVC en la Web.

Pensado para el programador que no tiene experiencia en desarrollo de aplicaciones web, este libro lo introducirá en un mundo donde es necesario un cambio en la forma de pensar las aplicaciones. Un mundo en donde la interacción con el usuario se asemeja bastante a dar indicaciones a un albañil acerca de cómo queremos una pared, mediante cartas que él se dignará a responder cuando quiera (si es que quiere). El libro intenta recapitular las dificultades que existen en cada paso, a fin de evitar que el lector tenga que tropezar varias veces con la misma piedra.

Al final de cada capítulo se presentarán interrogantes y problemas a resolver relacionados con la temática vista. Habrá preguntas que no son de fácil respuesta: debemos advertir que es muy probable no encontrarla al releer el capítulo. Tendremos que pensar, buscar información, hacer programas y probar para hallar la respuesta. Esa es la idea del libro.

*01

La programación web

La programación de aplicaciones web impone una nueva forma de pensar y programar aplicaciones. En este capítulo veremos qué es la programación web y algunas de sus características clave.

Introducción al mundo web

En los últimos años, internet dejó de ser un mero divertimento para pasar a ser un medio fundamental de desarrollo de negocios. Hoy en día, mediante internet podemos hacer mucho más que visitar páginas web y chatear; estamos acostumbrados a realizar muchas de nuestras tareas cotidianas: pagar las cuentas, alquilar una película en el videoclub, revisar nuestros e-mails, reservar un hotel y pasajes para nuestras próximas vacaciones, y muchas cosas más. Todo esto, independientemente del lugar donde estemos y del horario. En este contexto, estar en internet es condición indispensable para cualquier empresa.

En un principio, parecería suficiente con tener una mera página con información de contacto en la que se muestren productos y servicios, pero enseguida se vuelve imprescindible proveer nuevos servicios a los potenciales clientes a través de la red de redes, para no perder competitividad en el mercado. Un banco ya no puede dejar de ofrecer home banking , una línea aérea que no tenga página en internet donde se puedan consultar los vuelos actualizados y reservar o comprar pasajes pierde gran parte de su mercado. Y así con cualquier área de negocio en que pensemos. Programar este tipo de servicios puede parecer muy complicado y, si bien hay algunas aplicaciones críticas, como, por ejemplo, las bancarias –en donde la transferencia de efectivo no puede dar lugar a errores de sistemas y se requiere mucha inversión

en materia de seguridad–, veremos que normalmente desarrollar una aplicación web segura no es más complicado que programar cualquier aplicación stand-alone . De hecho, dados los reducidos requerimientos y limitaciones de este tipo de desarrollos y la gran cantidad de software open source que hay disponible para ser usado gratuitamente, en muchos casos suele ser más simple hacer una aplicación web que una de escritorio.

Internet

Para poder meternos de lleno en la programación web, es fundamental entender cómo funciona, a grandes rasgos, internet.

Cuando abrimos un navegador y lo apuntamos hacia una página web, por ejemplo, www.google.com , en el fondo se está generando una comunicación entre dos programas: un cliente y un servidor .

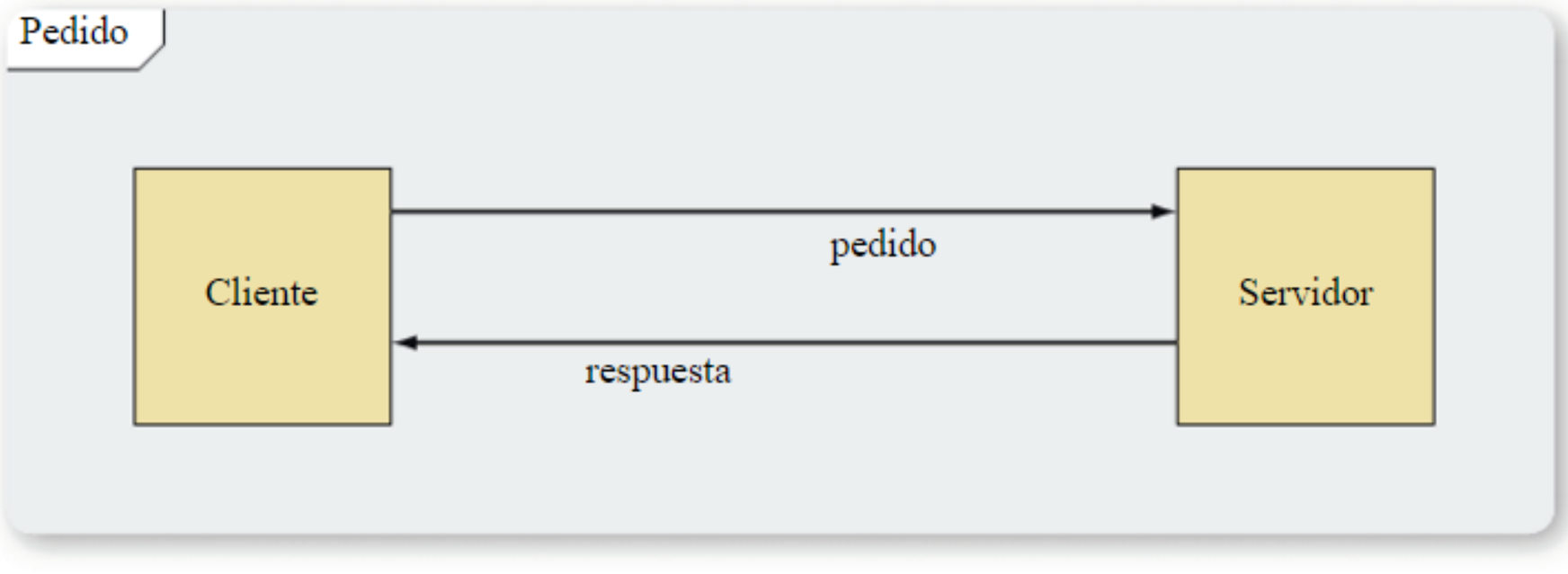


Figura 1. Al navegar por internet, básicamente estamos realizando una comunicación entre dos programas.

El cliente es nuestro navegador (Internet Explorer, Firefox, etcétera), que se comunica con el servidor. En este caso, el servidor es un programa que está constantemente escuchando peticiones de clientes y que devuelve, para cada pedido, una respuesta acorde.

La respuesta es tomada por nuestro navegador y se muestra en pantalla, y de esta forma navegamos, de pedido en pedido, respuesta tras respuesta, continuamente.

Sobre HTML

En este libro asumiremos que las páginas que devuelve el servidor son siempre páginas HTML. Si bien esto ocurre en la mayoría de los casos en los sitios de internet, también hay muchos sitios que trabajan con otras tecnologías, entre las cuales la más común es Flash .

HTML significa Hypertext Markup Language . Si lo traducimos al castellano: Lenguaje de Marcado de Hipertexto . Nos alcanza con saber que HTML es un lenguaje que le especifica ciertos atributos a su contenido, y los navegadores web saben muy bien cómo mostrar este contenido formateado según esos atributos.

Por ejemplo, un documento HTML (o página HTML) puede especificar que en cierto lugar del documento haya que insertar una imagen, o que cierto texto dentro de su contenido vaya en negrita o con determinada tipografía. O sea, no difiere mucho de lo que podemos hacer con un documento Word; de hecho, cualquier procesador de texto permite exportar documentos como archivos de tipo HTML.

La diferencia entre ambos documentos radica en el contexto en que van a utilizarse, y en que HTML es un documento escrito enteramente en texto plano. Esto significa que podemos crear documentos HTML en cualquier editor de texto sencillo.

Ahora que sabemos qué es HTML, vamos a desglosar su significado:

COOKIES

Existe bastante controversia sobre las cookies. Hay gente que las defiende y gente que dice que son una invasión a la privacidad, dado que el servidor está guardando información en nuestra PC y la puede usar para recabar información sin nuestro consentimiento. Por eso, la mayoría de los navegadores permite definir si queremos aceptar cookies o no y configurar varias opciones más sobre ellas.

- Hypertext : se dice que un documento HTML contiene hipertexto en el sentido de que este documento puede referenciar a otro documento HTML. Cada vez que un documento HTML referencia a otro, se dice que está conectado a ese otro documento mediante un enlace (o link).
- Markup : un documento HTML define sus secciones mediante marcas (etiquetas o tags) en su contenido.
- Language : ¡HTML es un lenguaje! Los navegadores saben hablar este lenguaje: cuando un servidor les provee una página HTML, saben cómo mostrarla en pantalla.

Páginas estáticas

En muchos casos, el servidor es un programa que simplemente toma un pedido y devuelve una página (compuesta por uno o más archivos HTML, imágenes, etcétera) que está guardada en algún lugar del disco. En estos casos, decimos que son páginas estáticas . A no ser que el administrador del sitio actualice su contenido, al ingresar en la página web siempre obtendremos el mismo resultado, no importa cuándo ingresemos ni desde dónde. Para la gran mayoría de sistemas de negocios, este tipo de servicio no es muy útil, ya que se suele necesitar que se devuelva una página con contenido dinámico , que cambie según quién pidió la página, según la fecha, etcétera.

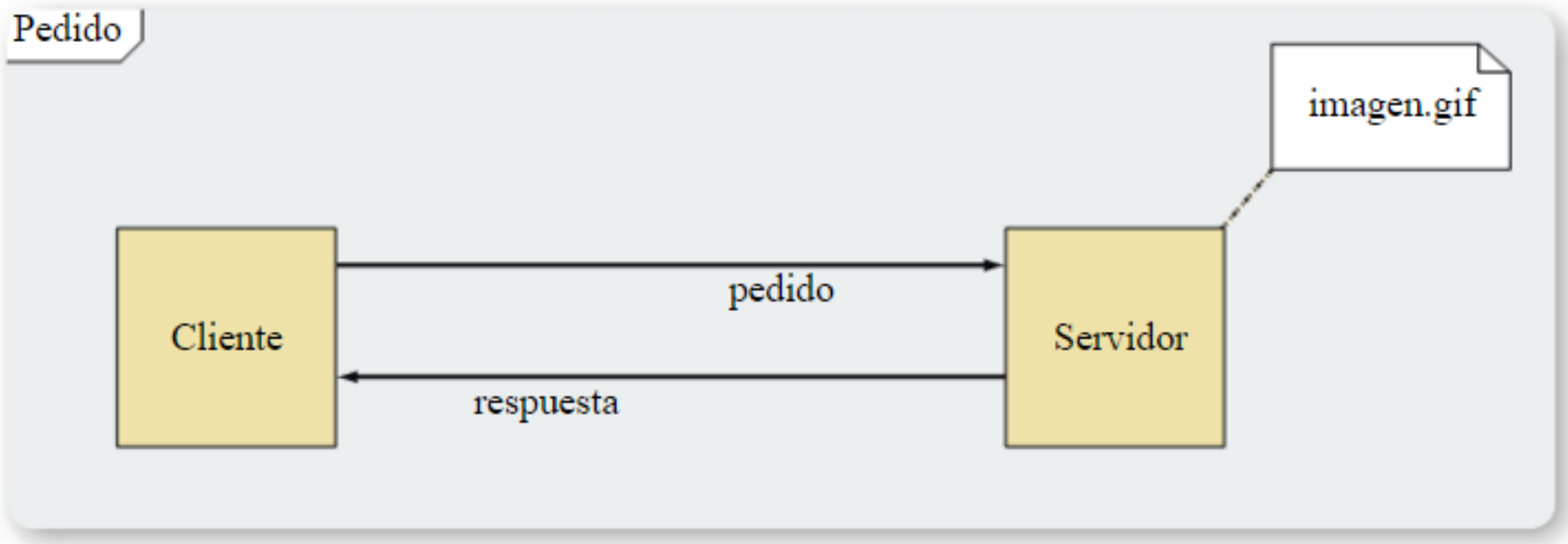


Figura 2. El servidor de páginas estáticas siempre devuelve el mismo recurso para el mismo pedido.

Sitios web con páginas dinámicas

Pensemos, por ejemplo, en el sistema de un banco. Si ingreso en el sitio del banco y quiero consultar mi saldo, debería devolverme mi saldo actual al momento exacto de pedirlo. Si el sitio web sólo provee contenido estático, entonces deberían tener una página guardada en el disco para cada saldo posible. Esto, obviamente, es impracticable. Y tampoco es viable que haya una persona que esté actualizando las páginas de saldos de todos los usuarios a medida que van realizando operaciones sobre sus cuentas. En esos casos, el servidor efectúa operaciones (ir a buscar datos a una base de datos, consultar con otro servidor o acceder a otro tipo de servicios de negocios) y devuelve una página dinámica ; esto es, una página que no está guardada en ningún lugar dentro del servidor, sino que fue creada en el momento para quien la pidió. El servidor accede a los datos variables (en este caso, el saldo del usuario), luego construye, en su memoria, la página con este dato y se la devuelve al cliente, que la muestra en pantalla.

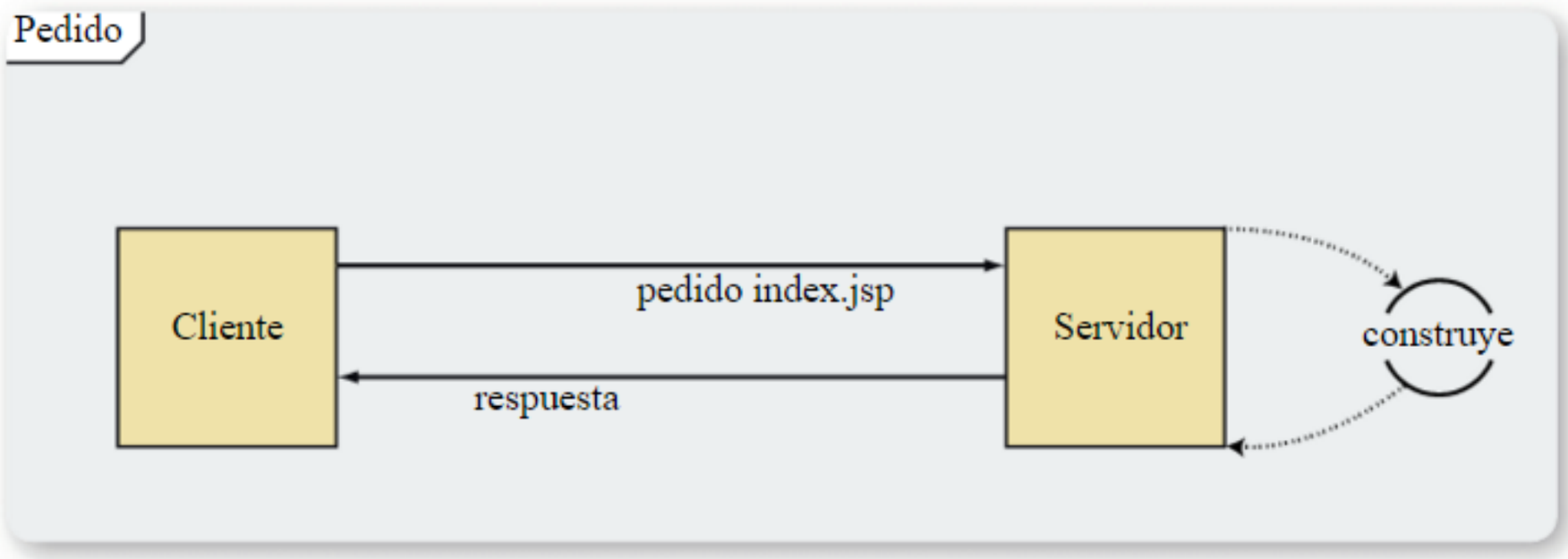


Figura 3. El servidor web dinámico construye una respuesta distinta para cada pedido.

La programación web, entonces, consiste en escribir programas que, dada una petición web realizada por un cliente (un navegador), procesen el pedido y generen y devuelvan un resultado.

Como en todo proceso de desarrollo de software, hay mucho de reutilización. Podremos ver que, por ejemplo, la parte de recibir el pedido y

enviar la respuesta por lo general no cambia, solamente cambia la lógica de negocios de cada pedido y el contenido de la respuesta generada. En este libro utilizaremos herramientas ya desarrolladas y de código abierto –open source – que nos ayudarán en la creación de aplicaciones web.

Programación web vs. tradicional

La programación web es considerada un nuevo paradigma, dado que impone ciertas restricciones que pueden resultar confusas en un principio. Estas restricciones se basan en que internet –al menos hasta ahora– trabaja sobre el protocolo HTTP (HyperText Transfer Protocol).

Cada vez que escribimos en un navegador http://direccion.web.com, estamos indicándole explícitamente que se conecte usando dicho protocolo. Si no lo escribimos, por lo general el navegador se encarga de rellenarlo automáticamente, aunque los navegadores a menudo también implementan otros tipos de comunicaciones entre ellos y un servidor. Por ejemplo, Internet Explorer y Firefox soportan comunicaciones de tipo FTP (File Transfer Protocol), que es un protocolo para la transferencia de archivos. Para acceder a un determinado recurso mediante ese otro protocolo, basta con escribir la dirección en el navegador (por ejemplo, ftp://ftp.uba.ar) y este sabrá, automáticamente, que debe establecer una conexión con el servidor mediante el protocolo FTP.

CGI

El primer sistema de páginas dinámicas fue desarrollado en 1993 y se llamó CGI (Common Gateway Interface). Estaba muy ligado al servidor de páginas web y se encontraba escrito en lenguaje C. Si bien se trataba de un sistema muy complejo, este tipo de programación fue –e incluso todavía es – usado por muchos programadores por su rapidez y seguridad.

HTTP

El protocolo HTTP, como su nombre lo indica, fue diseñado para transferir documentos de hipertexto (documentos HTML). En sus orígenes, cuando internet era ARPANET y era muchísimo más pequeña de lo que es ahora, y cuando la velocidad de conexión era enormemente inferior, texto plano era lo único que se transfería por ella. A nadie se le ocurría siquiera publicar un documento con imágenes o sonidos. A medida que avanzó el tiempo y la red se volvió más veloz, estas necesidades se hicieron evidentes y el protocolo fue mejorado para poder transferir cualquier tipo de datos, incluyendo voz, imágenes, video, etcétera.

HTTP tiene varias características, pero la que más nos va a interesar es que es un protocolo que no guarda el estado (stateless). Esto significa que no se mantiene constantemente una conexión entre el cliente y el servidor, sino que el cliente manda el pedido y corta la conexión, sin guardar información sobre pedidos anteriores. De esta forma, el servidor trata cada pedido en forma independiente del anterior, simplemente porque no puede saber si el pedido proviene del mismo cliente, aunque hayan ocurrido muy cerca en el tiempo. Esto puede sonar muy confuso. Una analogía que aclara bastante las cosas es la siguiente: el servidor es una persona sentada detrás de una puerta. Esta persona recibe papelitos con preguntas por debajo de la puerta, los mira, escribe en ellos una respuesta y los envía de vuelta, y esa es toda la comunicación que tiene con el mundo exterior. Esta persona no

HTTP

La actual versión del protocolo HTTP es la 1.1. En www.w3.org/Protocols/rfc2616/rfc2616.html se encuentra la especificación completa. El World Wide Web Consortium considera que la versión 1.1 supera todas las deficiencias de la versión 1.0 y, por lo tanto, no está trabajando en futuras versiones. Sí están trabajando en un protocolo relacionado que busca integrar HTTP con otro llamado XML Protocol .

sabe quién mandó cada papelito, solo los recibe y responde. Quizá del otro lado de la puerta hay una sola persona que es la misma siempre y manda diferentes preguntas, o hay cientos de personas, cada una con sus inquietudes que esperan ser respondidas; pero este humilde servidor no lo sabe, ni puede saberlo. Esto hace complicado mantener un hilo de conversación entre los clientes (quienes mandan los papelitos) y el servidor (quien los recibe y responde). Por ejemplo, si una persona hace una pregunta y, al obtener la respuesta, quiere hacer otra pregunta relacionada, tiene que formular la pregunta sabiendo que quien responde la tomará como si fuera una pregunta totalmente nueva.

> EL PROTOCOLO HTTP FUE DISEÑADO PARA TRANSFERIR DOCUMENTOS DE HIPERTEXTO

Esta problemática también hace imposible saber si alguien que acaba de mandar un papelito sigue estando tras la puerta o se fue hace rato. Un sitio web que se comunique con el cliente únicamente mediante documentos HTML a través del protocolo HTTP no tiene forma de saber (sin acudir a otras técnicas de programación, como applets, Flash, componentes ActiveX, etcétera) si el usuario sigue navegando en su sitio o si se fue a otra página o cerró el navegador.

Sesiones

Esta restricción del protocolo HTTP es un impedimento muy grande. Prácticamente toda aplicación necesita superar este inconveniente y poder mantener un registro de la conversación mantenida entre el usuario y el servidor. Para ello, se define el concepto de **sesión**, que se maneja de la siguiente forma:

- El cliente realiza un pedido al servidor.
- El servidor responde el pedido y, a su vez, le devuelve un identificador al cliente.

- El cliente deberá, en los sucesivos pedidos, incluir este identificador en cada pedido que realice al servidor.
- El servidor, al reconocer el identificador, puede mantener un estado de pedidos de un mismo cliente.

De esta forma, se soluciona el problema que acarrea la naturaleza sin estado del protocolo HTTP.

Hay dos formas de lograr que el navegador incluya este identificador de sesión en cada pedido al servidor. La primera es el uso de cookies. Las cookies (¡galletitas!) son pequeños archivos con información que el servidor envía al navegador para que este guarde, y que el navegador vuelve a enviar en cada pedido que realiza al servidor. De este modo, el navegador puede guardar información específica sobre el sitio que se está visitando, como el identificador de sesión o las preferencias del usuario (por ejemplo, el idioma o el color con que prefiere visualizar la página).

La segunda forma, llamada URL rewriting, hace que el navegador sobrescriba todos los enlaces que vuelven al servidor, agregándoles como parámetro el identificador de sesión. Esto es: cada acción que el usuario pueda hacer desde el navegador que vuelva al servidor (y que no sea un enlace externo a otro sitio), va a tener agregado un parámetro con el identificador. De esta forma, al hacer clic sobre cualquiera de los links que tenga la página, estará pasando a su vez el identificador de sesión, y así el servidor podrá identificarlo.

> LAS COOKIES SON ARCHIVOS CON INFORMACIÓN QUE EL NAVEGADOR GUARDA Y VUELVE A ENVIAR

Es importante destacar que todo tipo de información de estado se guarda en el servidor. Es este quien mantiene los datos asociados con la sesión del usuario. Por ejemplo: en el clásico sistema de compra online, donde hay un carrito de compras virtual, cada vez que un usuario agrega un producto, se agrega a la lista que ya contiene el carrito del usuario, pero este carrito y su información

reside en el servidor, asociado con el identificador de sesión del cliente. El cliente simplemente manda pedidos y se le muestran resultados, sin saber lo que ocurre del otro lado.

Pedidos HTTP

Hemos visto que los clientes realizan pedidos mediante el protocolo HTTP a servidores web. Lo que no vimos hasta ahora es que los pedidos HTTP pueden ser de varios tipos (también son llamados "métodos").

La Tabla 1 muestra los diferentes tipos de pedidos que existen actualmente para el protocolo HTTP, versión 1.1.

TIPOS DE PEDIDO PARA EL PROTOCOLO HTTP 1.1	
▼ TIPO DE PEDIDO	▼ DESCRIPCIÓN
OPTIONS	Se usa para preguntarle al servidor las diferentes formas de comunicación que soporta.
GET	Pide un recurso al servidor.
HEAD	Igual que GET, pero el servidor sólo devuelve el encabezado de lo pedido.
POST	Método que se usa para enviar información al servidor.
PUT	Usado para enviar recursos al servidor.
DELETE	Borra recursos del servidor.
TRACE	Se usa para pedir un rastreo del pedido.
CONNECT	Método reservado.

Tabla 1. Los diferentes métodos que define el protocolo HTTP.

Estos ocho métodos definidos en la especificación del protocolo permiten establecer conexiones muy avanzadas entre clientes y servidores, pero que no son usadas, por lo general, por las aplicaciones web estándar. El método TRACE, por ejemplo, se usa para testear que el servidor esté recibiendo los datos correctamente y para depurar las conexiones. La mayoría de los servidores en producción directamente tienen deshabilitados muchos de estos métodos o están asociados con algún tipo de directiva de seguridad que impide que cualquier usuario los ejecute. Caso contrario, cualquier usuario desde algún lugar remoto del planeta podría ejecutar un DELETE en una página y la borraría del servidor. En este libro vamos a referirnos únicamente a pedidos de los tipos POST y GET. Si bien a priori POST y GET parecen totalmente distintos entre sí, vamos a ver que pueden usarse con los mismos propósitos, aunque hay casos en los que se evidencia que uno es más idóneo que el otro. La especificación recomienda distintos usos para cada uno de ellos: dice que el método GET debería usarse solamente para obtener datos del servidor, y POST, para enviar información al servidor, como ser una orden de compra o una actualización de un dato. Así y todo, podemos usar GET para enviar información al servidor (con ciertas limitaciones) y POST para obtenerla.

Algo sobre seguridad

La seguridad en aplicaciones web excede ampliamente los alcances y contenidos de este libro. Es un tema muy tratado y se ha escrito mucho acerca de ello. Pero vamos a dar un pequeño pantallazo del tema, su problemática y algunas formas de mantenerlo bajo control. Los sistemas web implementan numerosos procesos en los que deben ofrecer seguridad.

Más adelante veremos una aplicación simple pero poderosa que se utiliza para brindar seguridad en lo referido a autenticaciones y autorizaciones.

- Autenticación es el proceso que se encarga de verificar que un usuario es realmente quien dice que es. Es común en muchos sitios web; cada vez que nos piden que ingresemos nuestro nombre de usuario

y contraseña, básicamente están pidiendo que demostremos que somos el usuario dueño de la contraseña, y por eso la ingresamos.

- Autorización consiste en verificar que un usuario dado (que, suponemos, ya se ha autenticado) tenga permisos para efectuar determinada operación. Por ejemplo, en determinado sistema, solo el usuario administrador puede borrar información; de manera que, cuando el servidor identifique que se está pidiendo borrar algo, primero deberá verificar que el usuario que está pidiendo el borrado sea un usuario de tipo administrador, y, en ese caso, autorizará la acción.

Sin embargo, el principal problema de seguridad que plantean las aplicaciones web (al igual que toda aplicación distribuida) es el transporte de datos. El protocolo HTTP transporta información entre computadoras a través de muchos nodos hasta llegar al servidor al cual se dirige. Esto trae muchos problemas de seguridad, ya que alguien puede tener acceso a la comunicación que se establece entre cliente y servidor y leer los datos que se transmiten, y estos datos pueden incluir números de tarjetas de crédito, claves, etc.

La solución que se usa hoy en día (aunque, en materia de seguridad informática, nada es ciento por ciento seguro) es un protocolo llamado HTTPS . Este protocolo especifica que las comunicaciones se siguen haciendo igual que con HTTP, con la diferencia de que, antes de enviar la información el cliente, la encripta . El servidor la recibe, la desencripta y luego la procesa. Así, si alguien logra interceptar la comunicación, es casi imposible que pueda descifrar su contenido.

RESUMEN

Hemos visto los servicios de valor que brindan los sitios web dinámicos. Describimos también sus limitaciones y cómo subsanarlas analizando la programación web independientemente del lenguaje de programación utilizado para implementar nuestras aplicaciones.

Actividades

TEST DE AUTOEVALUACIÓN

1. ¿Cuáles son los dos actores fundamentales en la navegación por internet?
2. ¿En qué se diferencia un servidor de páginas estáticas de uno de páginas dinámicas?
3. ¿Qué característica del protocolo HTTP es la que más limita la programación web?
4. ¿Qué solución se aplica a este problema?
5. ¿De qué formas el navegador puede enviar al servidor el identificador de sesión?
6. Usando telnet y simulando ser un navegador web, ingrese a su diario online favorito y obtenga la página principal.
7. ¿Qué es una cookie?
8. ¿Por qué la mayoría de los servidores web tienen deshabilitado el método HTTPDELETE ?
9. ¿Cuál es la diferencia entre autenticación y autorización?
10. ¿Por qué es necesario encriptar la información confidencial cuando es enviada a través de internet?

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: profesor@redusers.com .

*02

Servlets

En este capítulo veremos cómo es en la práctica una aplicación web Java. Aprenderemos qué es un servlet, configuraremos un servidor Tomcat para realizar pruebas y, finalmente, nos introduciremos en la codificación de algunos ejemplos.

¿Qué son los servlets?

Según vimos en el primer capítulo, para crear una aplicación web basta con desarrollar un servidor que escuche pedidos y devuelva recursos.

Pero es claro que el desarrollo web no consiste únicamente en eso. Sun creó (dentro del contexto del Java Community Process , que permite a cualquier persona del mundo participar en la toma de decisiones) la Java Servlet Technology , una especificación que define una serie de clases, paquetes y procedimientos que se utilizan para la creación de servlets , la base fundamental del desarrollo web con Java.

Algo a destacar acerca de los servlets es que la Java Servlet Technology es una especificación . Es decir, que la tecnología consta solamente de un documento, muy detallado, donde indica la funcionalidad que se provee y cómo deberá ser el software que la implemente, pero no proporciona ningún tipo de software en concreto. Si bien Sun desarrolló su propia implementación, la especificación nos permite trabajar con cualquier implementación que queramos, en tanto y en cuanto se atenga a la especificación. Podemos decir que esta última define un estándar que ha de cumplirse por los implementadores.

El hecho de trabajar en base a una especificación tiene muchas ventajas, pero también un problema: la especificación solo define cómo debe comportarse el software, pero no limita sus capacidades. Por lo tanto, algunos implementadores le agregan muchas funcionalidades

SERVLETS Y VERSIONES DE ESPECIFICACIÓN

Es muy probable que en un futuro salgan nuevas versiones, con más funcionalidades.

En este libro utilizaremos una de sus múltiples versiones. Recomendamos visitar la página www.oracle.com/technetwork/java/javaee/servlet/index.htm . para estar al tanto de los cambios de versión.

a la implementación que desarrollan, de forma tal de cumplir con la especificación, pero, además, soportar sus propios requerimientos. Esto conlleva el riesgo de que, si usamos funcionalidades extra de una implementación específica, después será más difícil migrar a cualquier otra implementación.

Un servlet es, entonces, un componente de software Java que se encarga de generar contenido dinámico en respuesta a pedidos HTTP. Todo servlet reside dentro de un servlet container, que es una parte del servidor web. El servidor web recibe los pedidos, revisa su sintaxis y los deriva al servlet container, que, a su vez, determina qué servlet debe invocarse, construye los objetos correspondientes que recibirá el servlet, le envía el pedido y el servlet devuelve la respuesta generada. El servlet container es un componente dentro del servidor.

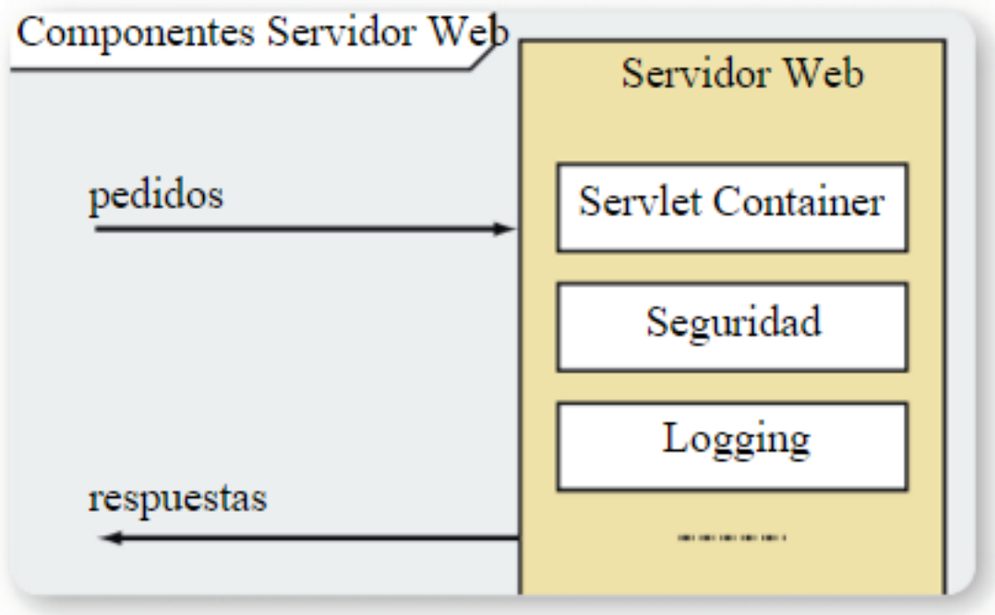


Figura 1. El servlet container es un componente más dentro del servidor.

SERVLET EN ENTORNOS DE DESARROLLO

Veremos que algunos de los métodos de los servlets no son utilizados en la programación web, sino que sirven para brindar información al desarrollador, principalmente cuando se trabaja con un entorno de desarrollo muy potente. De esta forma, en caso de tener muchos servlets, el entorno de desarrollo nos mostrará información extra para su mejor manipulación.

La especificación define una interfaz servlet que puntualiza los métodos que se describen a continuación:

```
void init(ServletConfig config);
/* Cuando el contenedor crea el servlet (ya sea por un pedido o
por inicialización del servidor), llama a este método para que
el servlet pueda inicializar los recursos necesarios durante su vida */

void destroy();
/* Método llamado cuando el contenedor decide destruir
el servlet. El servlet deberá liberar los recursos que tomó
y ejecutar acciones sabiendo que será destruido        */

ServletConfig getServletConfig();
/* Devuelve el objeto de configuración del servlet */

void service(ServletRequest req, ServletResponse res);
/* Método de ejecución del servlet. Dado un pedido
(encapsulado por un ServletRequest) y un objeto de respuesta
(el objeto ServletResponse), el servlet ejecuta su lógica en este método */

String getServletInfo();
/* Devuelve una cadena con información sobre este servlet */
```

Luego nos dedicaremos a los detalles, pero, básicamente, podemos ver que un servlet se inicializa, se le efectúan pedidos y finalmente se destruye.

La especificación es lo suficientemente amplia como para soportar servlets que respondan a cualquier tipo de pedido. Nosotros vimos que, para internet, las comunicaciones se realizan mediante el protocolo HTTP. Por lo tanto, es obvio que existen clases específicas para servlets que responden a pedidos HTTP, ya que son los más comunes hoy en día.

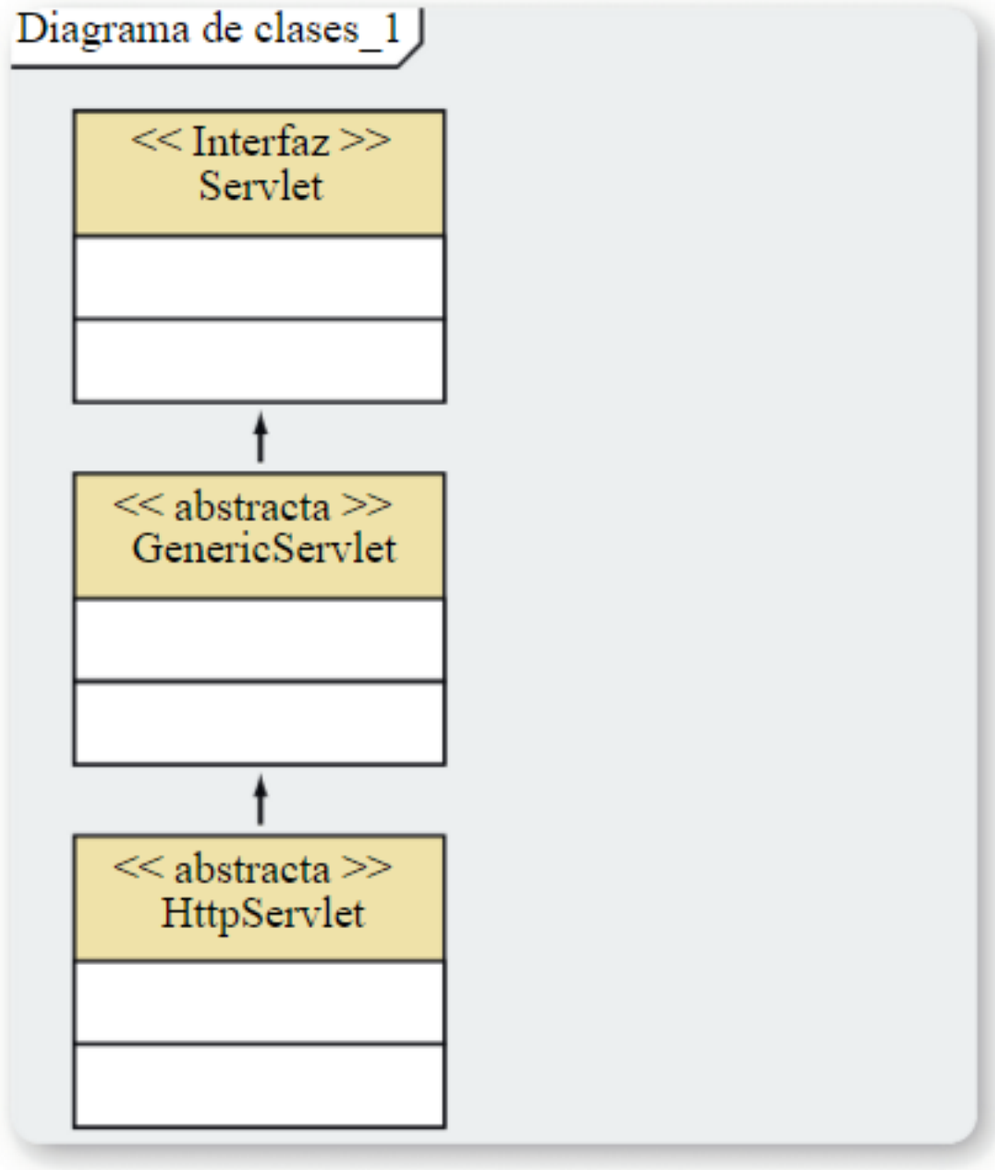


Figura 2. Diagrama de clases de servlets.

La clase abstracta HttpServlet define los siguientes métodos:

```
protected void doDelete(HttpServletRequest, HttpServletResponse);
protected void doGet(HttpServletRequest, HttpServletResponse);
protected void doHead(HttpServletRequest, HttpServletResponse);
protected void doOptions(HttpServletRequest, HttpServletResponse);
protected void doPost(HttpServletRequest, HttpServletResponse);
protected void doPut(HttpServletRequest, HttpServletResponse);
protected void doTrace(HttpServletRequest, HttpServletResponse);
protected void service(HttpServletRequest, HttpServletResponse);
void service(ServletRequest req, ServletResponse res);

protected long getLastModified(HttpServletRequest);
```

Veamos cómo funciona un servlet; en particular, un HttpServlet :

- Un pedido HTTP llega al servidor.
- El servidor interpreta el pedido y lo reenvía al servlet container.
- El servlet container parsea el pedido, crea los objetos correspondientes y crea o llama a la instancia del servlet que ha de manejarlo.
- El servlet asignado para manejar ese pedido es, asumamos, un HttpServlet .
- El servlet container invoca el método Service con los objetos HttpServletRequest y HttpServletResponse que creó como parámetros.
- El servlet inspecciona los objetos recibidos como parámetros e invoca el método correspondiente, según el tipo de pedido recibido.
- El método se ejecuta y la respuesta es generada.
- El servlet container envía la respuesta al cliente que originó el pedido.

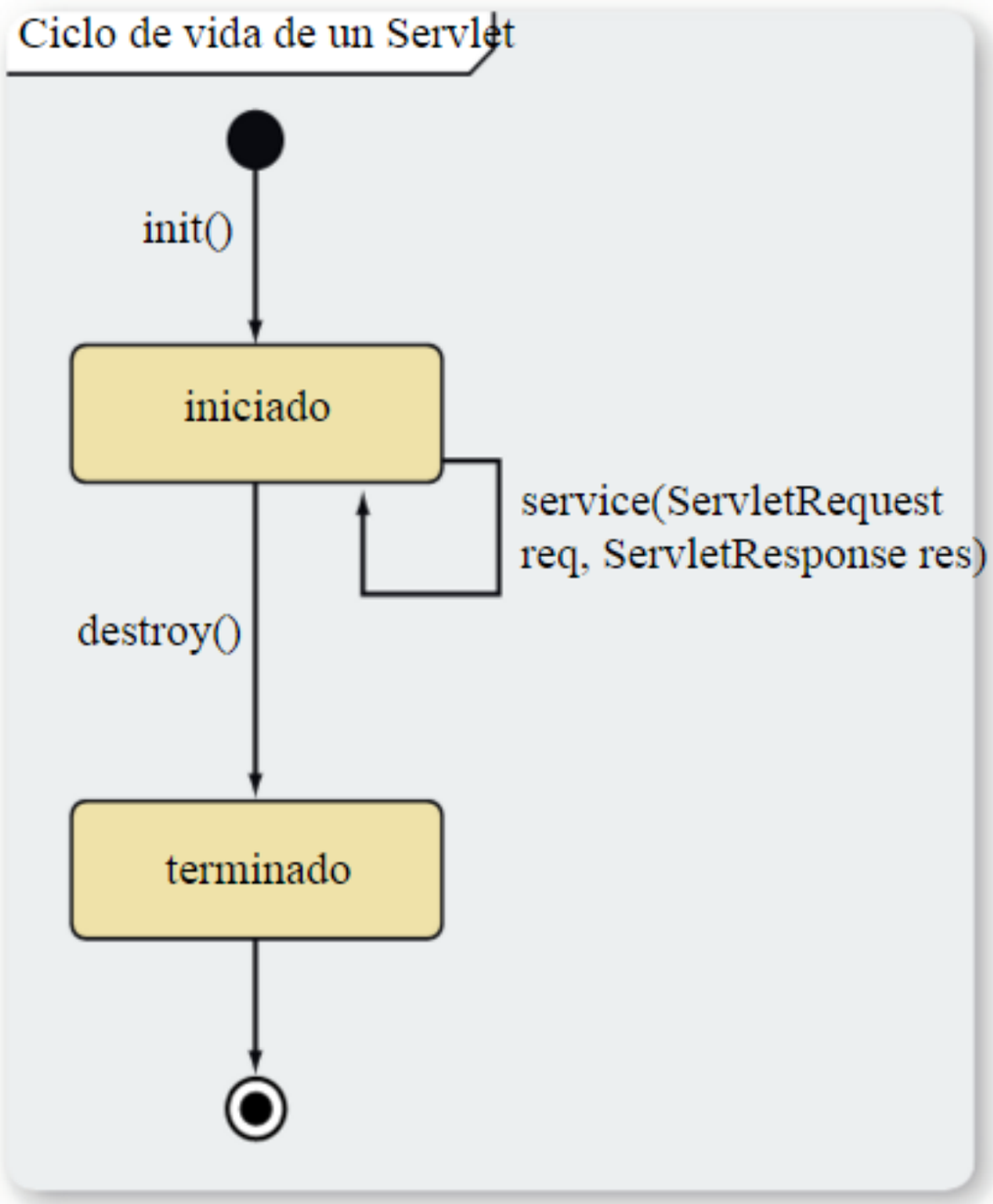


Figura 3. Ciclo de vida de un servlet.

El tipo de pedido es uno de los que vimos en el Capítulo 1 . Como vemos, hay métodos definidos para manejar cualquier tipo de pedido.

Un ejemplo: Hola mundo

Veamos un ejemplo de servlet, el clásico "Hola mundo!":

```
package capitulo2;

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;

/*
 * Extendemos la clase HttpServlet para definir
 * nuestra propia funcionalidad cuando el
 * servlet sea invocado
 */
public class HolaMundoServlet extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {

/* Llamamos al método doPost. Este servlet responde
de la misma manera tanto a pedidos GET como POST */
        doPost(req, res);
    }

    protected void doPost(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
```

```
/* Obtenemos el objeto que envía texto
   al cliente */
        PrintWriter out = res.getWriter();

// La respuesta
        out.println("Hola mundo!");

// Cerramos el stream
        out.flush();
        out.close();
    }
}
```

El servlet es invocado por el contenedor de servlets del servidor, y, si el pedido es de tipo POST o GET, el servlet responde con una página HTML (aunque no tenga ningún tipo de marca HTML, no deja de ser un documento HTML) que dice "Hola mundo!". Este servlet no es muy útil que digamos, ya que entrega siempre la misma cadena de texto.

Aunque es ejecutado como un servicio de contenido dinámico, está devolviendo contenido estático.

CONTENT-TYPE

Los servlets pueden especificar qué tipo de contenido están devolviendo (el content-type), para que el navegador lo pueda manejar. De esta forma, si un servlet entrega un contenido que no sea HTML, el navegador puede utilizar algún plugin o abrir otra aplicación, dependiendo del tipo de contenido que esté recibiendo (que podría ser un documento Word, un PDF o cualquier otro tipo que sea necesario).

Antes de pasar a otro ejemplo, veamos en qué consisten las clases HttpServletRequest y HttpServletResponse .

HttpServletRequest

Esta clase encapsula la información que contiene el pedido del cliente. La clase se extiende a partir de ServletRequest , una clase que contiene opciones de pedidos genéricos, y la extiende con información específica del protocolo HTTP. Un servlet accede a esta clase para saber qué contenido devolver. Aunque es teóricamente posible implementar toda la funcionalidad de un sitio web en un solo servlet basándose en parámetros del pedido, sería a costa de un código inmanejable, confuso y difícil de mantener. Para evitarlo, esta clase provee métodos para acceder a parámetros, encabezados, cookies y opciones de seguridad. Veamos algunos de los métodos (tanto de la clase HttpServletRequest como de su clase padre, ServletRequest , que usaremos más seguido:

```
Object getAttribute(String name)
Enumeration getAttributeNames()
void removeAttribute(String name)
void setAttribute(String name, Object o)
Locale getLocale()
Enumeration getLocales()
Map getParameterMap()
Enumeration getParameterNames()
String[] getParameterValues(String name)
Cookie[] getCookies()
HttpSession getSession()
HttpSession getSession(boolean create)
String getRequestedSessionId()
```

Un pedido puede tener asociados objetos, llamados atributos. Estos objetos no son asociados al pedido por el navegador web, sino por componentes en el servidor. Más adelante veremos cuáles son los usos de estos atributos. Los métodos getAttribute, getAttributeNames, removeAttribute y setAttribute manipulan dichos atributos asociados.

Los navegadores más recientes permiten al usuario definir el idioma en que prefiere navegar en la Web. Esto se traduce en un encabezado llamado Accept-Language que define el o los idiomas preferidos. Si programamos nuestras aplicaciones para que tomen en cuenta esta información, podremos mostrar al usuario páginas en su idioma, sin interacción por parte de él. Los métodos getLocale y getLocales devuelven el o los idiomas preferidos por el usuario, según el parámetro enviado por el navegador.

Los métodos getParameterMap, getParameterNames y getParameterValues nos permiten interactuar con los parámetros enviados por el navegador. Estos parámetros, a diferencia de los atributos, sí son asignados por el navegador cuando el usuario interactúa con la página, pulsando un enlace o enviando un formulario web. El método getCookies nos permite obtener las cookies que este cliente tiene alojadas.

Y por último, los métodos getSession y getRequestedSessionId nos sirven para tener acceso a la sesión a la cual pertenece este pedido. Notemos que getRequestedSessionId es el indicador de sesión del que hablábamos en el Capítulo 1, y getSession nos devuelve un objeto HttpSession que contiene toda la información de sesión en el servidor.

LENGUAJES Y NAVEGADORES

Al igual que con otras funcionalidades, no podemos depender de que el navegador que utilice el usuario nos envíe el encabezado Accept-Language, ya que puede suceder que ese navegador sí soporte esta funcionalidad pero que el usuario no la haya definido o no sepa cómo tiene que hacerlo.

HttpServletResponse

Esta interfaz define datos y métodos para manipular la información que será devuelta al cliente. Un servlet recibe de parámetro una instancia de este tipo de objeto y es su tarea modificar el objeto según su funcionalidad para que luego la respuesta le sea enviada al cliente. Veamos algunos de los métodos que usaremos de esta interfaz y de su superinterfaz, ServletResponse .

```
ServletOutputStream getOutputStream()
PrintWriter getWriter()
void addCookie(Cookie cookie)
void addHeader(String name, String value)
void setHeader(String name, String value)
boolean containsHeader(String name)
void sendRedirect(String location)
```

Los métodos getOutputStream() y getWriter() nos devuelven objetos en los que podemos escribir el contenido de la respuesta. En el primer caso, un OutputStream , útil para devolver contenido binario, y en el segundo caso, un PrintWriter , apto para devolver texto al cliente.

COOKIES QUE VENCEN

Las cookies tienen una fecha de expiración. Esta característica puede resultar muy útil si queremos guardar un dato sólo por determinado período de tiempo, por ejemplo, para recordar a un usuario durante una semana. El navegador es el encargado de eliminar las cookies que están vencidas.

Con addCookie(Cookie c) podemos enviar una cookie al cliente, que luego será reenviada en cada pedido y la podremos inspeccionar. Es importante notar que el usuario podría asignar la opción de no recibir cookies en su navegador, de manera que debemos prever esta situación y no escribir aplicaciones que dependan del uso de ellas. Los métodos addHeader(String name, String value) y setHeader(String name, String value) escriben contenido en el encabezado de la respuesta. La diferencia entre estos métodos consiste en que setHeader escribe el valor en el encabezado, pisando (si hubiera) un valor existente. En cambio, addHeader nos permite asignar múltiples valores a una misma entrada. Para evitar pisar valores, el método containsHeader(String name) nos permite saber si ya existe un valor asignado como encabezado para un nombre dado.
Por último, sendRedirect(String location) se utiliza para indicar al navegador que debe redirigirse a otra página.

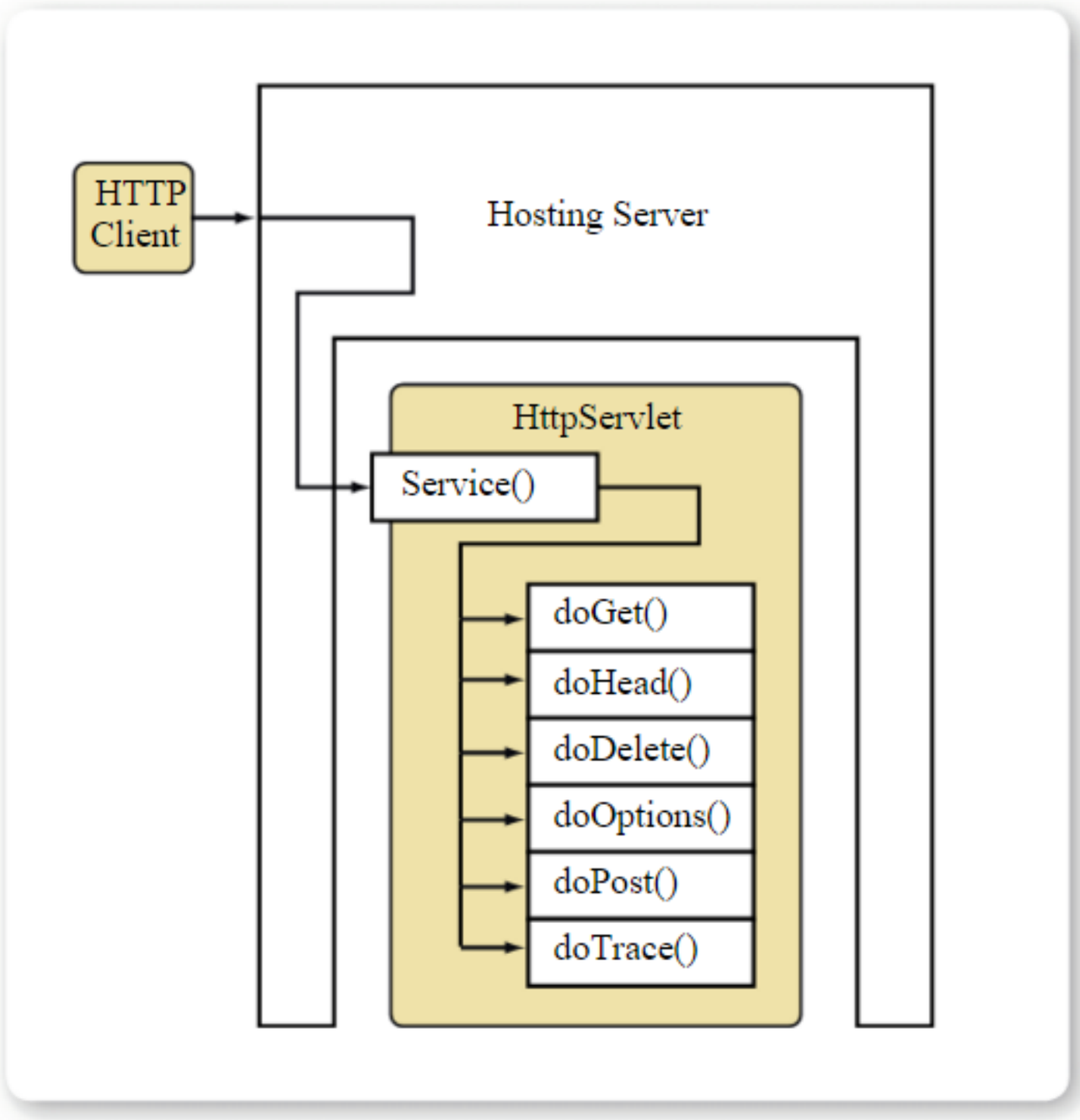


Figura 4. Gráfico que esquematiza la estructura de HttpServlet.

Utilizar servlets

Hemos visto algunos ejemplos de servlets, pero ahora llegó el momento de verlos en acción. Vamos a configurar un servidor web y a realizar un pedido a los servlets para verlos ejecutándose.

Para los ejemplos de este libro, vamos a utilizar las siguientes tecnologías:

- Eclipse : entorno de desarrollo. Si bien no es necesario, todo tipo de desarrollo se simplifica enormemente gracias a este framework. Eclipse poco a poco se va convirtiendo en la herramienta de desarrollo gráfico de facto. Basada en el viejo Visual Age de IBM, ahora de código abierto, es un software que emplea plugins en los que se puede agregar funcionalidades y soporte para muchísimas tareas con poco esfuerzo. De hecho, mediante el uso de plugins se puede utilizar Eclipse para desarrollo no solo de Java, sino también de PHP y otros lenguajes.

- Tomcat : servidor web y servlet container. Tomcat es un proyecto open source de muchísima calidad desarrollado por Apache, que actualmente es utilizado en entornos de producción por numerosas empresas. Usaremos la versión 8 que, al momento de redacción del libro, es la última versión estable y soporta la especificación de servlets 2.4 (que es la que seguimos en este libro). Debemos notar que Tomcat necesita un JDK para funcionar. Tomcat es, sin dudas, el servidor de código abierto más maduro hoy en día. Usado por una enorme cantidad de usuarios y empresas de todo el mundo, con muchísimo soporte, tanto de la comunidad como del ámbito empresarial, tiene tanto prestigio que fue el servidor elegido por Sun para ser la implementación de referencia de las nuevas especificaciones de servlets y JSP que va desarrollando.

> UN SERVLET GENERA CONTENIDO DINÁMICO EN RESPUESTA A PETICIONES HTTP

- Apache Struts : framework de desarrollo web creado por Apache. Es prácticamente un estándar en la industria, y, si bien hay varias alternativas para hacer desarrollos web, es un framework completo, con miles de usuarios, mucho soporte y documentación.

Eclipse

Eclipse se puede bajar de www.eclipse.org/downloads No ahondaremos mucho sobre las características de Eclipse en sí, que son muchísimas, sino que simplemente explicaremos los conceptos de desarrollo web mediante su uso.

Una vez descargado Eclipse (hay versiones para Windows y Linux), se instala y queda creada una carpeta donde se encuentra el ejecutable. Eclipse no genera entradas en el registro de Windows ni crea archivos en otro lugar que no sea la carpeta de instalación.

Existen diferentes versiones de Eclipse que podremos descargar desde su página. Nosotros descargaremos Eclipse IDE for Java EE Developers y obtendremos una carpeta con todo lo necesario para el desarrollo web.

Instalar Tomcat

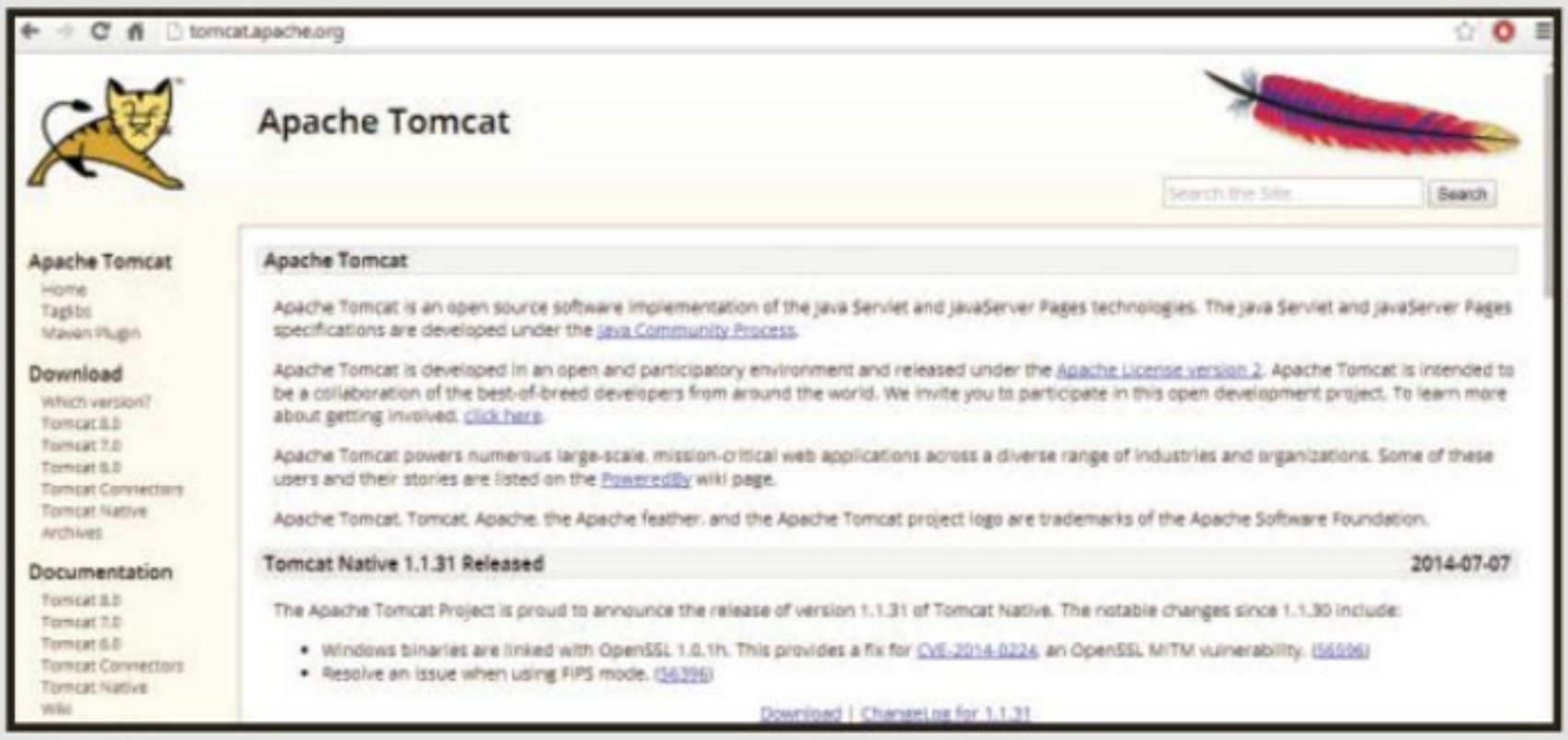
Vamos a usar Tomcat como servidor web y servlet container.
La instalación de Tomcat es sencilla, y vamos a integrarlo con Eclipse.

ECLIPSE

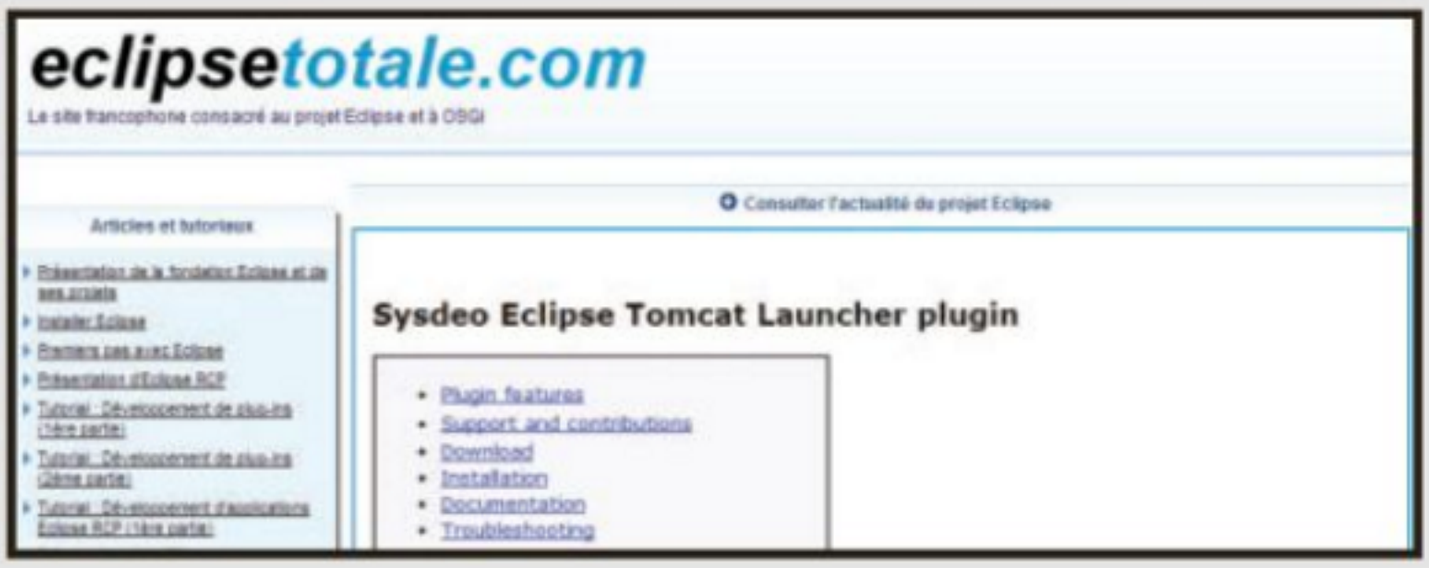
Eclipse es un producto gratuito desarrollado por Eclipse Foundation , una fundación sin fines de lucro que, en la actualidad, cuenta con el fuerte apoyo de más de 80 empresas, entre ellas IBM, Nokia, Intel, BEA, Sybase y Hewlett Packard. Fue fundado en el año 2004 por el Consorcio Eclipse.

PAP: INSTALAR TOMCAT

01 Para comenzar, descargue Tomcat desde http://apache.mesi.com.ar/tomcat . Si el enlace no funciona, busque algún otro servidor activo en la página de descargas de Tomcat: http://tomcat.apache.org .



02 Descomprima el contenido del archivo y elija dónde alojarlo. A modo de ejemplo, lo puede descomprimir en C:\tomcat-version . Para poder usar Tomcat con Eclipse, debe instalarle un plugin a este último, disponible en www.eclipsetotale.com/tomcatPlugin.html .



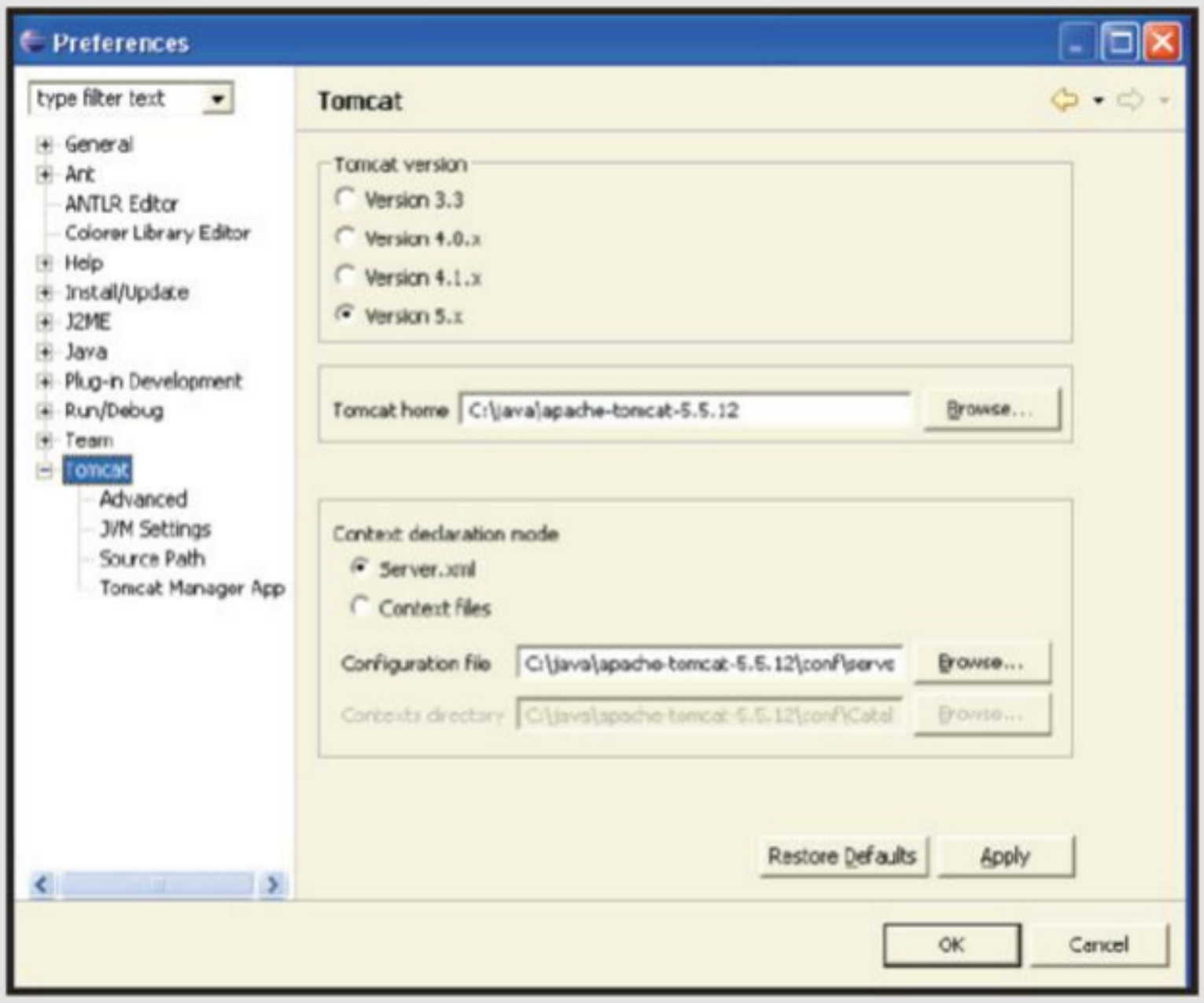
Este archivo es un .ZIP que contiene una carpeta llamada com.sysdeo.eclipse.tomcat_version . Esta carpeta es el plugin de Eclipse y, para instalarla, solo basta con copiarla (o moverla) dentro de la carpeta plugins de Eclipse.

Por ejemplo: si descomprimimos Eclipse en C:\eclipse, entonces debemos copiar (o mover) la carpeta descomprimida a C:\eclipse\plugins\ de forma tal que nos quede la carpeta C:\eclipse\plugins\com.sysdeo.eclipse.tomcat_version .

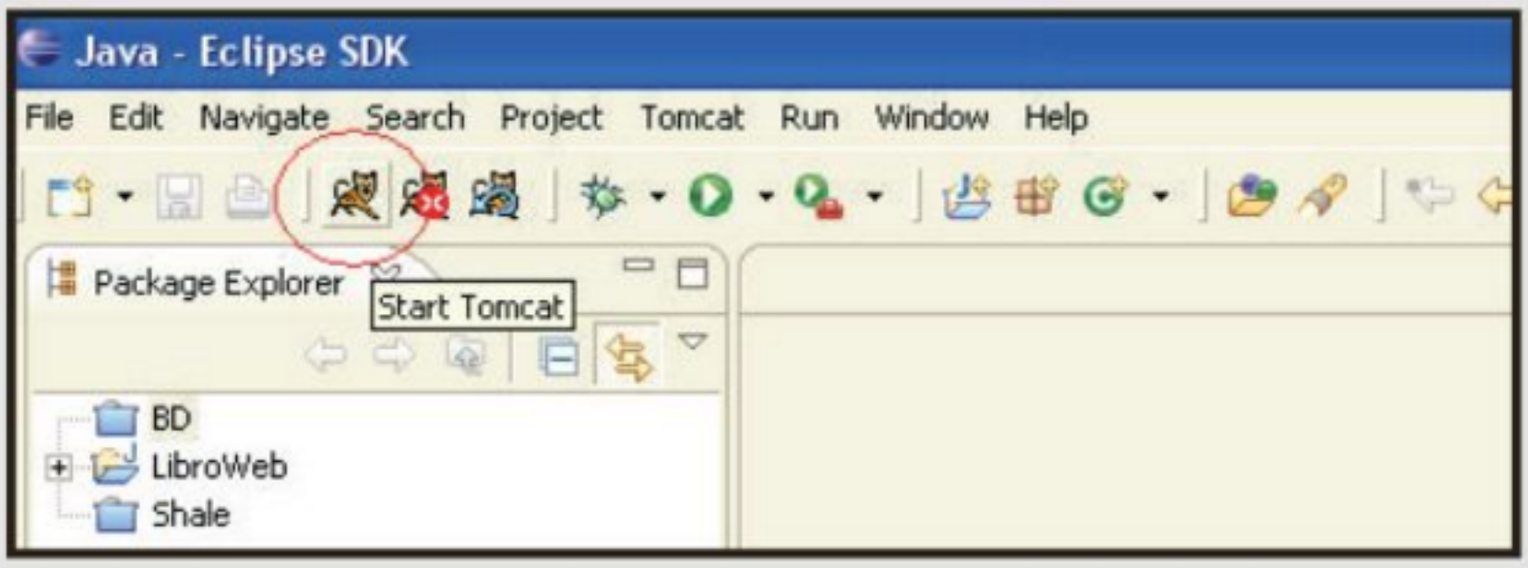
Una vez instalado el plugin Sysdeo , solo basta con iniciar (o reiniciar) Eclipse para que los cambios hagan efecto.

PAP: CONFIGURACIÓN Y EJECUCIÓN DE TOMCAT DESDE ECLIPSE

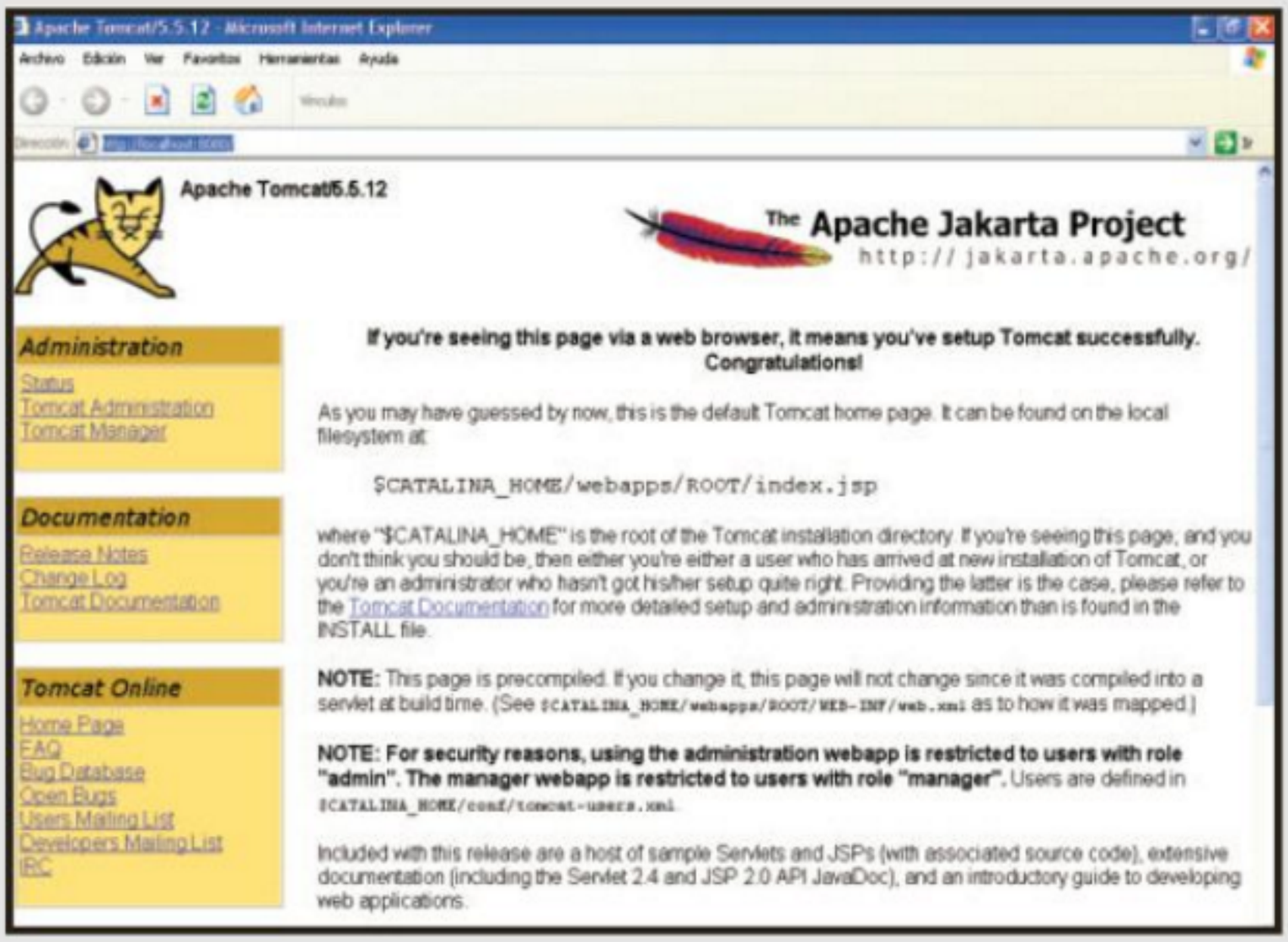
01 Para comenzar, acceda al panel de control de Eclipse desde Window/Preferences .



02 Seleccione la opción Tomcat en el árbol para ver las opciones del plugin. Debe seleccionar Version x.x como versión de Tomcat e ingresar la carpeta de instalación de Tomcat, por ejemplo: C:\tomcat-version .



03 Inicie la ejecución de Tomcat haciendo clic en el botón correspondiente.



Una vez ejecutado el servidor, por defecto estará configurado para escuchar pedidos en el puerto 8080. Bastará con abrir un navegador y escribir la dirección http://localhost:8080 para hacer correr Tomcat.

Ahora que tenemos un servlet container funcionando, vamos a ver cómo hacer para correr servlets.

Crearemos un nuevo proyecto en Eclipse, del tipo Tomcat Project, al que llamaremos Servlets. Para esto, iremos a File/New/Project y elegiremos el wizard Tomcat Project (ver Figura 5). Al resto de las opciones las dejamos como vienen por defecto.

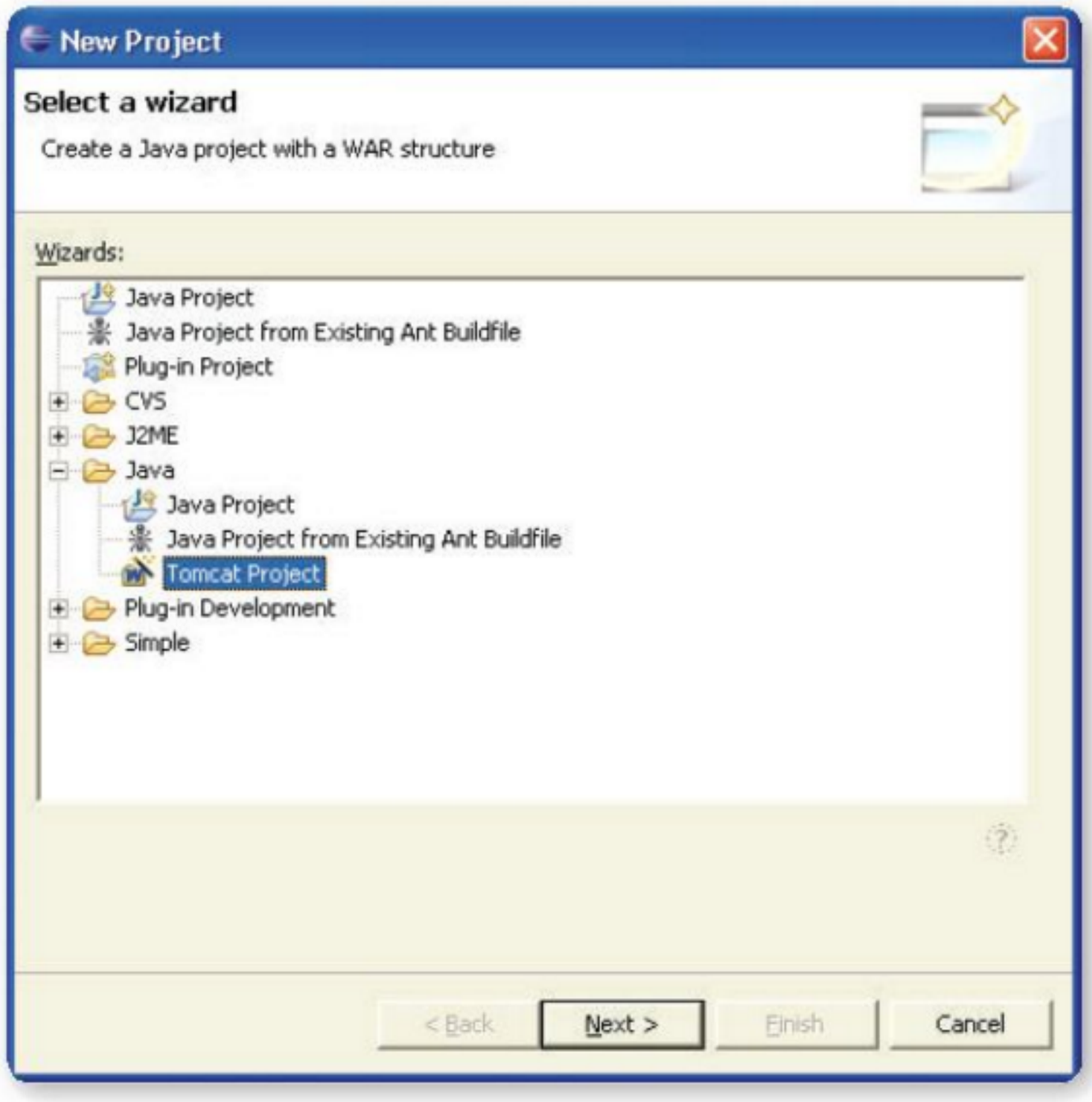
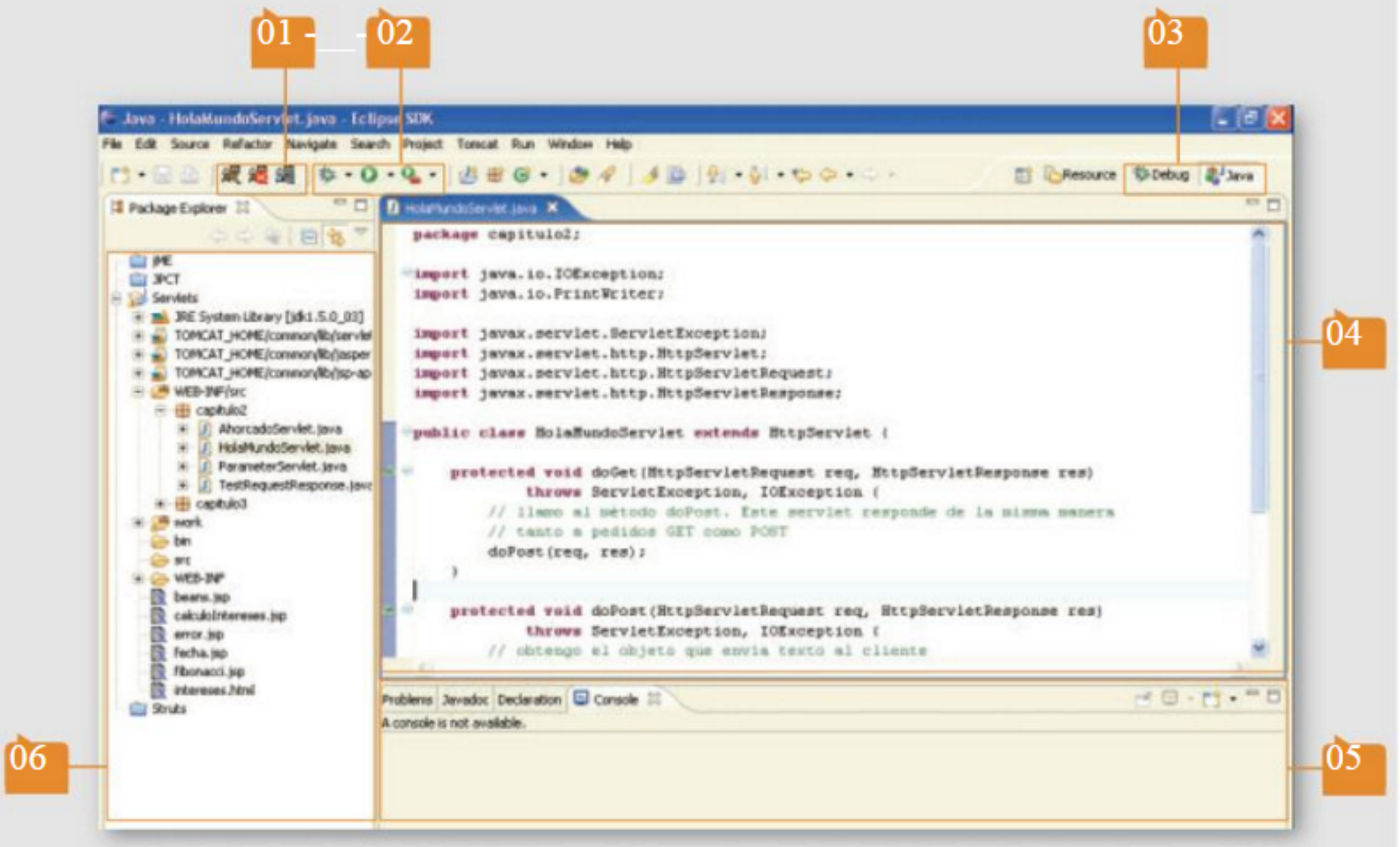


Figura 5. El proyecto de tipo Tomcat Project trae preconfigurada una estructura de directorios para poder desplegar los servlets y configuraciones.

Como resultado final, al ejecutar Eclipse, podremos ver la interfaz gráfica del IDE de la siguiente forma:

GV: COMPONENTES



01 TOMCAT: botones para arrancar, detener y reiniciar Tomcat.
Estos botones son provistos por el plugin SYSDEO.

02 EJECUCIÓN: botones para correr aplicaciones, depurarlas y ejecutar aplicaciones externas.

03 VISTAS: estos botones permiten intercambiar las distintas vistas posibles.
La vista Java es la más apropiada para el desarrollo de código.

04 CÓDIGO vista del código fuente, usando un editor que resalta la sintaxis propia de Java.

05 PANEL: el panel inferior suele ser usado para mostrar mensajes de la aplicación.
En este caso, vemos la consola de salida de Tomcat.

06 PANEL: los paneles son configurables y podemos agregar y quitar cuantas ventanas queramos. En este caso, tenemos una vista de la aplicación actual.

Contextos

Un servidor web puede tener corriendo varias aplicaciones web o contextos. Cada contexto puede ser tanto un sitio web completo como un módulo de una aplicación. Por ejemplo, muchos servidores compartidos corren una única instancia de Tomcat y tienen varios contextos que sirven a diferentes direcciones web. Cuando creamos el proyecto Tomcat en Eclipse, también definimos un contexto del mismo nombre en el Tomcat. Si pulsamos el botón derecho del mouse sobre el icono del proyecto y elegimos la opción Tomcat project, veremos que se presentan algunas opciones sobre el contexto que el proyecto define. Update context definition y Remove context definition agregan y quitan, respectivamente, la definición del contexto que el proyecto define al servidor web. De esta forma, podemos tener una instancia de Tomcat corriendo, y cargar y descargar contextos (aplicaciones web) de este sin necesidad de detener el servidor.

Primer proyecto

Al terminar la configuración, lo primero que vamos a desear es probar un servlet. Por lo tanto, en Eclipse vamos a File/New Project/Web/Dynamic web Project. Indicamos el nombre del proyecto, por ejemplo el clásico servletHolaMundo y asignamos, en Target runtime, la opción Apache Tomcat vx.0 (dependiendo de la versión que instalemos). Al crearse el proyecto, nos encontraremos con su contenido por defecto.

DYNAMIC CLASS LOADING

Tomcat tiene la capacidad de soportar la carga de clases de manera dinámica. Esto significa que al momento de modificar una clase que está realizando una carga y corriendo (como por ejemplo, un servlet), Tomcat detecta el cambio y, de ser necesario, reinicia también el contexto.

En Java Resources podremos poner los servlets pulsando el botón derecho del mouse en New/Servlet . Completamos la información necesaria de la ventana Create Servlet , como el nombre del paquete y la clase.

Presionamos el botón Next y de esta manera configuramos los métodos que deseamos para nuestro servlet; en este caso, solo dejaremos seleccionado el método doGet . Al finalizar, se generará el archivo HolaMundo.java con el contenido.

El código que debemos escribir dentro de doGet es el siguiente:

```
PrintWriter out;
                out = response.getWriter();
                out.println("<html>");
                out.println("<head><title>Ejemplo de Servlet</title></head>");
                out.println("<body>");
                out.println("<h1>Hola Mundo</h1>");
                out.println("</body></html>");
```

Este ejemplo se encuentra alojado en la web de RedUsers.

Al finalizar la codificación, hacemos clic en el botón RUN, seleccionamos el servidor que ya tenemos configurado y abrirá un navegador dentro de Eclipse con el resultado que programamos.

El archivo web.xml

La especificación define que la carpeta WEB-INF es donde debe ir toda la información relativa a la aplicación. En particular, define que dentro de ella debe existir el archivo web.xml . Este archivo es el deployment descriptor , un archivo XML que contiene toda la información de configuración de la aplicación. En el apéndice veremos una descripción completa de las posibilidades que ofrece este descriptor, pero por ahora echemos un vistazo a las funcionalidades más importantes que usaremos en este libro:

- display-name define el nombre que le queramos dar a nuestra aplicación.
- context-param contiene un par nombre-valor y define un parámetro de inicialización dentro del contexto. Pueden definirse 0 o más elementos context-param . Cualquier servlet podrá acceder a estos parámetros.
- filter y filter-mapping definen los filtros. En el capítulo sobre filtros veremos a fondo esta configuración.
- servlet y servlet-mapping contienen la definición de los servlets.

Veamos el código ejemplo por defecto:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<web-app version=”3.0” xmlns= ”http://java.sun.com/xml/ns/javaee”
xmlns:xsi=”http://ww w.w3.org/2001/XMLSchema-instance”
 xsi:schemaLocation=”http://jav  a.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd”>
    <servlet>
      <servlet-name>HolaMundoServlet</servlet-name>
      <servlet-class>org.prueba.servlet.HolaMundoServlet</servlet-class>
    </servlet>
    <servlet-mapping>
      <servlet-name>HolaMundoServlet</servlet-name>
      <url-pattern>/HolaMundoServlet</url-pattern>
    </servlet-mapping>
    <session-config>
      <session-timeout>30</session-timeout>
    </session-config>
</web-app>
```

- En servlet definimos los servlets que residirán en el contexto. Todo servlet que queramos usar debe estar declarado en el descriptor.
  Cada elemento debe contener un servlet-name , que es el nombre con

que será referenciado el servlet dentro del contexto, y un servlet-class , que es el nombre completo de la clase del servlet. Opcionalmente se puede definir uno o más init-param , que, similarmente al elemento context-param , consisten en un par param-name y param-value , que especifica nombre y valor del parámetro a definir, respectivamente. Estos parámetros de inicialización solo pueden ser vistos por el servlet en donde son definidos.

- servlet-mapping define una relación entre un servlet y un patrón de direcciones web. Es el que especifica qué servlet ha de cargarse según a qué dirección web (o URL , Uniform Resource Locator) se haya accedido. servlet-name indica a qué servlet se le está definiendo la relación (según el nombre definido previamente en el elemento servlet), y url-pattern especifica el patrón de direcciones web asociado.

Interactuar con el servlet

Para poder crear una aplicación web, es menester que podamos brindar interacción con el usuario. El servlet debe poder recibir información de entrada, procesarla y devolver una respuesta acorde. ¿Cómo enviarle datos al servlet? Básicamente hay dos formas, dependiendo de si usamos GET o POST como tipo de pedido HTTP (o ambos). Recordemos que GET, en teoría, se usaba únicamente para pedir un recurso; sin embargo, podemos usarlo para enviar cierto tipo limitado de información al servidor.

Un pedido GET no es ni más ni menos que una dirección web enviada al servidor (como vimos en el Capítulo 1). Cuando escribimos en el navegador una dirección web, estamos generando un pedido GET. Podemos también agregarle parámetros al pedido, y estos serán recibidos por el servidor. Para enviar parámetros, debemos agregar, luego de la dirección, el query string , que se construye de la siguiente forma:

> UN SERVLET PUEDE RECIBIR INFORMACIÓN DE ENTRADA Y DAR UNA RESPUESTA ACORDE

- un ‘ ?’ indicando que comienza la sección de parámetros;
- un par nombre y valor, separados por un ‘ =’;
- si hay más de un parámetro, estos se separan mediante ‘ &’.

Por ejemplo, si quiero pasar los parámetros “nombre” y “edad”, con valores “Juan” y “16” respectivamente, la dirección web resultante sería:

http://<sitio>/<recurso>?nombre=Juan&edad=16

El protocolo HTTP no impone ninguna restricción al largo de las direcciones web que pueden ser enviadas; sin embargo, los servidores sí. Aunque existen servidores que aceptan tamaños más grandes, por compatibilidad se recomienda que el tamaño total no supere los 256 caracteres.

Un pedido POST se crea utilizando formularios HTML. Al enviarse el formulario a la dirección especificada en el tag <form> (asumiendo que METHOD=”POST”), el navegador crea un pedido HTTP POST y lo envía a esa dirección. Veamos un pequeño ejemplo:

```
<form method=”post” action=”miServlet”>
        Nombre: <input type=”text” name=”nombre”><br/>
Edad: <input type=”text” name=”edad”><br/>
<input type=”submit”>
</form>
```

La clase HttpServletRequest abstrae el tipo de pedido que recibió el servlet, y sus métodos getParameter(String name) , getParameterMap() , getParameterNames() y getParameterValues(String name) sirven tanto para pedidos GET como POST.

Para probar el método POST y GET, vamos a crear dos archivos HTML usarGet.html y usarPost.html , con el siguiente código fuente.

Para el archivo usarGet.html :

```
<h1>Método GET</h1>
<form method=”GET” action= ”http://localhost:8080/EjemploUser/
UsarGetPost”>
    <input type=”submit”>
</form>
```

Para el archivo usarPost.html :

```
<h1>Método POST</h1>
<form method=”POST” action= ”http://localhost:8080/
EjemploUser/UsarGetPost”>
    <input type=”submit”>
</form>
```

Ahora creamos el servlet, UsarGetPost.java , y dentro del evento DoGet , escribimos:

```
response.setContentType(“text/html”);
                PrintWriter out = response.getWriter();
                out.println(“<html>”);
                out.println(“<body>”);
                out.println(“<h1>Hola Mundo (llamada GET)</h1>”);
                out.println(“</body>”);
                out.println(“</html>”);
```

Dentro del evento DoPost , escribimos:

```
response.setContentType("text/html");
                    PrintWriter out = response.getWriter();
                    out.println("<html>");
                    out.println("<body>");
                    out.println("<h1>Hola Mundo (llamada POST)</h1>");
                    out.println("</body>");
                    out.println("</html>");
```

Con este ejemplo, podremos saber cuándo estamos utilizando un método u otro. Este se encuentra disponible en el sitio web de RedUsers.

Problemática de los servlets

Los servlets son la piedra angular en el desarrollo de aplicaciones web con Java, y es muy importante que sepamos qué son y cómo funcionan. Sin embargo, los servlets no son la herramienta adecuada para el desarrollo web. Hacer una aplicación entera usando solo servlets resulta lento, engorroso y difícil de mantener. Los servlets plantean una problemática fundamental: no separan la presentación del contenido.

Hemos visto que en el servlet tenemos que imprimir el resultado completo, y esto incluye contenido HTML y contenido propio de la aplicación. Esto dificulta las cosas. Si quisiéramos cambiar el diseño gráfico de la aplicación, tendríamos que tocar el código fuente y recompilar. Pero, lo más importante: un diseñador gráfico que, generalmente, no sabe programar Java, tendría que modificar el contenido del servlet para agregarle diseño, por lo cual potencialmente podría dañar el código y generar problemas. En el próximo capítulo veremos una alternativa para esto: JavaServer Pages .

Un ejemplo completo

Vamos a desarrollar un ejemplo completo que abarque todo lo que vimos en este capítulo. Desarrollaremos el juego El Ahorcado .

La página HTML mostrará las letras elegidas, las acertadas y las chances restantes. El servlet deberá hacer uso de la sesión para guardar qué letras ya arriesgó el usuario y sobre qué palabra está adivinando (habrá varias). El número de chances será un parámetro de inicialización.

Veamos el código fuente:

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Random;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class AhorcadoServlet extends HttpServlet {

// Las palabras disponibles para jugar
   private static final String[] PALABRAS = { “GATO”,“CAPILLA”,“BABOR”,
         “MURCIELAGO”, “VENTANAL”, “HAMACA” };

   protected void doGet(HttpServletRequest req, HttpServletResponse res)
         throws ServletException, IOException {
      doPost(req, res);
   }

   protected void doPost(HttpServletRequest req, HttpServletResponse res)
         throws ServletException, IOException {

      HttpSession sesion = req.getSession();

      int maxIntentos = 5;
```

```
// La palabra sobre la que está adivinando
        String palabra = (String) sesion.getAttribute("palabra");

// Las letras que acertó
       String aciertos;

// Las letras que no acertó
        String errados;

/* Primera vez del usuario, no tiene palabra asignada */
       if (palabra == null) {
          Random rand = new Random();

/* Le agregamos una al azar de las disponibles */
          palabra = PALABRAS[rand.nextInt(PALABRAS.length)];
          aciertos = "";
          errados = "";

/* Guardamos los datos iniciales en "sesion" */
          sesion.setAttribute("palabra", palabra);
          sesion.setAttribute("aciertos", aciertos);
          sesion.setAttribute("errados", errados);
       }
       else {

/* Obtenemos los datos de este usuario de "sesion" */
          aciertos = (String) sesion.getAttribute("aciertos");
          errados = (String) sesion.getAttribute("errados");

/* Verificamos si la letra que arriesgó pertenece o no a la palabra */
          String letra = req.getParameter("letra");
          if (palabra.indexOf(letra) >= 0) {
             aciertos += letra;
```

```
            }
            else {
                errados += letra;
            }

// Guardamos los datos actualizados en "sesion"
            sesion.setAttribute("aciertos", aciertos);
            sesion.setAttribute("errados", errados);
        }

// Imprimimos el resultado
        PrintWriter out = res.getWriter();

        out.println("<html>");
        out.println("<head>");
        out.println("<title>AHORCADO</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>");

/* Iteramos por las letras de la palabra.
Si ya la acertó, la mostramos; si no, mostramos un "_" */
        for (int i = 0; i < palabra.length(); i++) {
            String letra = palabra.substring(i, i + 1);
            if (aciertos.indexOf(letra) >= 0) {
                out.println(" " + letra);
            }
            else {
                out.println(" _");
            }
        }
        out.println("</h1>");
        out.println("<br/>");
```

```
/* Nos fijamos si erró más de los intentos permitidos */
        if (maxIntentos > errados.length()) {

// Todavía está en juego
            out.println("Intentos: " + (maxIntentos - errados.length()));
            out.println("<br/>");

// Las chances restantes
            for (char c = 'A'; c <= 'Z'; c++) {
                if (aciertos.indexOf(Character.toString(c)) == -1
                        && errados.indexOf(Character.toString(c)) == -1) {

/* Mostramos letra como opción si no fue arriesgada aún */
                    out.println("<a href=\"AhorcadoServlet?letra=" + c + "\">" + c
                            + "</a>"); }
                }
            }
            else { // juego terminado

/* Invalidamos "sesion", limpiando todo su contenido */
                sesion.invalidate();
                out.println("<h2>Juego terminado!</h2>");
                out.println("<br/>");

/* Le damos la oportunidad de que juegue de nuevo */
                out.println("<a href=\"ahorcado\">Jugar de nuevo</a>");
            }
            out.println("</body>");
            out.println("</html>");
        }
}
```

Figura 6. El Ahorcado está compuesto por un solo servlet.

Como vemos, el juego usa mucho la sesión para guardar las letras que el usuario fue arriesgando. El juego va contando las letras arriesgadas por el jugador y, cuando superan el máximo número permitido, avisa al usuario que perdió. Sin embargo, no está realizada la funcionalidad para cuando el usuario acierta la palabra completa y gana. ¡Es un buen ejercicio para el lector!

En la página web de RedUsers encontraremos todos los ejemplos de este capítulo y, además, tres ejemplos de envío de parámetros.

RESUMEN

En este capítulo vimos la parte práctica de los conceptos que fueron desarrollados en el Capítulo 1 . Los servlets son la tecnología fundamental sobre la que se basa todo tipo de desarrollo web con Java. Vimos algunas aplicaciones, configuraciones y armamos un entorno de desarrollo. También pudimos ver que los servlets por sí solos plantean una problemática que hemos de resolver.

Actividades

TEST DE AUTOEVALUACIÓN

1. ¿Qué define la especificación Java Servlet Technology ?
2. ¿Qué objetos recibe un servlet al invocarse?
3. ¿Para qué se usa el método getLocale() de HttpServletRequest ?
4. ¿Dónde debe residir el deployment descriptor dentro de una aplicación web?
5. ¿Qué información contiene el deployment descriptor ?

EJERCICIOS PRÁCTICOS

1. Escriba un servlet que muestre todos los parámetros, tanto de contexto como de inicialización, con su nombre y valor.
2. Modifique el servlet TestRequestResponse para que responda en otro idioma de su elección, además de inglés y español.
3. Modifique el AhorcadoServlet para que le avise al usuario cuando ganó.
4. Modifique el AhorcadoServlet para que no repita nunca las palabras sobre las que el usuario ya arriesgó. ¿Es necesario usar la sesión?

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: profesor@redusers.com .

*03

JavaServer Pages

Una vez conocida la base del desarrollo de aplicaciones web con Java, los servlets, podemos comprobar que su utilización puede tornarse un poco difícil para grandes aplicaciones. En este capítulo veremos una alternativa más útil: JavaServer Pages (JSP).

¿Por qué no servlets?

En el Capítulo 2 vimos con detalle cómo crear páginas web dinámicas usando servlets. La principal dificultad de los servlets es que mezclan presentación con contenido. Como ya afirmamos, un diseñador gráfico tendría que tocar código fuente Java para poder aplicarle diseño a la página que creamos, y, a su vez, un programador tendría que tocar código lleno de elementos de diseño que dificultan el desarrollo.
Para ilustrar mejor este concepto, veamos un fragmento de una página web con mucho diseño gráfico embebida en un servlet.

```
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;

public class ServletConDisenio extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {

        doPost(req, res);
    }

    protected void doPost(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        PrintWriter out = res.getWriter();

        // la respuesta
        out.println("<html>");
        out.println("<head>");
```

```
        out.println("<title>Bienvenidos</title>");
        out.println("<meta http-equiv=\"Content-Type\"
content=\"text/html; charset=iso-8859-1\">
");
        out.println("<link href=\"css/estilos.css\" rel=\"
stylesheet\" type=\"text/css\">");
        out.println("</head>");
        out.println("<body class="body">");
        out.println("<table width=\"740\" border=\"0\" align=\"
center\" cellpadding=\"0\" cellspacing=\"0\">");
        out.println("<tr>");
            for (int i=0; i < 5; i++) {
        out.println("<td width=\"185\" height=\"64\" align=\"left\" valign=\"
middle\" bgcolor=\"#1AA440\"><a href=\".\"><img src=\"/_images/common/
logo"+i+".gif\"width=\"179\" height=\"60\" border=\"0\"></a></td>");
                    }
        out.println("</tr>");
        out.println("</table>");
        out.println("</body>");
        out.println("</html>");

        // cerramos el stream
        out.flush();
        out.close();
    }
}
```

Este código es muy difícil de leer y mantener. Tenemos que escapar cada comilla del código HTML, en el código Java, con \" , complicando su lectura. Además, si el diseñador gráfico quisiera cambiar algo de la estética, tendría que modificar el código fuente y reiniciar el servidor para que tome los cambios.

Por estos motivos, se desarrolló una solución: JavaServer Pages .

JavaServer Pages

JavaServer Pages , como su nombre lo indica, son páginas. Esta tecnología propone un cambio al modo en que se desarrollan páginas web con Java. En vez de escribir un servlet que, al ejecutarse, devuelva el contenido (tanto estático como dinámico), lo que propone es crear una página web con el contenido estático y el dinámico. Esto a priori suena igual que un servlet: ¿dónde radica la diferencia? La mejor forma de entender de qué trata JavaServer Pages es con un ejemplo. Veamos:

```
<html>
<head>
<title>Fecha en JSP</title>
</head>
<body>

Hola!<br/>
La fecha de hoy es <b><%= new java.util.Date() %></b>

</body>
</html>
```

JSPC

Algunas implementaciones (Apache Tomcat , entre ellas) proveen un compilador de JSP . Compilar el JSP se refiere al proceso de traducirlo a servlet, compilarlo y dejarlo listo para ser accedido por los clientes. De esta forma, se evita que el proceso de compilación se produzca al ser accedido el JSP por primera vez, lo cual puede tomar mucho tiempo y retrasar la respuesta al cliente.

Si creamos este archivo en la carpeta raíz de nuestro proyecto Tomcat en Eclipse, arrancamos el servidor Tomcat y apuntamos el navegador a http://localhost:8080/Servlets/fecha.jsp (asumiendo que nuestro proyecto se llama Servlets), veremos algo similar a la Figura 1.

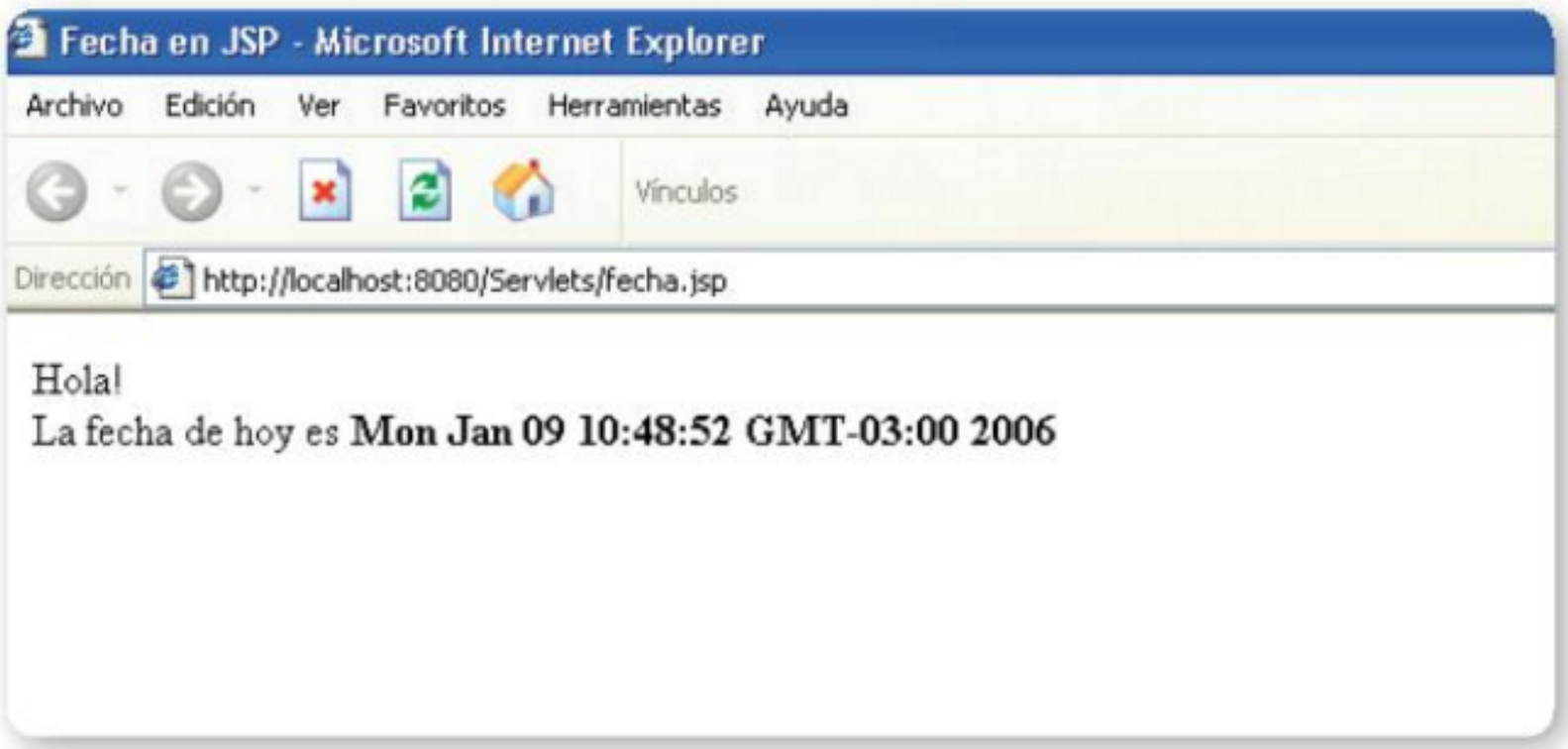


Figura 1. Las JavaServer Pages llevan por defecto la extensión .JSP.

Como vemos, una página JavaServer Pages (en adelante, página JSP) tiene contenido estático (HTML) y contenido dinámico. En este capítulo, vamos a ver en detalle cómo crear y definir el contenido dinámico en una página JSP; por ahora, concentrémonos en el ejemplo de la línea que sigue:

```
La fecha de hoy es <b><%= new java.util.Date() %></b>
```

Vemos que el contenido estático es simplemente código HTML, y el contenido dinámico se diferencia mediante ciertos tags. Al igual que HTML, las páginas JSP definen ciertos tags específicos para determinar sentencias propias, marcadores de contenido dinámico, de código Java, etcétera.

Antes de meternos de lleno en cómo programar páginas JSP, es importante aclarar algo acerca del funcionamiento de esta tecnología. JavaServer Pages surgió como una especificación (al igual que los servlets) en base a las necesidades de simplificar el desarrollo de aplicaciones web.

Esta simplificación, con respecto a los servlets, es lograda; sin embargo, detrás de escena, ¡una página JSP es un servlet!

Cuando accedemos a una página JSP por primera vez, un traductor se encarga de transformarla en un servlet, se compila y luego se ejecuta. Por eso notaremos que, al acceder a un JSP por primera vez o luego de haberle hecho alguna modificación, toma más tiempo en devolvernos la página que el resto de las veces; esto se debe a que se está transformando y compilando.

La buena noticia es que se puede acceder fácilmente al servlet generado, aunque, al ser código generado, es de muy difícil lectura. Por defecto, los servlets generados se guardan en la carpeta work.

La mala noticia es que si la página JSP no está bien programada y, al ejecutarla, se genera una excepción, esta ocurre en el servlet generado, que es en definitiva el que se ejecuta. El error nos informará cuál es la línea y la excepción arrojada –pero del servlet–, y este tipo de errores suele resultar muy confuso porque muchas veces no refleja realmente lo que produjo la excepción. Con las actuales versiones de JavaServer Pages se han incorporado muchas mejoras al respecto.

Sintaxis JSP

A continuación, vamos a definir algo de sintaxis JSP y luego la aplicaremos en algunos ejemplos.

JSP COMO XML

Desde su versión 2.0, la sintaxis JSP tiene una alternativa XML, para que la página en definitiva termine siendo además un documento XML, y así poder procesarlo, transformarlo, etcétera. En el sitio web www.oracle.com/technetwork/java/index.html podemos encontrar documentos sobre la sintaxis del formato JSP como XML.

Comentarios

JSP acepta dos tipos de comentarios. ¿Suena extraño? ¡Lo es! Nuevamente, un ejemplo para ilustrar; luego, presentaremos las explicaciones pertinentes. Veamos un código con comentarios:

```
<html>
<head>
<title>Pagina con comentarios</title>
</head>
<body>

<!-- Inicia sección tablas datos -->
<table>
...
</table>

<%-- Guardar resultado en la base de datos --%>
<% ... %>

</body>
</html>
```

El primer tipo de comentario es un comentario a nivel HTML , y el segundo es un comentario JSP . El primer comentario, al ser HTML, es enviado junto con el resto de la respuesta al cliente. El segundo, al ser un comentario del código JSP, no es incluido en la respuesta. Si accedemos a esta página JSP y vemos su código fuente HTML, veríamos algo parecido a esto:

```
<html>
<head>
<title>Pagina con comentarios</title>
</head>
<body>
```

```
<!-- Inicia sección tablas datos -->
<table>
...
</table>

</body>
</html>
```

En resumen:

```
<!-- Comentario HTML -->
<%-- Comentario JSP -->
```

Fragmento de código

Para introducir código Java en una página JSP, debemos utilizar los tags <% codigo %> . Recordemos que este código terminará siendo embebido dentro del servlet resultante, por lo que debe compilar al igual que cualquier programa Java, no olvidar los ; finales, etcétera.

Declaraciones

Podemos declarar métodos y variables a ser utilizadas luego en la página usando los tags <%! declaraciones %> . Estas declaraciones serán insertadas al comienzo de la clase (no importa en qué lugar de la página JSP las declaremos), por lo que serán variables y métodos de instancia.

Expresiones

En JSP una expresión es una porción de código que se evalúa y su resultado se imprime. Los tags de utilización son <%= expresión %> . Un detalle importante a notar es que el contenido de esta expresión no debe finalizarse con ; .

Un ejemplo integrador

Veamos un ejemplo que utilice los tags vistos hasta ahora:

```
<html>
<head>
<title>N&uacute;meros de Fibonacci</title>
</head>
<body>

<%-- Declaramos el método fibonacci --%>
<%!
	private int fibonacci(int n) {
		if (n == 0) {
			return 0;
		}
		if (n == 1) {
			return 1;
		}
		else {
			return fibonacci(n-1)+fibonacci(n-2);
		}
	}
%>

Los primeros 10 n&uacute;meros de Fibonacci:<br/><br/>
<%
for (int i=0; i < 10; i++) {
%>
	Fibonacci(<%= i %>) = <%= fibonacci(i) %><br/>
<%
}
%>

</body>
</html>
```

Al acceder a la página, vemos el resultado como en la siguiente figura:

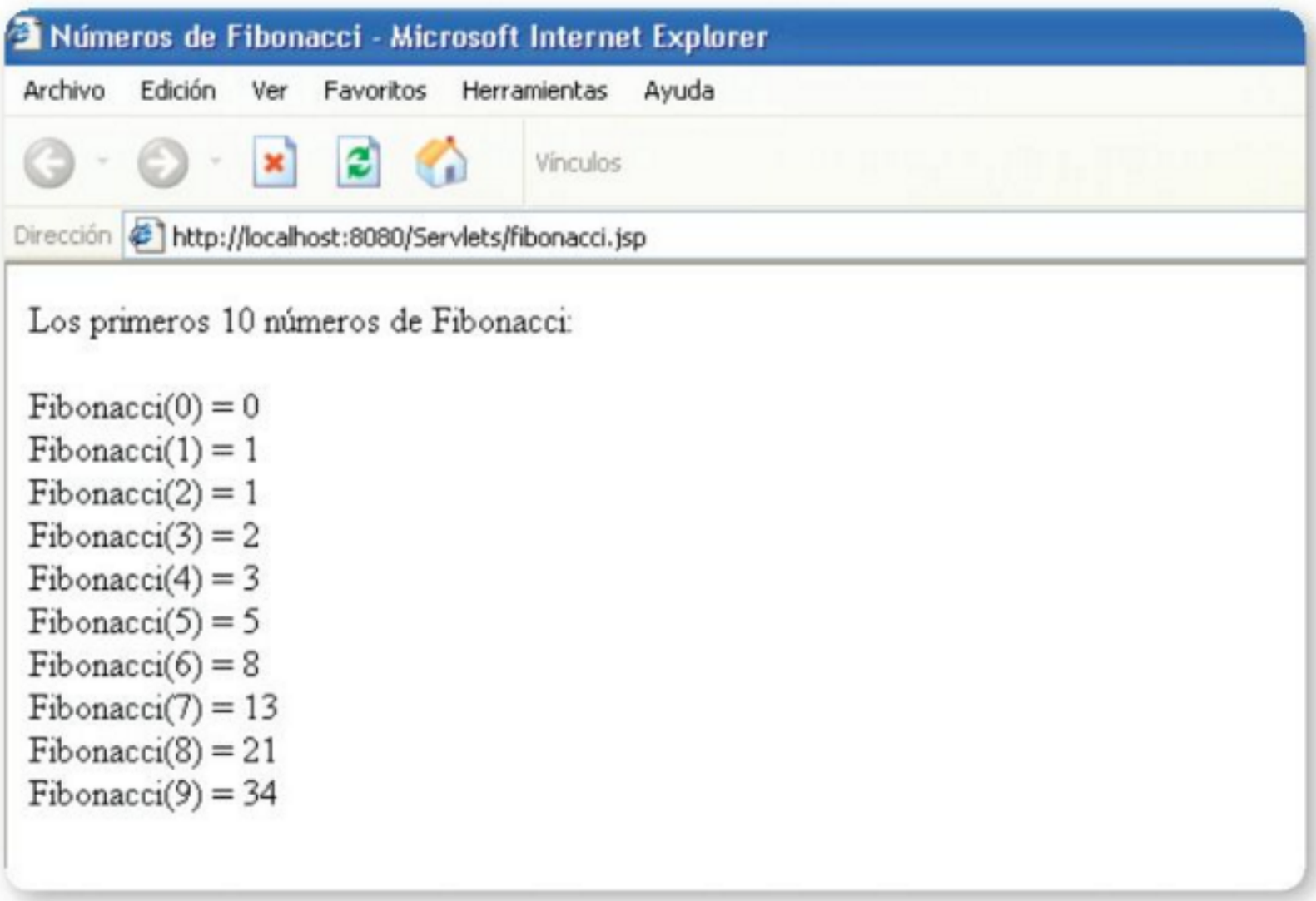


Figura 2. La página es traducida y compilada una sola vez y ejecutada en cada acceso.

Al haber declarado el método usando los tags de declaración, sabemos que podemos llamarlo desde cualquier lugar de la página y estará accesible. No ocurre lo mismo en la línea siguiente

```
Fibonacci(<%= i %>) = <%= fibonacci(i) %><br/>
```

SOBRE LAS DECLARACIONES

Bien podríamos efectuar las declaraciones en fragmentos de código; no es necesario que usemos explícitamente los tags de declaración. Sin embargo, usándolos, las variables y métodos que declaremos estarán definidos como variables y métodos de instancia y, por ende, serán accesibles desde cualquier parte del código.

con la variable i, que es accesible solamente porque está dentro del ciclo for definido previamente.

Variables implícitas

Al programar servlets, teníamos acceso a una serie de variables que necesitábamos para poder interactuar con el entorno y con los parámetros de entrada y salida, como el pedido originario (HttpServletRequest), el resultado de salida (HttpServletResponse), la sesión (mediante el pedido), etcétera. Hasta ahora no hemos visto cómo acceder a estas variables en una página JSP. Como esta termina siendo, en definitiva, un servlet, tenemos acceso a ciertas variables implícitas que podemos asumir que existen y están instanciadas.

En la siguiente tabla tenemos un listado de las variables y su tipo.

VARIABLES IMPLÍCITAS EN JSP	
▼VARIABLE	▼TIPO
request	subclase de ServletRequest
response	subclase de ServletResponse
pageContext	PageContext
session	HttpSession
application	ServletContext
out	JspWriter
config	ServletConfig
page	Object
exception	Throwable

Tabla 1. Listado de variables implícitas.

- request y response: son los objetos que recibe el servlet como parámetro para su ejecución.
- pageContext : contiene información sobre el contexto donde se ejecuta la página JSP y provee acceso a variables de contexto.
- session : es el resultado de ejecutar pageContext.getSession() .
- application : es el resultado de ejecutar pageContext.getServletContext() .
- out : es el resultado de ejecutar pageContext.getSession() .
- config : es el resultado de ejecutar pageContext.getSession() .
- page : es el servlet resultante luego de ser transformado; esta variable no suele ser usada en páginas JSP.
- exception : es definida solamente cuando la página JSP es declarada como isErrorPage . Más adelante veremos un poco más sobre esto.

Directivas de página

Un tag especial que se aplica a las páginas JSP es <%@ page %> .

Este tag define una serie de atributos que se usan para configurar la página web. Los atributos configurables son los siguientes:

- language : JavaServer Pages es una especificación que, en teoría, permite cualquier lenguaje en su contenido. Hoy día, solo Java es soportado, por lo que este atributo no tiene mucho sentido de ser explicitado. Su valor por defecto es: Java.

JSP MULTI-THREADED

Debemos ser muy cautelosos si creamos una página con isThreadSafe=false .
Si esa página es accedida por varios usuarios, puede ser un cuello de botella y frenar la navegación enormemente. Como regla general, debemos procurar escribir las páginas JSP cuidando que varios hilos puedan ejecutarla sin problemas.

- extends : podemos definir qué clase ha de extender el servlet resultante. Por defecto, el servlet container es el que define qué clase ha de extender, por lo tanto, no se recomienda cambiar este valor.
- import : con este atributo podemos importar tantas clases o paquetes como queramos para ser utilizados dentro de la página. Podemos definir todas las clases y paquetes dentro de una sola sentencia import , separándolos por comas, o declarar varias sentencias import .
- session : mediante este atributo definimos si el usuario va a tener acceso a la sesión o no (valores true / false). Valor por defecto: true . Si declaramos este atributo con valor false , no tendremos acceso a la variable implícita session .
- buffer : la variable implícita out (de tipo JspWriter) contiene un búfer interno para manejar el contenido que se va devolviendo al navegador. Con este atributo definimos el tamaño del búfer (en kilobytes) o none si no queremos búfer de salida. Valor por defecto: 8 Kb.
- autoFlush : si este atributo es true , entonces, al llenarse el búfer de salida, los datos serán automáticamente enviados al cliente. Si es false y el búfer se llena, se produce una excepción. Notemos que no es posible definir este atributo como false si también definimos el tamaño del búfer como none . Valor por defecto: true .
- isThreadSafe : con este atributo declaramos si la página es segura para el acceso multi-threaded. Si lo especificamos como true , el container puede mandar varios pedidos concurrentes a esta página; si no, serializará los pedidos mandando de a uno por vez. Valor por defecto: true .

- info: en este atributo podemos especificar un texto que será devuelto mediante el método getServletInfo() .
- errorPage : aquí podemos definir una página para manejo de errores. Si una excepción ocurre durante la ejecución de esta página, entonces se redirige el pedido a la página de error que esté declarada en esta variable.
- isErrorPage : con este atributo distinguimos las páginas de error. Estas tienen acceso a la variable implícita exception que contiene la excepción ocurrida.
- contentType : este atributo identifica el tipo de contenido MIME que estamos devolviendo. Opcionalmente podemos definir también el mapa de caracteres que usamos para codificar. Valor por defecto: text/html;charset=ISO-8859-1 .
- pageEncoding : aquí podemos definir el mapa de caracteres de la codificación de la página. Valor por defecto: ISO-8859-1 .
- isELIgnored : JSP 2.0 incorpora soporte para EL (Expression Language o lenguaje de expresiones). Al ser una sintaxis nueva, por compatibilidad podemos optar por pedirle al traductor que la ignore. Valor por defecto: false .

Veamos ahora algunos ejemplos de directivas de página:

```
<%@ page import="java.util.*" errorPage="error.jsp" contentType="text/plain"%>

<%@ page info="JSP que maneja el panel de control" isELignored="true"%>

<%@ page buffer="none" autoFlush="false"%>      ¡INCORRECTO!
```

JSP y JavaBeans

JavaServer Pages fue diseñada teniendo presente la integración con JavaBeans . Ya desde la versión 1.0 se definían los tags que se usan para instanciar y acceder a JavaBeans. Para aquellos lectores no familiarizados con los JavaBeans, haremos una pequeña introducción.

JavaBeans

Los componentes JavaBeans no son más que clases Java que siguen ciertas convenciones. No es necesario que extiendan alguna clase ni que implementen ninguna interfaz, simplemente deben proveer métodos para poder acceder a sus propiedades de una manera estándar. Si una clase Java cualquiera cumple con estas condiciones,
la clase puede ser considerada un JavaBean.
Estas convenciones no son excluyentes,
la clase puede, además de los métodos que acceden a las propiedades,
definir cualquier cantidad de métodos propios, extender cualquier clase
e implementar cualquier interfaz. Veamos las convenciones:

> LOS JAVABEANS PROVEEN MÉTODOS PARA ACCEDER A SUS PROPIEDADES DE MODO ESTÁNDAR

- Dada una propiedad, podemos definirla como de lectura y/o de escritura.
- Una propiedad no necesariamente debe estar respaldada por una variable de instancia; basta con que implemente los métodos requeridos.
- Si una propiedad es de lectura, entonces debe definir un método getter para accederla. Este método debe ser público y de la forma public [tipo] get[nombre de la propiedad]() .
- Si una propiedad es de escritura, entonces debe definir un método setter para escribirla. Este método debe ser público y de la forma public void set[nombre de la propiedad]([tipo] nombre) .
- Además, un componente JavaBean debe proveer un constructor que no tome parámetro alguno.

Notemos que los nombres de los métodos de acceso deberán ser camel-case , o sea, seguir la convención de mayúsculas que Java propone. Si queremos implementar un getter sobre una propiedad de nombre peso , entonces el método deberá llamarse getPeso (notar la P mayúscula).

Veamos un ejemplo sobre los métodos getter y setter para aclarar la sintaxis abstracta que hemos definido. Supongamos tenemos la propiedad

de tipo string de nombre descripcion y queremos que esta propiedad pueda leerse y escribirse. Entonces, debemos implementar estos dos métodos:

```
public String getDescripcion() { ... }

public void setDescripcion(String str) { ... }
```

Si la propiedad es de tipo booleano (ya sea del primitivo boolean o de la clase Boolean), el método accesor (getter) debe llamarse isPropiedad(); en cualquier otro caso: getPropiedad().

Por lo general, las propiedades de los componentes JavaBean están relacionadas una a una con variables de instancia, pero esto no es necesario. Veamos un ejemplo completo de un componente JavaBean:

```
public class Circunferencia {

    /**
     * Radio
     */
    private double radio;

    /**
     * Posición del centro
     */
    private int x, y;

        /**
         * Hueco o lleno
         */
        private boolean hueco;

/**
 * Un constructor sin parámetros
 */
public Circunferencia() {
    super();
```

```
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    public void setRadio(double r) {
        this.radio = r;
    }

    public double getArea() {
        return Math.PI * radio * radio;
    }

        public boolean isHueco() {
                return hueco;
        }

    /**
     * Para establecer el área nueva y ser consistentes, debemos modificar el radio
     * @param al área nueva que queremos para la circunferencia
     */
    public void setArea(double area) {
        radio = Math.sqrt(area / Math.PI);
    }
}
```

En el ejemplo, definimos una clase llamada Circunferencia . La clase tiene variables de instancia que, a su vez, son propiedades del componente JavaBean. Las variables x e y que identifican la posición de la circunferencia en el plano son tanto de lectura como de escritura, por lo que hemos definido métodos getter y setter . El radio es una propiedad de solo escritura, por lo que definimos solo un método setter . Tenemos una propiedad booleana, cuyo método getter empieza con is. A su vez, tenemos la propiedad area , que, según vemos, es de lectura y escritura, y esta propiedad no está directamente relacionada con una variable de instancia.

Scopes

Antes de ver cómo interactuar con JavaBeans desde un JSP, veamos primero los tipos de alcance (scopes) que existen y su utilización. Cuando definimos un JavaBean desde un JSP, como veremos luego, debemos darle un alcance. Este alcance también puede ser pensado como visibilidad . Se define un componente con cierto alcance, y puede ser visto por otros objetos que están en el mismo alcance que el componente. Los tipos de alcance son:

TIPOS DE ALCANCE

TIPO	UTILIZACIÓN
application	El objeto es visible desde todo el contexto.
session	El objeto permanece en la sesión del usuario.
request	El objeto es accesible desde cualquier componente que tenga acceso al pedido.
page	El objeto solamente puede ser accedido por la página JSP que lo creó.

Tabla 2. Scopes.

Usar JavaBeans desde un JSP

JSP define ciertos tags específicos para la interacción con JavaBeans. La idea es delegar todo tipo de lógica de negocios en los JavaBeans y limitar la funcionalidad de las páginas JSP a la interacción con el usuario, manejo de excepciones, redirección a páginas, etcétera. De esta forma, tenemos componentes reutilizables (la principal cualidad deseable de un JavaBean) a través de la aplicación web.

Básicamente la interacción con un JavaBean se reduce a:

- crearlo u obtener una instancia;
- acceder a propiedades; y
- escribirle propiedades.

jsp:useBean tiene un doble propósito. Sirve tanto para crear un bean como para acceder a uno ya creado. Si el bean ya existe en el scope, se obtiene una referencia, y si no existe, se lo crea. La sintaxis es:

```
<jsp:useBean id=”beanName”
scope=”page|request|session|application”
class=”package.class” [ type=”package.class” ] />
```

Otra forma, asignando atributos al crear (o localizar) directamente, es:

```
<jsp:useBean id=”beanName”
scope=”page|request|session|application”
class=”package.class” [ type=”package.class” ]>
        <jsp:setProperty ... />
<jsp:setProperty ... />
...
</jsp:useBean>
```

La segunda forma usa un tag que veremos luego, que se utiliza para escribir propiedades a un bean.

- id es el nombre del bean. Esto es análogo al nombre de una variable. En el caso de que el bean se cree, se creará con ese nombre; si ya existe un bean de ese nombre (en el scope definido), se devuelve una referencia a ese bean.
- scope define el alcance del bean, según lo definido previamente. Si no especificamos este atributo, el valor por defecto es page .
- class es el nombre completo de la clase que este bean ha de instanciar. Como todo JavaBean provee un constructor sin parámetros, siempre podremos crear un bean sin necesidad de contar con otro dato que no sea el nombre de la clase a instanciar.
- type determina el tipo del bean a instanciar. Esto puede sonar confuso. ¿No estamos declarando la clase a instanciar en el atributo class ? El atributo type es opcional: si no lo definimos, el tipo del bean será el mismo que la clase que se instancia. Sin embargo, podemos querer que el tipo sea una superclase del bean instanciado o una interfaz que se implemente.

Ejemplos:

```
<jsp:useBean id=”circ1” class=”Circunferencia” />

<jsp:useBean id=”circ2” scope=”session”
class=”Circunferencia” type=”FiguraGeometrica” />
```

JAVABEANS EN JSP

Usar JavaBeans en las páginas JSP es una excelente forma de separar responsabilidades. Los JavaBeans se encargan de la lógica de los objetos de negocios sin saber siquiera que están siendo usados en un entorno web, y las páginas JSP se ocupan de la presentación.

Veamos el código generado para el primer ejemplo. Un detalle a notar es que cada implementación es libre de generar el código que quiera al traducir un JSP a servlet; sin embargo, sí o sí tiene que ser consistente con lo que pide la especificación. En nuestro caso particular, Jasper 2 , el componente de Tomcat encargado de traducir los JSP, generó la siguiente porción de código:

```
capitulo3.Circunferencia circ1 = null;
synchronized (_jspx_page_context) {
    circ1 = (capitulo3.Circunferencia)
_jspx_page_context.getAttribute("circ1",
PageContext.PAGE_SCOPE);
    if (circ1 == null) {
        circ1 = new capitulo3.Circunferencia();
        _jspx_page_context.setAttribute("circ1", circ1,
                PageContext.PAGE_SCOPE);
    }
}
```

Como vemos, el código hace exactamente lo esperado. Declara una variable de tipo Circunferencia , de nombre circ1, e intenta obtenerla del scope de página. En caso de que no exista, el bean es creado y guardado en dicho scope.

JASPER 2

Jasper 2 es un componente muy poderoso y configurable. En un entorno de producción con mucho requerimiento de procesador, puede ser útil configurarlo paramejorar su performance. En la página web http://tomcat.apache.org/ encontraremos información específica.

jsp:getProperty se usa para obtener una propiedad de un bean.
Su sintaxis es muy sencilla:

```
<jsp:getProperty name=”beanName”
property=”propertyName” />
```

- name debe ser un bean ya creado u obtenido previamente mediante jsp:useBean .
- property es el nombre de la propiedad que deseamos acceder.

Ejemplos:

```
<jsp:getProperty name=”circ1” property=”x” />
<jsp:getProperty name=”circ2” property=”area” />
```

jsp:setProperty es utilizado para escribir propiedades en el bean.
Su sintaxis es un tanto complicada, veámosla con detalle:

```
<jsp:setProperty name=”beanName”
{ property=”*” |
property=”propertyName” [ param=”parameterName” ] |
property=”propertyName”
value=”{string | ‘${‘ Expression ‘}’ | <%= expression %>}” }
/>
```

- name : debe ser un bean ya creado u obtenido previamente mediante jsp:useBean .
- property : es el nombre de la propiedad que vamos a escribir, o “*” para indicar todas las propiedades del mismo nombre que un parámetro del request.
- param : es el nombre del parámetro del request que tiene el valor a asignarle a la propiedad.

- value : es el valor, constante o en forma de expresión, a asignarle a la propiedad.

Veamos unos ejemplos clarificadores:

```
<!-- Establecemos en 5 el valor de la propiedad x
del bean circ1 -->
<jsp:setProperty name="circ1" property="x" value="5" />

<!-- Cambiamos el valor de la propiedad y de circ1
al resultado de la expresión -->
<jsp:setProperty name="circ1" property="y"
value="<%= Math.sqrt(49) %>" />

<!-- Seteamos el valor de la propiedad radio de
circ1 al valor del parámetro del mismo nombre (radio) -->
<jsp:setProperty name="circ1" property="radio" />

<!-- Cambiamos el valor de la propiedad
radio de circ1 al valor del parámetro 'ratio' -->
<jsp:setProperty name="circ1" property="radio" param="ratio"/>

<!-- Cambiamos todos los valores de las propiedades
del bean circ1 a los valores de los parámetros del mismo nombre -->
<jsp:setProperty name="circ1" property="*" />
```

En el último ejemplo, para cada parámetro que contenga el request, intentará escribir el valor del parámetro en una propiedad del mismo nombre, si existiera.

Inclusión de páginas

Si intentamos apuntar a la reutilización de código (y también páginas), podemos incluir otros documentos (sean JSP, HTML o lo que queramos)

dentro de una página JSP. La idea básica es abstraer funcionalidades y lograr una mayor modularización del sistema. Típicamente, los encabezados, pies de página, firmas, etcétera, son implementados como páginas incluidas. Un ejemplo más complicado es la seguridad de la página. Si queremos verificar en cada pedido de cada página que el usuario esté autenticado y autorizado a su vez para ver la página en cuestión, entonces en toda página JSP incluiremos la funcionalidad de verificar la seguridad.

JSP provee dos formas de incluir contenido: inclusión estática e inclusión dinámica. El primer tipo es equivalente a insertar el código, ya sea estático (una página HTML, por ejemplo) o dinámico (un JSP), en el documento. Al traducirse la página a servlet se inserta el contenido. La sintaxis es:

```
<%@ include file=”archivo”%>
```

El segundo tipo también permite incluir cualquier tipo de documento (estático o dinámico), aunque tiene más sentido al incluir páginas dinámicas. Este tipo de inclusión se procesa durante la ejecución del JSP. Cuando un pedido llega al JSP, es redirigido a cada documento incluido del segundo tipo, se procesa y el resultado es entonces incorporado en el documento que lo incluye.

La sintaxis para inclusiones del segundo tipo es:

```
<jsp:include page=”includedPage” />
```

Un ejemplo completo

Vamos a desarrollar una aplicación con páginas JSP para ejemplificar los conceptos del capítulo. La aplicación consiste en un calculador de intereses para plazos fijos. Dada la inversión inicial, la cantidad de meses que será depositado el dinero y la tasa nominal anual que el banco paga, podremos saber las ganancias obtenidas.

Desarrollaremos la aplicación de la siguiente forma: un bean contendrá la información (inversión, meses y tasa) y además será el encargado de efectuar el cálculo con los datos. Una página HTML (sin funcionalidad JSP) será la puerta de entrada a la aplicación. En ella, se pedirán los datos y se reenviará la información al JSP. Este recibirá el pedido, creará el bean, escribirá los valores de sus propiedades según los valores del pedido y obtendrá el resultado (también una propiedad, pero en este caso, solo de lectura).

Además, la página JSP definirá una página de error por si ocurriera una excepción en la entrada de datos (en este caso, si se ingresara un dato que no fuera un número).

intereses.html

```
<html>
<head>
<title>Calculador de intereses JSP</title>
</head>
<body>

<!-- Los datos del formulario los maneja la pagina JSP -->
<form action=”calculoIntereses.jsp” method=”post”>
        Inversi&oacute;n: <input type=”text” name=”inversion”><br/>
        Meses: <input type=”text” name=”meses”><br/>
        Tasa Anual %: <input type=”text” name=”tasa”><br/>

        <input type=”submit” value=”Calcular”>
</form>

</body>
</html>
```

InteresesBean.javacalculoIntereses.jsp

```
package capitulo3;

public class InteresesBean {

    private double inversion;

    private int meses;

    private double tasa;

    public void setInversion(double inversion) {
        this.inversion = inversion;
    }

    public void setMeses(int meses) {
        this.meses = meses;
    }

    public void setTasa(double tasa) {
        this.tasa = tasa;
    }

    public double getInversion() {
        return inversion;
    }

    public int getMeses() {
        return meses;
    }

    public double getTasa() {
        return tasa;
    }

    // El cálculo de las ganancias
    public double getGanancias() {
        return inversion * (tasa / 100.0) * (meses / 12.0);
    }
}
```

calculoIntereses.jsp

```
<%@ page errorPage=”error.jsp”%>

<html>
<head>
<title>Calculador de intereses JSP</title>
</head>
<body>

<jsp:useBean id=”intereses” class=”capitulo3.InteresesBean”>
        <jsp:setProperty name=”intereses” property=”*” />
</jsp:useBean>

Inversi&oacute;n : <jsp:getProperty name=”
intereses” property=”inversion” /><br/>
Meses: <jsp:getProperty name=”intereses” property=”meses” /><br/>
Tasa Anual %: <jsp:getProperty name=”intereses” property=”tasa” /><br/>
<br/>
Ganancias: <jsp:getProperty name=”intereses” property=”ganancias” /><br/>

</body>
</html>
```

error.jsp

```
<%@ page isErrorPage=”true”%>
<html>
<head>
<title>Error en pagina JSP</title>
</head>
<body>
```

```
Error en la entrada de datos: <br/>
<%= exception.getMessage() %>
<br/>

<a href=”javascript:history.go(-1)”>Volver</a>

</body>
</html>
```

Problemas de usar solo JSP

Al igual que con los servlets, usar solo JSP para desarrollar un sitio complejo no es muy eficiente. Mezclar código Java junto con el contenido web puede resultar en código difícil de leer y mantener, como en el ejemplo que se ve a continuación:

```
<%
  int total = 0;
  for(int i=0; i < productos.size; i++) {
        Producto pr = (Producto) productos.elementAt(i);
        total += pr.getPrecio();
%>

<span class=”producto”>
Producto <%= pr.getNombre() %>
</span>

<%
        if (pr.getTipo() == “A”) {
%>
        El producto tiene un <b>10%</b> de descuento
<%
        }
        else {
```

```
%>
        El producto tiene un <b>15%</b> de descuento
<%
        }
}
        if (precio > 1000) {
%>
```

Este tipo de código puede ser mejorado considerablemente si movemos la lógica de negocios a un JavaBean. También puede ayudar mucho usar EL (Expression Language), un lenguaje que sirve para acceder a variables, expresar ciclos, condiciones, etcétera, sin necesidad de insertar código Java en el JSP. Sin embargo, tenemos el problema del control de flujo, y hay ciertos inconvenientes que simplemente hacen el código difícil de entender y mantener. El control de flujo invariablemente implica código dentro de un JSP, y tener el control de flujo de la aplicación disperso en las páginas agrega complejidad al sistema.

En el próximo capítulo, veremos cómo usar una combinación de servlets y JSP para resolver estos problemas, planteando un nuevo paradigma, y veremos un framework que nos va a ayudar mucho: Struts .

RESUMEN

En el capítulo vimos las JavaServer Pages , una especificación que se diseñó para resolver algunas complicaciones que presenta el desarrollo con servlets. Vimos cómo usarla para generar contenido dinámico y cómo interactuar con JavaBeans . También vimos que JSP no es la panacea y que trae sus complicaciones si lo queremos utilizar exhaustivamente.

Actividades

TEST DE AUTOEVALUACIÓN

1. ¿Cuál es la principal ventaja de usar JSP respecto de usar servlets?
2. ¿Por qué hay variables implícitas en un JSP? ¿Cómo son inicializadas?
3. ¿Qué ventajas tiene escribir un JSP como un documento XML?
4. ¿En qué casos es necesario usar un tipo de inclusión de documentos al JSP porque el otro tipo no serviría?

EJERCICIOS PRÁCTICOS

1. Escriba una página JSP que salude al usuario de diferentes formas según la hora del día (mañana, tarde o noche).
2. Escriba una página JSP que use un JavaBean para acceder y escribir propiedades. Luego vea el servlet desarrollado.
3. Convierta el AhorcadoServlet del Capítulo 2 en una página JSP, manteniendo su funcionalidad.

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: profesor@redusers.com .

*04

Struts

En este capítulo haremos una introducción a Struts, una potente herramienta de desarrollo web que nos posibilita una mesa de trabajo (framework) del cual podemos disponer para potenciar los alcances de nuestra programación.

Introducción a Struts

El desarrollo de aplicaciones web usando JSP o servlets es conocido como Model 1. Bajo este primer paradigma, accedíamos directamente a páginas o servlets, estos procesaban información y devolvían un resultado. Esta forma de desarrollo arrastra problemas tales como la mezcla de lógica con contenido, un flujo de la aplicación bastante confuso, una reutilización de código que se torna dificultosa y, en definitiva, al final nos queda una aplicación difícil de mantener tanto para los programadores como para los diseñadores.

Para remediar esto, se propuso usar servlets para el control de flujo y JSP para generar el HTML resultante. A esta conjunción de tecnologías se la conoce como Model 2. Este nuevo modelo es una adaptación del patrón de diseño Model-View-Controller.

Este nuevo paradigma propone separar el modelo, la vista y el controlador en tres componentes disjuntos y desacoplados (en lo posible), a fin de poder modificar un componente independientemente de los demás y que el resultado impacte poco o nada sobre los otros componentes. Por ejemplo, podemos rediseñar completamente la parte gráfica de nuestra aplicación sin tener que modificar en absoluto su funcionalidad subyacente: estamos modificando la vista sin necesidad de modificar el modelo o el controlador.

Antes de seguir ahondando en el paradigma, veamos con detalle cada una de sus partes.

STRUTS ACTION VS. STRUTS SHALE

Actualmente, el proyecto Struts –desarrollado por Apache Foundation– se encuentra en proceso de cambios. Por un lado, siguen mejorando el proyecto Struts original, llamado Struts Action, y, por otro lado, están desarrollando una versión alternativa llamada Struts Shale. Esta última se basa en JavaServer Faces.

El modelo

Se trata de los componentes que representan la información con la cual debe interactuar la aplicación. Usualmente, son implementados con JavaBeans que contienen todo lo necesario para soportar la funcionalidad que es requerida por el sistema. Por ejemplo, si la aplicación consiste en una agenda de contactos, el modelo contendrá las personas, sus datos, sus relaciones, etcétera.

La vista

Es la representación gráfica (y, generalmente, interactiva) de nuestra aplicación. En nuestro caso, esta representación será finalmente una página HTML, pero podríamos diseñar aplicaciones que tengan varias vistas. Por ejemplo, un sitio web accesible desde un navegador con vista HTML y desde un teléfono celular con vista WML.

El controlador

Es el encargado del control del flujo de la aplicación, responde a los eventos generados en la vista e invoca las acciones pertinentes (en el modelo) y devuelve el control al usuario (en la vista).

El funcionamiento de una aplicación (web, en nuestro caso, pero puede ser de cualquier tipo) bajo este paradigma es el siguiente:

(1) El usuario hace un pedido.
(2) El controlador recibe el pedido, lo analiza y decide qué acción debe ejecutar.
(3) La acción se ejecuta y el modelo es modificado (o no) para ser consistente con el pedido del usuario.
(4) El controlador recibe el control de vuelta del modelo y decide qué vista activar.
(5) La vista se muestra, accediendo al modelo -de ser necesario- para generarse.

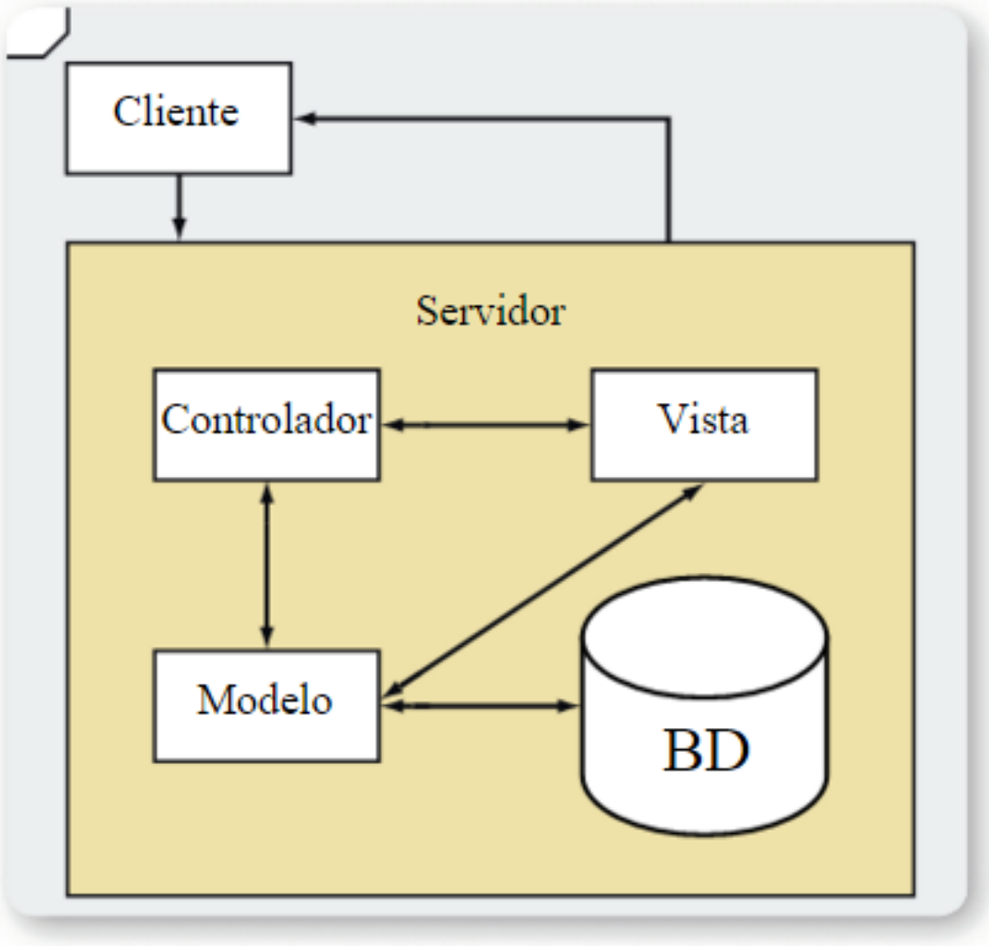


Figura 1. Los componentes son independientes pero interactúan entre sí.

Struts

Struts es un framework para el desarrollo de aplicaciones web bajo el paradigma MVC. Debemos recordar que Struts 2 no es una continuación de Struts 1; es una unificación de dos framework: WebWork y Struts. Es simplemente un servlet, está configurado para responder a una serie de pedidos. Al invocar al servlet desde una página web, este accede a su configuración que, según el pedido que se esté efectuando, redirige el control a una acción particular. Esta acción (una clase específica) se ejecuta y finalmente llama al controlador para que cree y devuelva la vista correspondiente. El controlador

MVC

Este paradigma existe desde 1979, fue ideado por Trygve Reenskaug, un noruego que desarrollaba en Smalltalk para Xerox. Actualmente es muy usado en aplicaciones web y también fue el paradigma que eligieron los ingenieros de Sun al diseñar Swing.

obtiene la vista que ha de mostrar (nuevamente mediante la configuración) y la muestra. Por lo general, al ser contenido dinámico, la vista necesita acceder al modelo para obtener los datos a mostrar. Podemos ver que esta lógica de funcionamiento se adapta al patrón Model-View-Controller .

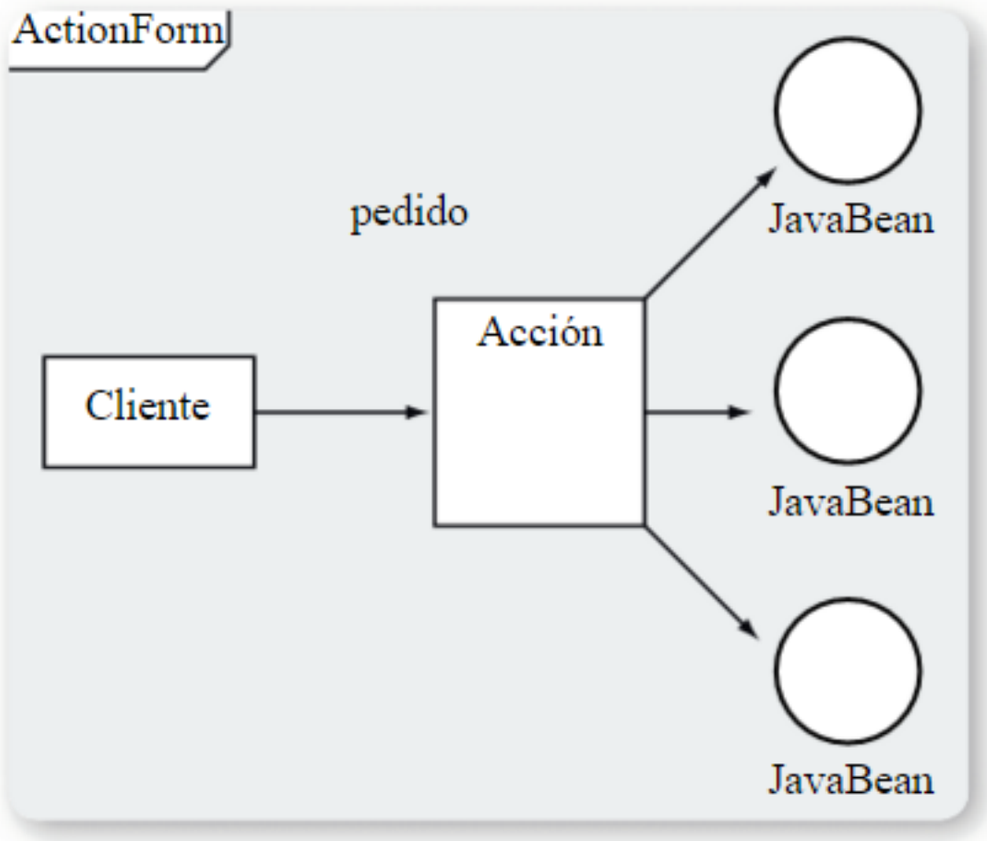


Figura 2. La representación de componentes Struts como paradigma MVC.

Hola Struts

Veamos un ejemplo de "Hola Mundo" usando Struts, para luego discutir al respecto. En http://struts.apache.org/download.cgi descargamos Struts. Hay varias distribuciones posibles de descarga:

JARS EN STRUTS

Muchos componentes que utiliza Struts para su implementación son provistos por Jakarta Commons: BeanUtils (manejo de JavaBeans), Digester (proceso de archivos XML),FileUpload (envío de archivos), Logging (registración de eventos) y Validator (validación de formularios). También incluye al proyecto ANTLR, un parser para interpretar validaciones complejas definidas por el usuario.

STRUTS ES UN FRAMEWORK PARA EL DESARROLLO DE APLICACIONES WEB BAJO MVC

- Binaries : esta distribución contiene la distribución completa de Struts en archivos compilados, con documentación y, además, aplicaciones de ejemplo.
- Sources : contiene el código fuente de todo el framework y aplicaciones de ejemplo y documentación.
- Library : contiene solamente los archivos JAR empaquetados para ser usados en aplicaciones web.

La distribución Sources es ideal para indagar y entender el funcionamiento completo de Struts por dentro. A los efectos de seguimiento del libro, usaremos la distribución Library.

Una vez realizado el Paso a paso de la siguiente página, podremos comprobar que ya tenemos todos los archivos de Struts necesarios para integrarlo a nuestra aplicación. El siguiente desafío consistirá en aprender cómo configurar la aplicación para que use Struts.

Recordemos que la configuración de una aplicación web se realizaba mediante el archivo web.xml , residente en la carpeta WEB-INF . Para ello, en la página 95, veremos un archivo web.xml básico que incluye Struts.

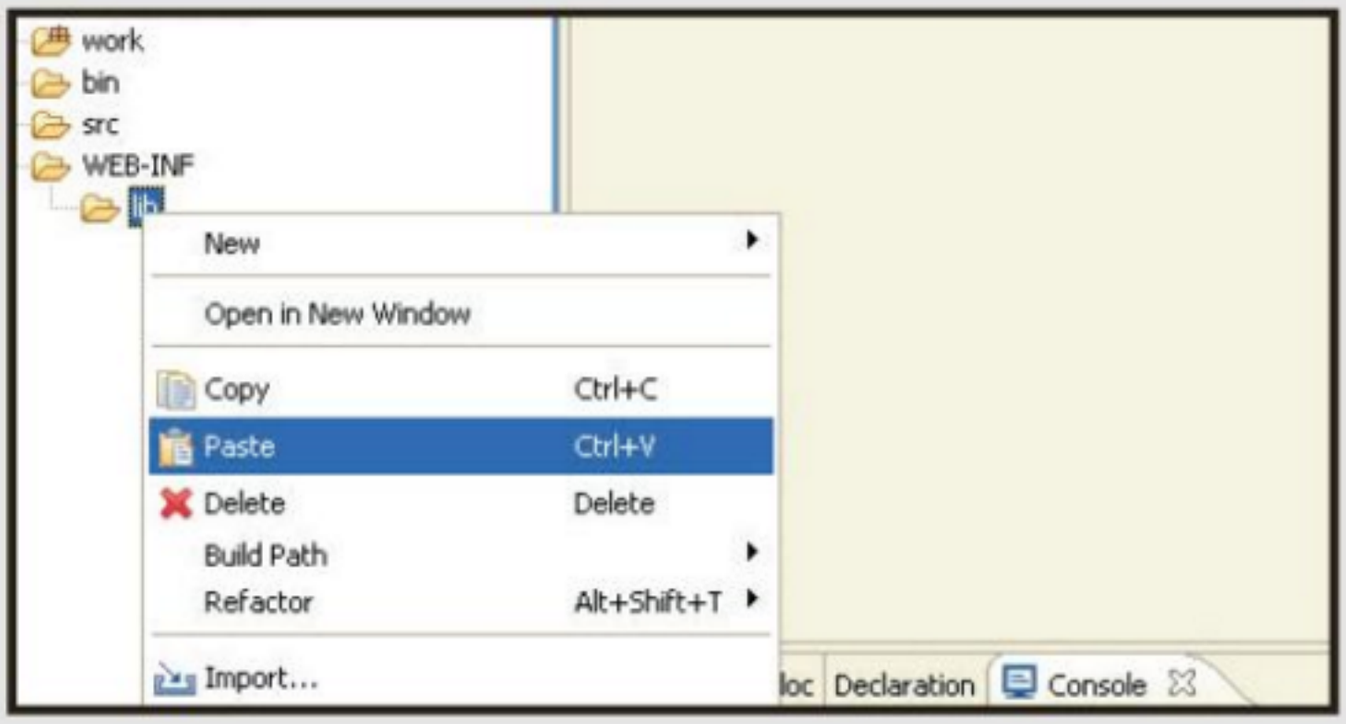
STRUTS Y LA VISTA

Struts está diseñado para ser independiente de la vista, y aunque JSP es la opción más usada y con mayor soporte de funcionalidad, también podemos usar otros frameworks que se integran fácilmente con Struts, como Velocity (similar a JSP pero basado en plantillas), Cocoon (un framework de desarrollo web) y Stxx (una extensión de Struts que implementa Model 2X).

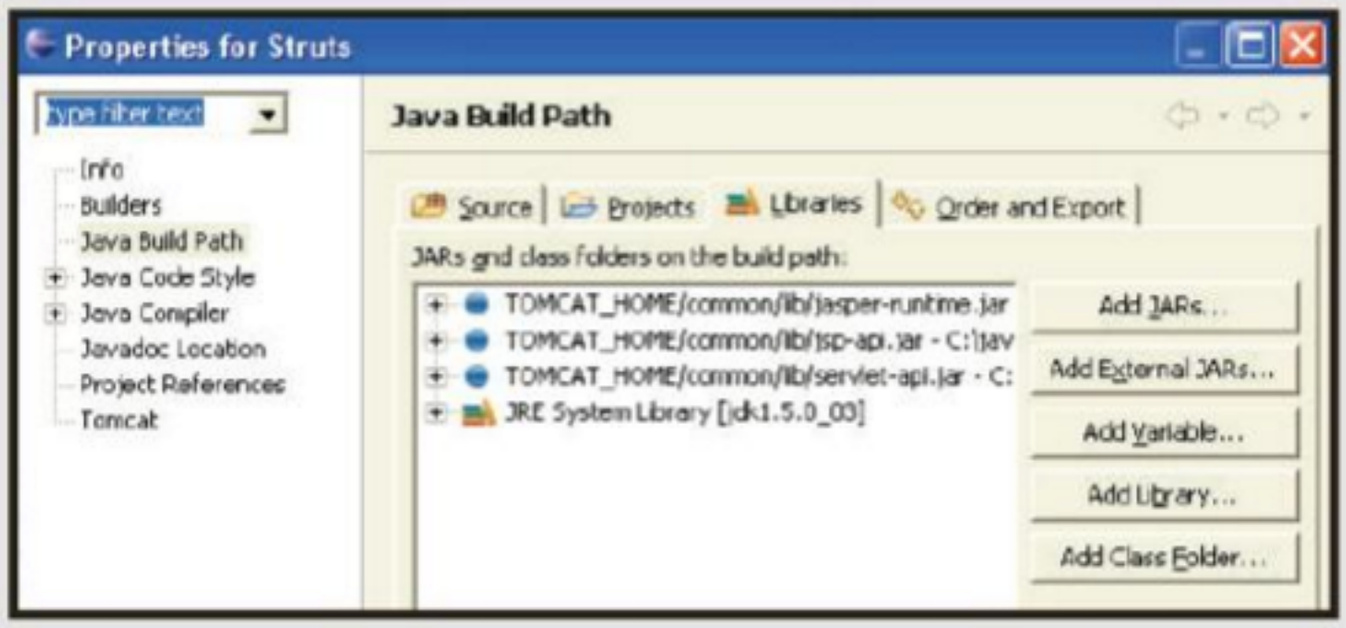
PAP: INSTALAR STRUTS EN LAS APLICACIONES WEB

01 Cree un nuevo proyecto accediendo a File/New/Project .
Elija el tipo de proyecto Tomcat Project .

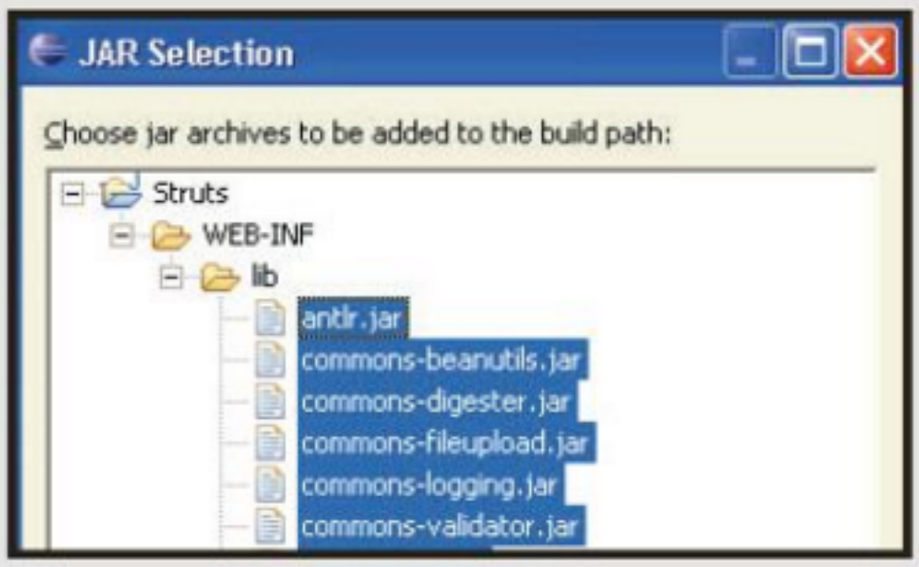
02 Copie solo los archivos de extensión . JAR de la distribución Struts en la carpeta WEB-INF/lib del proyecto que se ve en la imagen.



03 Agregue los archivos .JAR al classpath accediendo a las propiedades del proyecto (botón derecho sobre el nombre del proyecto / Properties) y en la pestaña Libraries , haciendo clic en Add JARs .



04 Seleccione todos los archivos bajo WEB-INF/lib y presione OK



05 Presione OK en el cuadro de propiedades del proyecto. Luego, copie todos los archivos de extensión . TLD de la distribución Struts en la carpeta WEB-INF del proyecto.

```
<?xml version=”1.0” encoding=”ISO-8859-1”?>

<!DOCTYPE web-app
  PUBLIC “-//Sun Microsystems, Inc.//DTDWeb Application 2.2//EN”
  “http://java.sun.com/j2ee/dtds/web-app_2_2.dtd”>

<web-app>
  <display-name>Aplicación con Struts</display-name>

 <!-- Definimos el servlet de Struts -->
 <servlet>
   <servlet-name>action</servlet-name>
   <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>

       <!-- Indicamos a Struts dónde buscar el archivo de configuración -->
   <init-param>
     <param-name>config</param-name>
     <param-value>/WEB-INF/struts-config.xml</param-value>
   </init-param>

       <!-- Esta directiva indica que el servlet ha
de cargarse al inicializarse Tomcat -->
   <load-on-startup>2</load-on-startup>
</servlet>


<!-- Asociamos el servlet de Struts a todos los pedidos que terminen con .do -->
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>

<welcome-file-list>
```

```
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>


<!-- Descriptores de la librería de tags de Struts -->
<taglib>
  <taglib-uri>/tags/struts-bean</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>


<taglib>
  <taglib-uri>/tags/struts-html</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/tags/struts-logic</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/tags/struts-nested</taglib-uri>
  <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/tags/struts-tiles</taglib-uri>
  <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
</taglib>

</web-app>
```

Se define un servlet que es el controlador de Struts y se asocia todo pedido que termine con .do a ese servlet. Tradicionalmente, las aplicaciones Struts asocian esta extensión, pero podemos definir cualquier tipo que queramos. Notemos también que dicho servlet toma un parámetro. Este parámetro es la ubicación del archivo de configuración de Struts, que contiene toda la información necesaria para el funcionamiento de la aplicación. Más adelante volveremos con detalles sobre este archivo.

Finalmente, están las declaraciones de taglibs. Struts provee una serie de tags muy útiles para usar en las páginas JSP (la vista). Para poder usar dichos tags, los declaramos en el descriptor web.

Veamos un ejemplo minimalista del archivo de configuración de Struts (struts-config.xml) para ejecutar "Hola Mundo" como aplicación web con Struts.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
        "http://jakarta.apache.org/struts/dtds/struts-config_1    _2.dtd">

<struts-config>

    <action-mappings>
<!-- Acción "Hola Mundo" -->
    <action
path="/holamundo"
type="capitulo4.action.HolaMundoAction">

<!-- Una posible vista de la acción -->
          <forward name="ok" path="/jsp/holamundo.jsp" />
      </action>
    </action-mappings>

</struts-config>
```

El archivo de configuración de una aplicación completa es inmensamente más grande y complejo que este. En esta reducida y sencilla versión simplemente declaramos que nuestra aplicación tiene una única acción posible. Esta se activará cuando se acceda a la dirección holamundo y la acción a ejecutarse es la que define la clase capitulo4.HolaMundoAction . A su vez, esta acción tiene una sola vista asociada, de nombre ok y asociada a la página JSP ubicada en /jsp/holamundo.jsp .

Unos detalles más a destacar (sin entrar aún en detalles sobre este archivo): el path o punto de acceso de la aplicación determina la dirección que queremos asociar a la acción. Notemos que no estamos agregando la extensión .do al path; sin embargo, en el descriptor de la aplicación definimos que asociamos todas las direcciones que terminan con .do al servlet de Struts. Esto implica que si queremos acceder a la acción definida, asumiendo que estamos ejecutando Tomcat en la máquina local y que el proyecto se llama Struts, debemos acceder a la siguiente dirección web: http://localhost:8080/Struts/holamundo.do .

Veamos la implementación de la clase HolaMundoAction :

```
package capitulo4.action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class HolaMundoAction extends Action {

        // Debemos sobreescribir el método execute
    public ActionForward execute(ActionMapping map, ActionForm form,
          HttpServletRequest req, HttpServletResponse res) throws Exception {

       // Queremos ir a la vista de nombre "ok"
       return map.findForward("ok");
    }
}
```

Una acción que defina una clase para ser ejecutada debe referirse a una clase que extienda org.apache.struts.action.Action . Típicamente sobrescribiremos el método execute , que es el método llamado por Struts. En el ejemplo, simplemente le informamos al controlador que devuelva la vista de nombre ok.

A no preocuparse si esto no se entiende totalmente; simplemente estamos mostrando un pantallazo del funcionamiento de Struts, para luego poder proseguir con más detalle y con ejemplos más completos.

Para terminar esta mini aplicación, nos falta la vista. Según el archivo de configuración de Struts, esta se encuentra ubicada en /jsp/holamundo.jsp .

Creamos la carpeta y dentro de ella creamos la página JSP:

```
<%@ taglib uri="/tags/struts-bean" prefix="bean"%>

<jsp:useBean id="hm" class="capitulo4.model.HolaMundoBean" />

<b>
<bean:write name="hm" property="saludo" />
</b>
```

Esta página JSP declara que usaremos un conjunto de tags de Struts, instancia un JavaBean (esto es exactamente igual que en el Capítulo 3) y finalmente utiliza un tag específico de Struts para imprimir el saludo

JAR Y CLASSPATH

En el paso 3 del Paso a paso , agregamos los .JAR de los proyectos que utiliza Struts a nuestro classpath. Esto es necesario para compilar nuevas clases que tengan referencias a clases de estos proyectos. Para la ejecución, Tomcat automáticamente agrega a su classpath los archivos que estén dentro de las carpetas WEB-INF/lib de cada proyecto.

en pantalla. En este ejemplo, el tag de Struts no aporta mayor funcionalidad a la que podríamos haber logrado usando <jsp:getProperty .../> , pero luego veremos que estos tags provistos por Struts son muy poderosos y útiles.

Vemos que hemos instanciado un bean. El código de dicho bean es:

```
package capitulo4.model;

public class HolaMundoBean {
    // Una propiedad
    private String saludo = "Hola mundo con Struts";

    // La propiedad es de lectura
    public String getSaludo() {
        return saludo;
    }
}
```

Sin más, un bean con una propiedad de tipo string y un método accesor.

Si arrancamos Tomcat y apuntamos el navegador a la dirección http://localhost:8080/Struts/holamundo.do veremos lo siguiente:

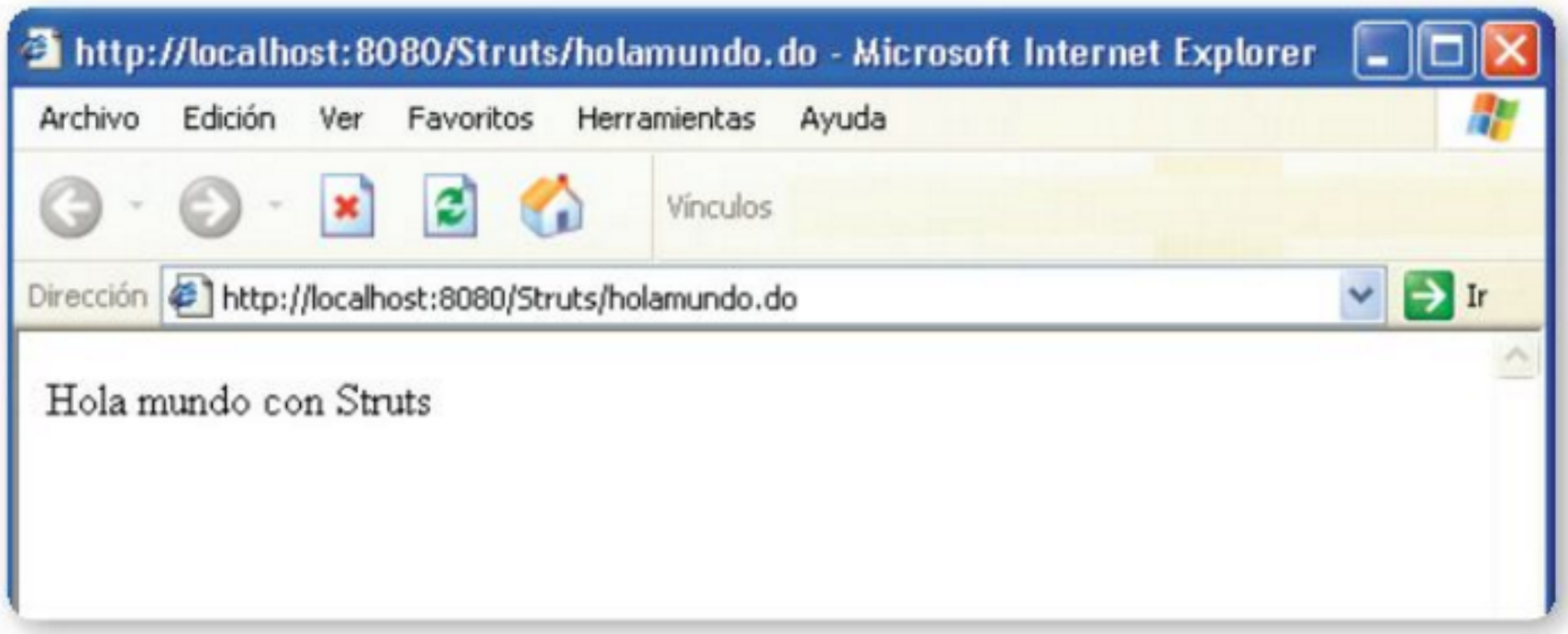


Figura 3. El resultado producido es el de siempre, la diferencia es que lo hemos producido mediante Struts.

Esto quiere decir que, después de tanto trabajo, ¡obtenemos exactamente lo mismo que hubiésemos logrado con una sola línea de JSP!

Struts no es muy eficiente para escribir aplicaciones del estilo Hola Mundo , pero pronto veremos que, para aplicaciones que requieran un mínimo de complejidad, aporta mucho y simplifica el desarrollo.

Ahora que hemos visto un ejemplo concreto en Struts, vamos a identificar los componentes del modelo MVC.

- El modelo en este caso consiste simplemente en un JavaBean (HolaMundoBean) que contiene los datos a utilizar por la aplicación.
- La vista es una página JSP que no tiene funcionalidad propia, simplemente muestra los datos provistos por el modelo.
- El controlador es Struts, que provee el servlet, las clases necesarias para obtener la configuración, la lógica interna, etcétera, y, además, la clase HolaMundoAction , encargada de definir las acciones y el flujo de nuestra aplicación.

Utilidad de Struts

Struts nos va a ayudar mucho a desarrollar aplicaciones web.
Algunas de las utilidades que provee son:

- Desarrollo en componentes según el paradigma MVC, separando la lógica del contenido.

MÚLTIPLES VISTAS

El hecho de que una acción tenga varias vistas asociadas se refiere a que las acciones, por lo general, dependen de ciertos parámetros de entrada, y puede ocurrir que prefiramos mostrar diferentes resultados que dependen de estos parámetros.

- Utilidades para mapear JavaBeans con formularios (mediante ActionForms) y cargar los formularios con el contenido del bean.
- Internacionalización de la presentación En base al lenguaje del navegador, Struts automáticamente puede mostrar el contenido apropiado.
- Validación . Para validar formularios enviados por el usuario, Struts provee el Validator Framework, muy útil para validadores simples (y no tan simples) que nos permiten efectuar validaciones sin necesidad de escribir código. También podemos crear nuestros propios validadores o definir la validación específica de los formularios, si lo necesitamos.
- Subclases de Action . No tenemos que limitarnos a extender la clase Action para cada acción que deseemos. Struts provee varias subclases que proveen ciertas funcionalidades.
- Manejo de excepciones . Podemos definir manejadores de excepciones a nivel global o por acción; de esta forma, cuando ocurra una excepción, podremos realizar acciones de contingencia, registrar lo ocurrido para depuración, etcétera.

Soporte para DataSources . En el archivo de configuración podemos declarar uno o más elementos data-source para definir accesos a bases de datos y utilizarlos desde acciones o servlets.

ACTION ES LA CLASE MADRE DE LAS ACCIONES QUE SE REALIZAN EN LA APLICACIÓN

STRUTS EN EVOLUCIÓN

Al ser Struts un proyecto de código abierto (open source), está en constante evolución. Actualmente, cientos de colaboradores de todas partes del mundo aportan código e ideas para futuras versiones del proyecto.

- Plugins . Podemos definir plugins que se enchufan en la aplicación para realizar diversas tareas en la inicialización o terminación de esta.
- Upload de archivos . Struts provee una API sencilla para manejar casos en los que el usuario envía archivos al servidor como parte de un formulario.
- Reusabilidad de componentes de vista usando Struts Tiles.

Action y ActionForm

En los próximos capítulos analizaremos en detalle estos componentes, pero es menester aquí una pequeña explicación para entender el funcionamiento de Struts.

Como vimos, Action es la clase madre de las acciones que se realizan en la aplicación. Los objetos Action son un adaptador entre un pedido HTTP y la lógica que ha de efectuarse en respuesta al pedido. Al recibir un pedido HTTP, Struts lo recibe y lo envía al Action correspondiente, proporcionando además objetos propios de Struts para su ejecución.

La aridad es la signatura del método: qué parámetros toma y en qué orden. Recordemos la aridad del método execute :

```
ActionForward execute(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response)
```

Una clase a destacar en esta discusión es ActionForm .

Un ActionForm es un JavaBean cuyas propiedades son relacionadas con parámetros del pedido. Al enviarse un formulario, se carga ActionForm con los valores establecidos por el usuario y se envía a Action para su ejecución.

Desarrollando aplicaciones con Struts

Sabiendo cómo desarrollar aplicaciones con servlets y con JSP, veamos ahora en qué se diferencia el desarrollo al utilizar Struts.

Siguiendo con la temática del paradigma MVC, el desarrollo de aplicaciones web se reduce a escribir estos componentes:

- Model: objetos, por lo general JavaBeans, aunque pueden ser objetos de cualquier tipo que contienen la información y la lógica de lo que queremos representar. Si bien los ActionForms por lo general contienen propiedades compartidas con objetos del modelo, son componentes del controlador. Los objetos del modelo idealmente deben estar completamente desacoplados del entorno web. Esto incrementa las chances de poder reutilizarlos en otros entornos. En concreto, no deberíamos depender de ninguna clase específica de Struts o servlets.

- View: la vista consiste en generar páginas HTML o crear componentes que las generen, como páginas JSP. Es importante recordar que Struts no se limita a usar JSP como capa de presentación. ¡Ni siquiera tenemos que generar HTML como resultado! Podríamos generar y devolver al usuario cualquier tipo de documento, si así lo quisiéramos.

- Controller : escribir estos componentes consiste en desarrollar objetos Action , ActionForm y asociarlos entre sí en el archivo de configuración. Los objetos ActionForm no se limitan a replicar las propiedades del formulario al cual están asociados, sino que también desarrollan su validación. Los objetos Action son los encargados de procesar los pedidos e indicarle al controlador la vista a mostrar como resultado.

En los próximos capítulos conoceremos cada uno de estos componentes, sus posibles variantes y su interacción.

Model 2X

Al poco tiempo de popularizarse Struts en el mercado, algunas personas notaron que el paradigma MVC implementado por Struts usando JSP como capa de presentación no termina de aislar completamente el

contenido de la presentación. Si bien las buenas prácticas dicen que debemos usar JSP únicamente como capa de presentación y acceder al modelo para el contenido, nada impide que escribamos código en los documentos JSP. Si esto ocurre, los beneficios del paradigma MVC se pierden, y la funcionalidad queda esparcida entre el modelo y la vista. Si encima ejecutamos lógica en las acciones, ¡tendremos funcionalidad en el modelo, la vista y el controlador!

Una alternativa planteada, llamada Model 2X, consiste en hacer que las clases Action generen documentos XML y luego estos sean procesados mediante transformadores como XSLT o XLS:FO. Este acercamiento separa completamente el contenido de la presentación y tiene numerosas ventajas.

RESUMEN

En el presente capítulo vimos una introducción a Struts, su funcionamiento según el paradigma Model–View–Controller, sus componentes, y cómo utilizarlo para el desarrollo de aplicaciones web. Vimos cómo emplearlo en un proyecto web nuevo, creamos una pequeña aplicación de ejemplo y dimos un pantallazo por el Model 2X.

Actividades

TEST DE AUTOEVALUACIÓN

1. ¿En qué consiste el paradigma MVC?
2. ¿Qué función tiene cada componente del paradigma MVC?
3. ¿Cómo implementa Struts el paradigma Model-View-Controller?
4. ¿Cuál es la utilidad de los ActionForm ?
5. Apegándonos al paradigma MVC, ¿conviene ejecutar código de la aplicación en las clases Action ?

EJERCICIOS PRÁCTICOS

1. Modifique la aplicación para que Struts maneje los pedidos que terminen con .struts en vez de .do .
2. Modifique el modelo en la aplicación para que devuelva la fecha actual además del saludo.
3. Modifique la vista en la aplicación para que devuelva la fecha actual además del saludo.

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: profesor@redusers.com .

*05

ActionForms

En este capítulo conoceremos los ActionForms. Veremos con detalle qué son, cómo programarlos y cuál es su importancia en el desarrollo de aplicaciones. Luego del enfoque tradicional, contemplaremos otras alternativas.

Interactuando con el usuario

Por lo general, desarrollamos aplicaciones web que interactúan con usuarios. Aunque es válido crear aplicaciones que simplemente muestren contenido (dinámico) que vaya cambiando con el correr del tiempo, solemos buscar que el usuario interactúe con la aplicación, de forma tal que podamos brindarle la información específica que necesita.

En la programación web con páginas HTML, el usuario tiene diversas formas de interactuar con una aplicación. Por ejemplo, cuando necesitamos que el usuario ingrese datos, usamos un formulario.

Los formularios en HTML no son más que una aplicación gráfica que existe solamente en el cliente (el navegador web) para ayudar a generar el pedido HTTP. En teoría, podríamos no usar formularios y dejar que el usuario cree el pedido él mismo, pero sería una aplicación muy poco amigable.

Al ser los formularios la herramienta fundamental para recibir información del usuario, las aplicaciones web trabajan con ellos intensamente. Struts provee funcionalidad para ayudarnos a simplificar la tarea de escribir formularios y, sobre todo, para mantener su estado entre pedidos.

¿A qué nos referimos con mantener el estado de un formulario ?

Es muy común en los sitios web tener un formulario de registración donde se pide una gran cantidad de datos, como nombre, clave, confirmación de clave, dirección, teléfono, correo electrónico, etcétera. Supongamos (como suele suceder) que tenemos ciertas restricciones sobre ese formulario: los campos clave y confirmación de clave deben contener el mismo valor; el nombre de usuario no debe existir ya en nuestra base de datos. Imaginemos que el usuario ingresa algunos datos y envía el formulario pulsando el botón correspondiente. Estos datos llegan al servidor y son recibidos. El servidor, al procesarlos, encuentra, por ejemplo, que el

> EL FORMULARIO ES UNA HERRAMIENTA FUNDAMENTAL QUE RECIBE INFORMACIÓN DEL USUARIO

campo de correo electrónico, que es un campo requerido, no contiene datos. Entonces agrega el mensaje de error correspondiente y devuelve el mismo formulario al usuario para que corrija el error.

Siguiendo estos pasos aquí descritos, todo parece estar correctamente; sin embargo, el usuario recibirá el mensaje de error correspondiente, ¡y el formulario todo en blanco! Deberá entonces volver a completar todos los datos además de realizar la corrección, algo sumamente largo y tedioso.

Aunque aún hoy se ven en internet aplicaciones que funcionan de esa manera, debemos asegurarnos de devolver el formulario cargado con los datos que el usuario ya había ingresado, y esto es a lo que nos referimos con mantenimiento del estado del formulario. Struts, entre otras cosas, nos provee lo necesario para no tener que implementar nosotros la funcionalidad de completar los campos que el usuario ya había ingresado al devolverle un formulario.

ActionForms

Un ActionForm representa un formulario en una página web. Las acciones en Struts (implementadas en la clase Action) rara vez interactúan directamente con HttpServletRequest o HttpServletResponse .

Implementar un ActionForm es sencillo: simplemente debemos crear un JavaBean que extienda la clase org.apache.struts.action.ActionForm y definir sus propiedades y métodos de acceso como cualquier otro JavaBean.

ACTIONFORM COMO FIREWALL

Una analogía útil es pensar que el ActionForm es un firewall entre el pedido HTTP y el Action que se ejecuta. De esta forma, solo enviamos al Action propiedades que declaramos en el ActionForm y, a su vez, podemos validar los datos que son pasados, de forma de poder programar el Action asumiendo que los datos son los necesarios y son válidos.

Una vez definido el ActionForm, debemos declararlo en el struts-config y asociarlo con acciones. Al asociar un ActionForm con acciones, le estamos diciendo a Struts que, cuando llegue un pedido para una acción asociada, intente completar los datos del ActionForm con los datos del pedido, y ese ActionForm es el que recibirá el método execute de la clase Action .

Desarrollemos un ejemplo de aplicación web con un formulario donde el usuario debe ingresar nombre, sexo, edad y si es fumador o no. La página resultante se verá como en la Figura 1 .

Figura 1. En entornos de producción deberíamos restringir el acceso directo a las páginas JSP; el usuario debería interactuar solamente con acciones de Struts.

Luego, nos dedicaremos a la parte de creación de la vista, que no es como en un JSP tradicional, sino que se crea utilizando tags de Struts; mientras tanto, veamos cómo sería nuestro ActionForm.

```
import javax.servlet.http.HttpServletRequest;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

public class DatosForm extends ActionForm {
    private String nombre;
```

```
    private Integer edad;

    private boolean fumador;

    public Integer getEdad() {
        return edad;
    }

    public void setEdad(Integer edad) {
        this.edad = edad;
    }

    public boolean isFumador() {
        return fumador;
    }

    public void setFumador(boolean fumador) {
        this.fumador = fumador;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

Nuestro ActionForm no es más que un JavaBean que extiende la clase ActionForm de Struts.

Vemos que usamos un Integer para representar la edad y un boolean para indicar su status de fumador. ¿Por qué Integer y no el primitivo int ?

Recordemos que toda la comunicación cliente-servidor se realiza mediante el protocolo HTTP, que en definitiva transmite cadenas de caracteres.

> ACTIONFORM ES UN FORMULARIO EN UNA PÁGINA WEB Y ES SENCILLO DE IMPLEMENTAR

Struts recibe los datos e intenta hacer la conversión de tipos correspondiente entre los valores del pedido y las propiedades del ActionForm. Si en vez de `Integer`, hubiésemos puesto `Vector`, por ejemplo, al enviar el formulario desde el navegador obtendríamos un error de tipo.

Luego, le brindamos al usuario un campo de texto donde pueda ingresar la edad y esperamos un número que represente una edad. Pero el usuario puede ingresar cualquier texto. Si el usuario ingresa una cadena que no puede ser transformada a un número, Struts nos devuelve el valor por defecto del objeto (en este caso, un número con valor cero).

Esto funciona tanto para el primitivo `int` como para el objeto `Integer`, pero no podremos discernir si el usuario no ingresó valor o ingresó un valor incorrecto, o si realmente ingresó el número 0.

Si usamos como tipo de datos el primitivo `int`, no tendremos forma de saberlo. En cambio, usando `Integer` podemos indicarle a Struts que, cuando no pueda realizar las conversiones de tipo, establezca `null` como valor de los objetos en vez de su valor por defecto. De esta forma, si recibimos `null` como valor, podremos aseverar que hubo un error de conversión de tipo y que el usuario no ingresó un número válido.

Esta configuración se realiza asignando el parámetro de inicialización `convertNull` en el archivo `web.xml`. En nuestra aplicación, aplicando los cambios, el fragmento que inicializa Struts quedaría de la siguiente forma:

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class> org.apache.struts.action.ActionServlet
</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml </param-value>
```

```
    </init-param>
    <init-param>
      <param-name>convertNull</param-name>
      <param-value>true</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>
```

Configurar un ActionForm

Una vez definido un ActionForm, debemos asociarlo con determinadas acciones para que sea instanciado y cargado por Struts cuando dichas acciones sean accedidas. Los forms dentro del struts-config se definen como una serie de tags form-bean encerrados dentro de un tag form-beans . Por ejemplo:

```
<form-beans>
        <form-bean ...>
<form-bean ...>
<form-bean ...>
</form-beans>
```

STRING O NO STRING, ESA ES LA CUESTIÓN

Algunos desarrolladores optan por definir todas las propiedades de los ActionForm como objetos string . Así, tienen más control sobre lo que reciben, ya que no sufre ninguna transformación de tipo y pueden realizar validaciones propias. Otros prefieren usar tipos más concretos (integer , boolean) cuando corresponde. Ambos enfoques son válidos y tienen sus pros y sus contras.

Cada elemento form-bean puede poseer tres atributos:

- className define la clase que ha de configurar al bean que estamos definiendo (subclase de org.apache.struts.config.FormBeanConfig). Puede ser útil usar esta configuración cuando tenemos varios beans iguales.
  En vez de configurarlos uno por uno en el struts-config , creamos una clase FormBeanConfig y se la pasamos a cada bean para que se configure, aunque por lo general es más práctico configurarlos en el struts-config .
- name es el nombre que queremos darle a este bean.
- type es el nombre de la clase que instanciará el bean.

En nuestro ejemplo, la configuración para definir el bean en Struts sería:

```
<form-beans>
        <form-bean
                name="datosForm"
                type="capitulo5.DatosForm"/>
</form-beans>
```

Es un solo bean, de nombre datosForm , que utiliza la clase que definimos previamente. Declaremos ahora una acción que use el ActionForm:

```
<action
        path="/ingresoDatos"
        name="datosForm"
        scope="request"
        type="capitulo5.IngresoDatosAction">

        <forward name="ok" path="/jsp/capitulo5/datos.jsp" />
</action>
```

Notamos dos atributos nuevos, relacionados con form-beans :

- name define el nombre del form-bean que esta acción va a usar. Este nombre debe existir entre los form-bean declarados u obtendremos un error.
- scope indica el contexto donde buscar el form-bean . Este puede residir en request o en session . Si el form-bean es guardado en sesión, cada vez que accedamos a una página que lo use estaremos accediendo a una instancia, posiblemente, con valores ya guardados de antes. Esto puede ser de utilidad si queremos dividir una carga de información en varias páginas.

ActionForm reset

Un método importante que define la clase ActionForm es:

```
public void reset(ActionMapping map, HttpServletRequest req)
```

Este método es invocado por Struts antes de completar las propiedades del form. A priori, ¡este método no tiene sentido! Dado un form con propiedades, se llama a un método que reinicializa sus propiedades, volviéndolas a su valor por defecto, para luego establecerle los valores que vienen del cliente. Entonces, ¿para qué volverlas a un valor por defecto si en definitiva se van a reescribir? En la práctica, hay un caso en el que este método es necesario.

Si tenemos un form guardado en sesión (esto es, declarado en la acción que lo utiliza con scope="session") y este formulario tiene algún checkbox (casilla de selección binaria, como, en el ejemplo, la propiedad fumador), es necesario reestablecer el valor de las propiedades asociadas con los checkboxes. Esto se debe a que un checkbox, que tiene dos

> ACTIONFORM RESET REINICIA SUS PROPIEDADES PARA LUEGO RESTABLECERLAS

estados, solo envía su estado cuando está marcado. Cuando se encuentra desmarcado, no envía nada al servidor.

Bajo este funcionamiento, si el form es reutilizado y el valor de una propiedad booleana asociada con un checkbox era verdadero, nunca va a cambiar a falso aunque la opción esté desmarcada en el navegador, simplemente porque no se recibirá información alguna sobre esa propiedad y mantendrá su estado.

Validación de ActionForms

Por lo general, al interactuar con el usuario, necesitamos validar lo que este ingresa en los formularios. Algunos datos son requeridos y el usuario no puede no completarlos, otros tienen un rango que no puede excederse, otros deben pertenecer a cierto dominio, etcétera.

La clase ActionForm posee un método apropiado para validar los datos ingresados por el usuario, y su aridad es:

```
public ActionErrors validate(ActionMapping map,
                             HttpServletRequest req)
```

Este método puede ser invocado cada vez que se realiza un pedido. Nosotros debemos proveer una implementación particular para cada uno de los ActionForms que creemos.

ActionErrors es una clase que encapsula mensajes de error. Cada mensaje de error (una instancia de la clase ActionMessage) puede ser global o estar a asociado con un campo en particular. En nuestro ejemplo, si el usuario no escribe un nombre, el mensaje indicador estará asociado con el campo nombre .

Volvamos al ejemplo previo y asumamos que los campos nombre y edad son requeridos, y que fumador puede ser verdadero solo si la edad es mayor que 18. Entonces debemos implementar el método validate de la siguiente forma:

```
public ActionErrors validate(ActionMapping map, HttpServletRequest req) {
    ActionErrors ret = new ActionErrors();

    if (nombre == null || nombre.trim().equals("")) {
        ret.add("nombre", new ActionMessage("Falta ingresar nombre",false));
    }
    if (edad == null) {
        ret.add("edad", new ActionMessage("Falta ingresar edad", false));
    }
    else {
        if (fumador && edad.intValue() < 18) {
            ret.add("fumador", new ActionMessage(
                    "Peque&ntilde;o fumador!", false));
        }
    }

    return ret;
}
```

Si no hay mensajes de error, podemos devolver un objeto ActionErrors vacío (como en el ejemplo) o bien null. Es conveniente devolver todos los errores juntos y no frenarnos al encontrar el primer error; si no, el usuario tendría que hacer las correcciones una por una.

Es muy simple agregar un mensaje de error a un ActionErrors , invocando el método:

```
add(String property, ActionMessage message)
```

El primer parámetro es el nombre de la propiedad a la cual está asociado el mensaje de error. Si queremos un mensaje de error global, usamos la constante ActionErrors.GLOBAL_MESSAGE .

Los objetos ActionMessage fueron diseñados para trabajar con internacionalización y externalización de cadenas de caracteres, a fin de no tener que embeber texto en el código.

Configurar la validación

Dijimos que el método validate puede ser invocado con cada pedido. La invocación o no del método es configurable para cada acción. Podemos querer que algunas acciones validen el contenido y otras que no, siempre refiriéndonos al mismo ActionForm. Para configurar las acciones con la validación, debemos establecer los valores de dos atributos en cada elemento action del struts-config .

- validate define si esta acción efectúa la validación del form (valores true / false).
- input define la página adonde redirigir al usuario si la validación falla.

Configurar Struts con MessageResources

Al respecto de textos externalizados, Struts necesita que definamos los archivos en donde hallar estos textos (en varios idiomas). En los próximos capítulos, ahondaremos sobre esos temas, pero es probable que Struts no funcione correctamente si falta esta definición.

ACTIONERROR VS. ACTIONMESSAGE

Suena extraño que, existiendo la clase ActionError , los mensajes de error se creen usando la clase ActionMessage . La clase ActionError es casi exactamente igual a ActionMessage . En las últimas versiones, su código fue unificado y, en las próximas, ya no se incluirá más ActionError como clase.

Para solucionar el problema, basta con agregar la siguiente línea en struts-config.xml , luego de los action-mappings :

```
<message-resources parameter="MessageResources" />
```

Esta línea indica que Struts debe buscar los mensajes en un archivo de nombre MessageResources.properties . Struts busca el archivo en las ubicaciones del classpath, por lo que una buena recomendación es ubicar este archivo en la carpeta WEB-INF/src . Por ahora este archivo puede estar en blanco, no lo usaremos.

El resto del ejemplo

Habiéndonos focalizado en el desarrollo del ActionForm, ahora veamos el código completo del ejemplo. Vale recordar que la estructura de carpetas del proyecto dependerá de cómo esté configurado por defecto.

Creamos el archivo de configuración struts-config.xml :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
        "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>

        <form-beans>
                <!-- Definimos un form-bean de nombre
datosForm, implementado por la clase capitulo5.DatosForm -->
                <form-bean
                name="datosForm"
                        type="capitulo5.DatosForm"/>
```

```
        </form-beans>

    <action-mappings>
                    <!-- Definimos una acción que usa el form datosForm,
que requiere validación y cuya página de salida,
si la validación falla, es formulario.jsp -->
            <action
                                path=”/ingresoDatos”
                                name=”datosForm”
                                scope=”request”
                                validate=”true”
                                input=”/jsp/capitulo5/formulario.jsp”
                                type=”capitulo5.IngresoDatosAction”>

                                <forward name=”ok” path=”/jsp/capitulo5/datos.jsp” />
                    </action>
        </action-mappings>

        <!-- Configuramos el archivo con mensajes externos -->
        <message-resources parameter=”MessageResources” />

</struts-config>
```

LIBRERÍAS

Para ejecutar las librerías de Struts, debemos agregarlas al proyecto; en este caso, no nos olvidemos de tener la correspondiente. Si nos falta alguna de ellas, podemos visitar la siguiente página web: www.java2s.com Particularmente podemos encontrar la que necesitamos para este proyecto en este sitio web: www.java2s.com/Code/Jar/s/Downloadstrutstaglib139jar.htm .

Creamos el archivo de formulario, que es el punto de entrada: jsp/capitulo5/formulario.jsp .

```
<%@ taglib uri=”/tags/struts-html” prefix=”html”%>

<%-- Este formulario se enviará a la acción ingresoDatos --%>
<html:form action=”ingresoDatos”>

Nombre: <html:text property=”nombre” />

<%-- El mensaje de error si algo falla respecto al nombre --%>
<b><html:errors property=”nombre” /></b>
<br/>

Edad: <html:text property=”edad” />

<%-- El mensaje de error si algo falla respecto a la edad --%>
<b><html:errors property=”edad” /></b>
<br/>

Fumador: <html:checkbox property=”fumador” />

<%-- El mensaje de error si algo falla respecto
a la condición de fumador --%>
<b><html:errors property=”fumador” /></b>
<br/>

<br/>
<html:submit />

</html:form>
```

Creamos la clase del ActionForm correspondiente al formulario:
WEB-INF/src/capitulo5/DatosForm.java .

```
package capitulo5;

import javax.servlet.http.HttpServletRequest;

import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;

public class DatosForm extends ActionForm {

    // Las propiedades
    private String nombre;

    private Integer edad;

    private boolean fumador;

    // Los métodos accesores
    public Integer getEdad() {
        return edad;
    }

    public void setEdad(Integer edad) {
        this.edad = edad;
    }

    public boolean isFumador() {
        return fumador;
    }

    public void setFumador(boolean fumador) {
        this.fumador = fumador;
    }
```

```
public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

/* Sobreescribimos el método reset para reestablecer el valor
 de la propiedad fumador */
public void reset(ActionMapping map, HttpServletRequest req) {
    fumador = false;
}

/* Validamos el contenido de este formulario y agregamos
mensajes de error en caso de fallas */
public ActionErrors validate(ActionMapping map, HttpServletRequest req) {
    ActionErrors ret = new ActionErrors();

    if (nombre == null || nombre.trim().equals("")) {
       ret.add("nombre", new ActionMessage("Falta ingresar nombre",
                    false));
    }
    if (edad == null) {
       ret.add("edad", new ActionMessage("Falta ingresar edad", false));
    }
    else {
       if (fumador && edad.intValue() < 18) {
          ret.add("fumador", new ActionMessage("Peque&ntilde;o fumador!",
                false));
       }
    }

    return ret;
  }
}
```

Creamos el archivo de la acción que se ejecuta: WEB-INF/src/capitulo5/ IngresoDatosAction.java .

```
package capitulo5;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class IngresoDatosAction extends Action {


    public ActionForward execute(ActionMapping map, ActionForm form,
            HttpServletRequest req, HttpServletResponse res) throws Exception {

        /* No hacemos nada aquí, simplemente redirigimos a la página de salida */
        return map.findForward("ok");
    }
}
```

UBICACIÓN DEL ARCHIVO MESSAGERESOURCES

Un error muy frecuente es ubicar el archivo MessageResources junto con las clases, en la carpeta denominada WEB-INF/classes . Esta carpeta está en el classpath, pero muy a menudo es borrada o regenerada, y junto con ella puede desaparecer también nuestro archivo. Por lo tanto, debemos evitar esta ubicación.

Finalmente, creamos la página de salida: jsp/capitulo5/datos.jsp .

```
<%@ taglib uri=”/tags/struts-bean” prefix=”bean”%>

<%-- Simplemente mostramos los datos ingresados por
 el usuario (notar que los estamos tomando del objeto
datosForm que fue incluido por Struts en el contexto) --%>

<bean:write name=”datosForm” property=”nombre” /><br/>
<bean:write name=”datosForm” property=”edad” /><br/>
<bean:write name=”datosForm” property=”fumador” /><br/>
```

La Figura 2 muestra un posible resultado de la aplicación en ejecución.

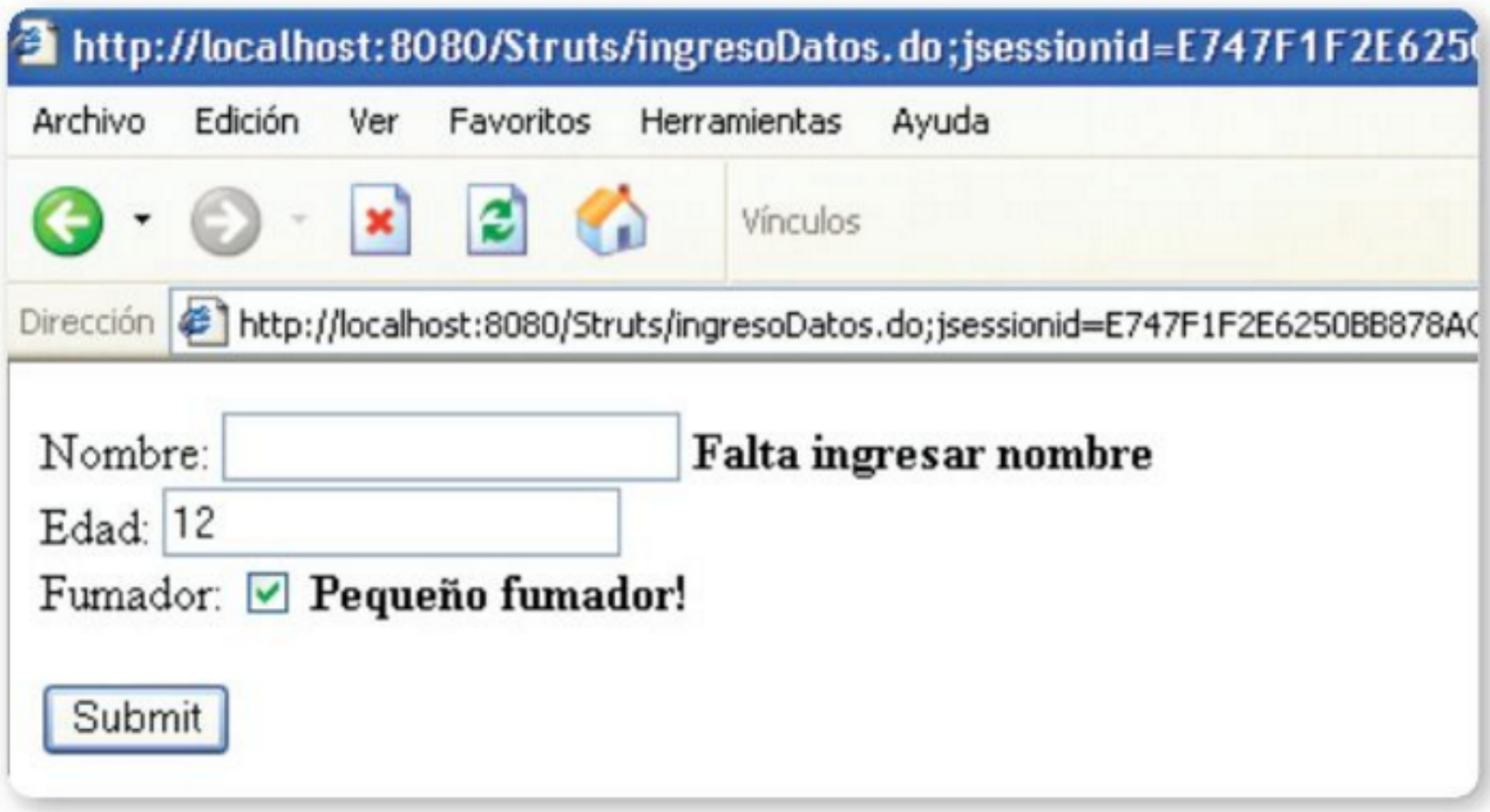


Figura 2. La validación se realiza en el servidor antes de invocarse la acción.

Vemos que en el formulario usamos tags propios de Struts para mantener el estado del formulario (entre otras cosas). En los próximos capítulos, veremos los tags que Struts provee junto con la descripción y utilización de cada uno de ellos.

Alternativas

A medida que nuestra aplicación web va creciendo, también lo hace la cantidad de clases ActionForm . Si usamos la funcionalidad de Struts para validación (Struts Validator , que veremos en los próximos capitulos), entonces nuestras clases se convierten en objetos JavaBean que son meramente un repositorio de propiedades con métodos de acceso y que no aportan funcionalidad. En estos casos, es conveniente buscar una alternativa, en vez de tener que crear una clase con sus propiedades y accesos para cada formulario que necesitemos.

DynaActionForm

Si no aportamos funcionalidad a los forms, estos terminan cumpliendo el papel de un diccionario o mapa: una colección de campos con sus respectivos valores. Struts provee la clase DynaActionForm , que se comporta exactamente de esta manera, ahorrándonos el trabajo de escribir forms por doquier. Para usar este tipo de forms, simplemente necesitamos definirlos en el archivo struts-config y declarar sus propiedades. Por ejemplo:

```
<form-bean
        name=”datosForm”
        type=”org.apache.struts.action.DynaActionForm”>
    <form-property name=”nombre” type=”java.lang.String” />
    <form-property name=”edad” type=”java.lang.Integer”/>
    <form-property name=”fumador” type=”java.lang.Boolean”/>
</form-bean>
```

Esta definición podría reemplazar a la clase DatosForm que definimos al principio del capítulo, aunque con este DynaActionForm perdemos la validación. Si bien la clase DynaActionForm permite tipos primitivos en sus

propiedades, si se intenta asignarles un valor nulo obtendremos una excepción, por lo que es conveniente usar siempre objetos.

Las opciones del elemento form-property son:

- className : nombre de la clase que configura este elemento. Es un atributo que funciona igual que en la definición de un form-bean . La superclase que deben extender las clases de definición de propiedades es org.apache.struts.config.FormPropertyConfig .
- initial : valor inicial que tomará esta propiedad (representada por una cadena de texto). Si omitimos este valor, la propiedad será creada usando el constructor sin argumentos de la clase.
- name : es el nombre de la propiedad.
- size : es el número de elementos a crear si el tipo de esta propiedad es un arreglo y no se especificó un valor inicial de creación.
- type : es el nombre de la clase que instanciará el bean.

Los arreglos son necesarios cuando el usuario tiene que elegir una cantidad de ítems de una lista múltiple.

Un aspecto positivo de los DynaActionForm es que no necesitamos modificar nada de código para acceder a ellos en las páginas JSP. Si estamos migrando viejos ActionForm a DynaActionForm , sí tenemos que modificar ligeramente la forma de acceder a las propiedades. El método para obtener los valores de los campos en un DynaActionForm es el siguiente:

INTERNACIONALIZACIÓN

Idealmente, no deberíamos tener ningún tipo de texto en nuestro código; Struts nos ayuda con la internacionalización. Los mensajes son externalizados a archivos, y eso nos facilita hacer múltiples versiones en diferentes idiomas, para poder mostrarle al usuario un mensaje en su propio idioma.

```
public Object get(String name)
```

El parámetro corresponde al nombre de la propiedad a acceder y el resultado es el valor de dicha propiedad.

La ventaja principal de usar DynaActionForm es prescindir de escribir una clase completa para cada formulario web que necesitemos. Sin embargo, este cambio no es gratis. Entre sus desventajas, podemos nombrar:

- El método get devuelve un Object , lo cual implica que debemos castear el resultado. Al ser el casteo dinámico, no se lo puede predecir en tiempo de compilación, y, si cometimos un error, nos enteraremos (eventualmente) en tiempo de ejecución.
- Lo mismo ocurre con el nombre del campo. Podemos equivocarnos en el nombre y el compilador interpretará una llamada a función totalmente válida. Recién al ejecutar nos enteraremos del error.

Pese a estas dificultades, usar DynaActionForm es muy conveniente cuando el formulario precisa validaciones simples que pueden implementarse usando el Struts Validator.

LazyValidatorForm

Esta clase se basa en la clase LazyDynaBean , un bean dinámico como el usado por DynaActionForm , pero con la posibilidad de agregarle y quitarle elementos en tiempo de ejecución. Esto nos libera incluso de tener que definir las propiedades del form en el struts-config .

Aunque no es necesario, es conveniente declarar los arreglos que vayamos a usar, porque, si no lo hacemos, puede haber complicaciones al usar esta clase en conjunto con el validador de Struts.

Ante la ventaja de no tener que declarar las propiedades de los formularios, tenemos la desventaja de que LazyDynaBean no sabe los tipos que queremos asignarle, por lo que, al recibir cadenas de texto, siempre

tendremos cadenas de texto y no hará conversiones de tipo. Esto podría no ser un inconveniente si no dependemos de las conversiones de tipo y usamos las propiedades de los forms siempre como objetos de tipo string.

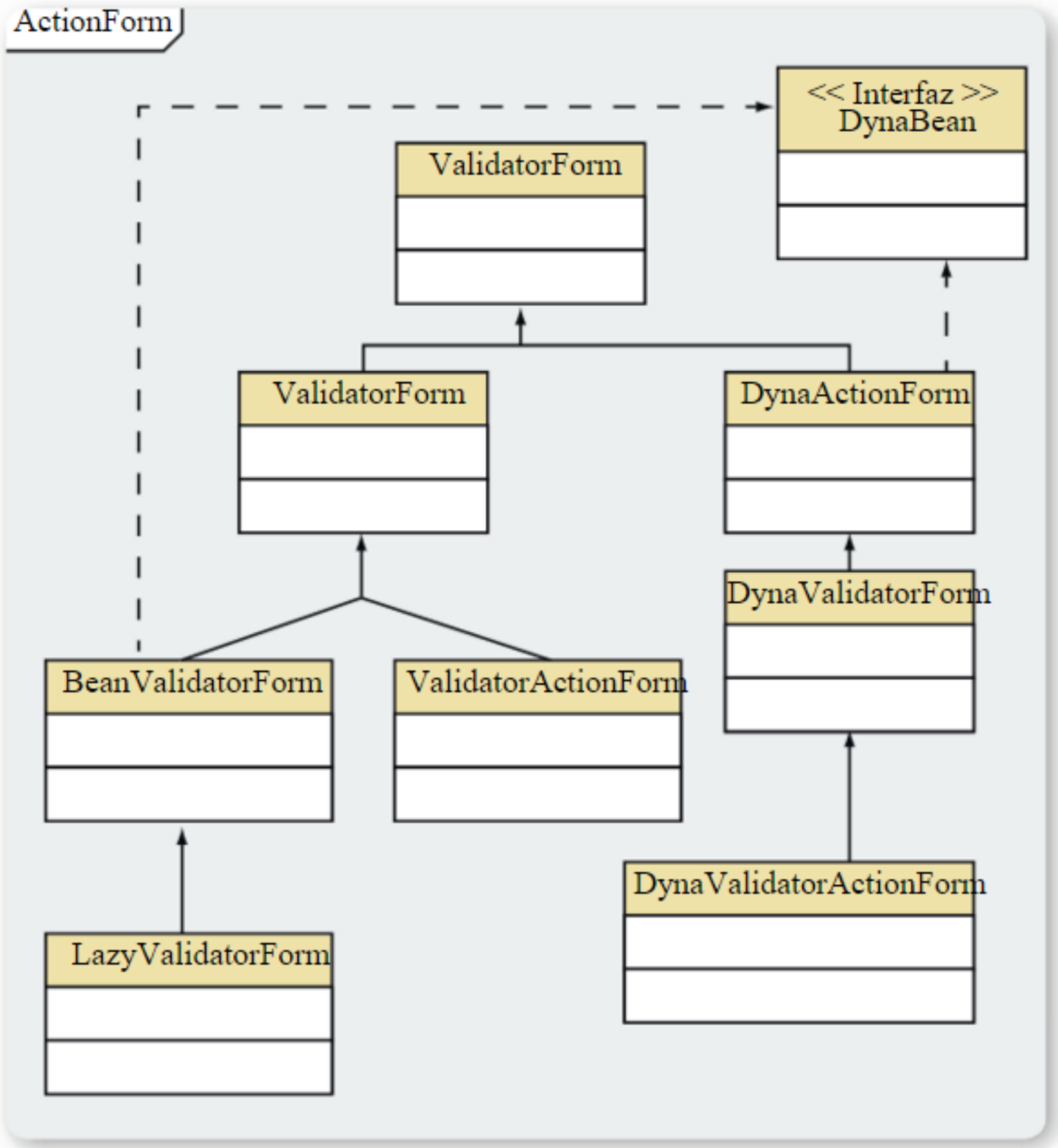


Figura 3. Todas las alternativas siguen siendo ActionForm.

¿Lazy Validator?

El lector atento habrá notado que las otras clases eran ActionForm, DynaActionForm y que, ahora, esta clase no se llama, como podríamos asumir, "LazyActionForm", sino LazyValidatorForm . ¿Por qué?

De hecho, tanto ActionForm como DynaActionForm tienen de subclases a ValidatorForm y DynaValidatorForm , respectivamente.

Estas clases —¡adivinaron!— están relacionadas con el uso del validador de Struts. En el capítulo dedicado a validación, veremos que tendremos que usar ValidatorForm o DynaValidatorForm si queremos usar dicho paquete. El autor de LazyValidatorForm implementó directamente la versión que funciona con validador. A causa de esto, es posible que un ejemplo que use LazyValidatorForm no funcione si no está definido el plugin de Struts Validator.

Ejemplo en JSP

Para aplicar lo que aprendimos en este capítulo, proponemos el siguiente ejemplo de formulario en páginas JSP para luego aplicar las facilidades que vimos en ActionForm y DynaValidatorForm .

Tenemos dos archivos .jsp que utilizaremos: uno hará de formulario y el otro, de resultado. Veamos el código. Creamos el archivo index.jsp :

```
<html>
<head>
<title>Formulario Principal</title>

</head>
<body>

<form action=”proceso.jsp” method=”post”>
        Nombre:
        <input type=”text” name=”nombre”>
        <br/>
        Apellido:
        <input type=”text” name=”apellido”>
        <br/>
        Edad:
        <input type=”text” name=”edad”>
```

```
<br/>
Lenguaje preferido:
<select name=”lenguaje”>
 <option value=”java”>java
 <option value=”jsp” selected>jsp
 <option value=”php”>php
 <option value=”C/C++”>C/C++
 <option value=”C#”>C#
  <option value=”Asp”>Asp
  <option value=”AS3”>AS3
</select>
<br/>
Me gusta el:
<br/>
<input type=”Radio” name=”preferencia” value=“Diseño”checked>Diseño
<br/>
<input type=”Radio” name=“preferencia”
value=”Programacion”>Programacion
<br/>
<input type=”Radio” name= “preferencia”value=”Modelado”>Modelado
<br/>
<input type=”Radio” name= “preferencia”
value=”Gerencia”>Gerencia de proyectos
<br/>


<p><input type=”submit” value=”Enviar”></p>
</form>

</body>
</html>
```

Creamos el archivo de resultado proceso.jsp :

```
<html>
<head>
<title>HOLA FORMULARIOS</title>

</head>
<body>

<%
  /*podemos leer los datos del request a una variable*/
  String edad=(String)request.getParameter("edad");
  String prefieres=(String)request.getParameter("preferencia");
  out.print("tu nombre es "+request.getParameter("nombre")
+""+request.getParameter("apellido"));
  out.print("<br/>");
  out.print("tienes "+edad+" años");
  out.print("<br/>");
  out.print("tu lenguaje favorito es "+request.getParameter("lenguaje"));
  out.print("<br/>");
  out.print("y prefieres el(a) "+prefieres+" de un proyecto");
  out.print("<br/>");
  /*podemos usar los datos directamente desde el request*/
  out.print("Bienvenido a jsp "+ request.getParameter
("nombre").toString().toUpperCase());
 %>

</body>
</html>
```

Luego ejecutamos el proyecto desde el archivo index.jps en el servidor Tomcat predeterminado. Los resultados deberían ser los siguientes:

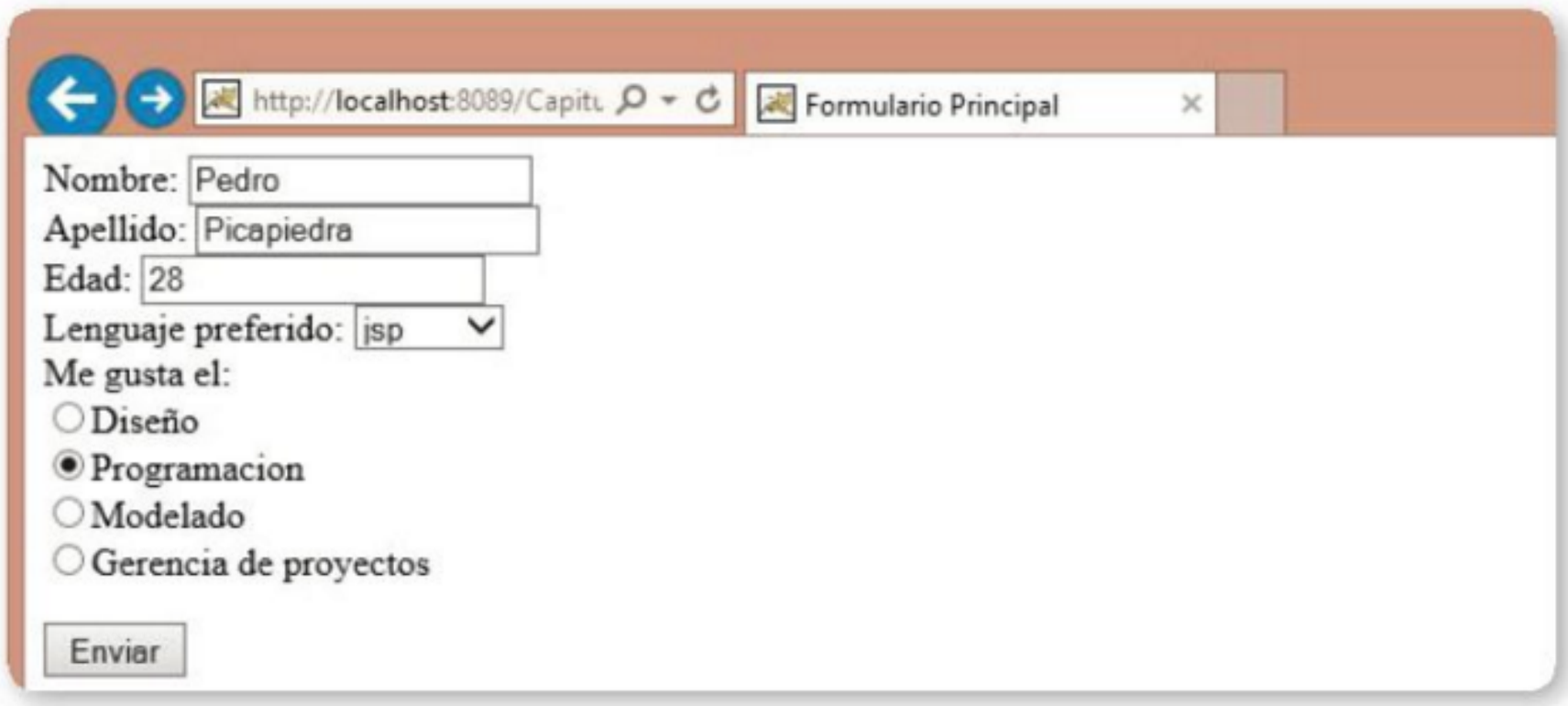


Figura 4. Resultado del modelo de páginas dinámicas.

RESUMEN

En este capítulo vimos los ActionForms, su importancia dentro de Struts y cómo usarlos para obtener información del usuario en una forma portable y elegante. También vimos cómo validar los datos que el usuario ingresa y alternativas para no escribir una clase por cada formulario.

Actividades

TEST DE AUTOEVALUACIÓN

1. ¿Qué aportan los ActionForms?
2. ¿Por qué es preferible usar ActionForms en vez de interactuar directamente con el `HttpServletRequest` ?
3. ¿Por qué se considera un ActionForms como un firewall entre el pedido HTTP y la acción que se ejecuta?
4. ¿Cuál es la complicación de usar tipos primitivos como propiedades de los ActionForm?

EJERCICIOS PRÁCTICOS

1. Modifique la clase `DatosForm` para que, además, verifique que `edad` sea un número positivo.
2. Cree un ActionForm con propiedades `fechaDeNacimiento` y `talle` . Escriba un método para validar ambos campos según su criterio (ayuda: use `SimpleDateFormat` para validar fecha).
3. Tome el último ejemplo de JSP y mejore su programación con ActionForm.
4. Tome el último ejemplo de JSP y mejore su programación con DynaValidatorForm.

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: profesor@redusers.com .

*06

Actions en Struts

Luego de estudiar las facilidades de ActionForms, nuestro próximo paso es tratar los Action en Struts. En este capítulo veremos para qué sirven, cómo programarlos, las diversas alternativas que ofrecen y cómo interactúan con la vista usando tags propios de Struts.

Programar el controlador

En el capítulo anterior vimos que un ActionForm es como un firewall entre el pedido que hace el cliente y el servidor: no deja pasar información no deseada, y, la que pasa debe ser validada.

Las clases Action hacen las veces de interfaces entre el pedido HTTP que efectúa el cliente y las clases que han de efectuar la lógica subyacente. Bien podríamos realizar todas las operaciones dentro del código en los Action, pero esto implica mezclar responsabilidades: la lógica estará altamente atada al entorno web (y, en particular, a Struts) y no podremos reutilizar componentes de software.

Una buena práctica de programación web es programar los Action como adaptadores entre objetos propios del entorno web y de Struts (HttpServletRequest, ActionForward, etcétera) y los JavaBeans. Deben recibir la información proveniente de las páginas, validarla, procesarla y enviársela a los JavaBeans para que hagan lo suyo.

Finalmente, envían el resultado a la vista, en donde esta consultará a los beans su estado para mostrar al cliente una página web.

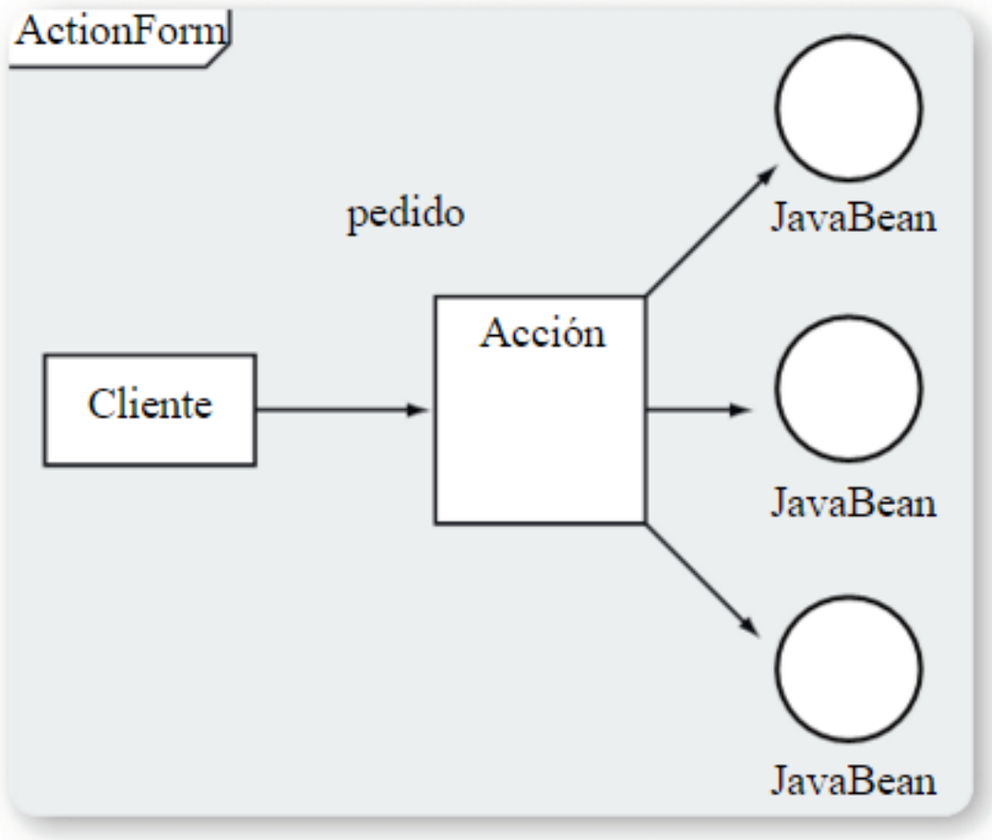


Figura 1. Si aislamos los JavaBeans del entorno web, nos resultará mucho más fácil la reutilización en otros entornos.

Struts reutiliza las instancias de las clases Action que definamos para manejar pedidos simultáneos; debemos programar las Action de forma segura a la ejecución concurrente: las variables (tanto de instancia como de clase) no deben usarse para mantener información del estado actual del pedido, y el acceso a recursos externos debe estar sincronizado si dicho recurso lo requiriera.

Configurar un Action

Veamos las opciones que tenemos para configurar un action. Los siguientes son algunos atributos posibles de Action en el struts-config :

- attribute : es el nombre que queramos darle al ActionForm para ser guardado en el pedido o en la sesión. Si no especificamos este atributo, se usará el nombre del ActionForm tal como aparece en el atributo name .
- forward : es la dirección del recurso (otro action, un JSP, un servlet u otro recurso que pueda manejar el pedido) que ha de procesar este pedido.
- include : dirección del recurso que se ha de incluir en la respuesta que genere este pedido.

La diferencia de funcionalidad entre estos dos últimos atributos es similar a los distintos tipos de inclusión que existen en las páginas JSP.

Usando forward creamos una acción que envía el pedido al recurso que especifiquemos, y ese recurso se encargará de procesarlo y generar la respuesta. En cambio, si usamos include, estamos incluyendo en nuestra respuesta

EVITANDO SINCRONIZACIONES

En general, una buena práctica consiste en no utilizar directamente ningún tipo de variables globales (ni de instancia, ni de clase) dentro de las clases y declarar todas las variables a ser usadas como locales dentro del método execute .

el resultado que genere el recurso especificado, pero no estamos derivando completamente el pedido a dicho recurso.

- type : nombre completo de la subclase de org.apache.struts.action.Action que procesará los pedidos que este elemento describe.

En cada elemento action los atributos forward , include y type son excluyentes entre sí, solo puede haber uno de ellos.

- input : dirección del recurso a invocar si el ActionForm asociado al elemento (en caso de que exista) reporta errores de validación.
- name : nombre del form-bean asociado (tal como fue definido dentro del elemento form-beans), si hubiere.
- path : dirección que este elemento procesará.
- parameter : parámetro de uso general, usado para pasar información extra al Action. Podemos crear una clase llamada BDAction y que tome de este parámetro el nombre de la conexión a la base de datos. Luego, las acciones que requieran conexión a una base simplemente extenderán BDAction
- scope : contexto en donde reside el form-bean : request o session (este tema fue expuesto en el Capítulo 5).
- unknown : marcador (solo puede tomar valores true o false) usado para configurar la acción por defecto a ser ejecutada. Si se recibe un pedido que no está asociado con ninguna acción, el usuario recibirá un error de parte de Struts. Si declaramos una acción con el atributo unknown verdadero, será la acción por defecto a ejecutarse en estos casos; esto es útil para mostrar al usuario un mensaje de error agradable o redirigirlo a una página de navegación. Solo puede haber una sola acción con este atributo con valor true .
- validate : marcador (solo puede tomar valores true o false) usado para indicar si el form-bean debe ser validado antes de ser enviado al objeto Action y este, ejecutado. Por defecto, este atributo lleva el valor true .

En todos los atributos donde hemos de especificar direcciones, estas deben empezar con una barra (/).

Veamos ejemplos de configuraciones, con sus respectivas explicaciones:

```
<action
        path="/formulario"
        forward="/jsp/capitulo6/formulario.jsp">
```

Un elemento simple que asocia una acción directamente con una página JSP. Este tipo de configuración es útil para esconder las páginas JSP detrás de acciones, una buena práctica por temas de seguridad (que veremos luego).

```
<action
        path="/footer"
        include="/jsp/comun/footer.jsp">
```

Similar al anterior, pero usando include , este ejemplo es de utilidad para pies de página o elementos que se repiten mucho. De nuevo, esconde las páginas JSP de la interacción directa con el usuario.

```
<action
        path="/accionXYZ"
        name="xyzForm"
        validate="false"
        input="xyz.jsp"
        type="xyz.XYZAction">
```

En este raro ejemplo, declaramos que el bean no debe ser validado; sin embargo, estamos definiendo la página donde debe reenviarse en caso de que la validación falle. Si bien la página no será accedida nunca, esto es sintácticamente correcto.

```
<action path="/" unknown="true" forward="/jsp/error.jsp" />
```

Este ejemplo sirve para manejar todos los pedidos que Struts reciba y no tengan asociados un elemento `action`, y redirigirlos a una página JSP donde se muestre un mensaje de error.

ActionForward

Todo elemento `action` puede tener opcionalmente un conjunto de elementos que describan `ActionForward`, los cuales pueden ser accedidos desde el `action` para comunicarle al controlador hacia dónde debe seguir el flujo de la acción. Un `ActionForward` es una clase que contiene información sobre hacia dónde debe enviar el pedido el controlador como resultado de una acción.

Los tres atributos elementales para definir un `ActionForward` son:

- `name` : nombre que queramos darle al `ActionForward`.
- `path` : dirección de destino.
- `redirect` : si este atributo toma valor `true`, entonces un nuevo pedido será emitido para acceder al destino de este `ActionForward`.

ACTIONFORWARD CONTIENE LA INFORMACIÓN SOBRE HACIA DÓNDE ENVIAR UNA ACCIÓN

Al emitirse un nuevo pedido para el destino del `ActionForward`, estamos perdiendo todos los datos que el pedido actual tenía: datos de formularios, datos guardados por un `Action`, entre otros, se pierden. Esto puede ser beneficioso o perjudicial. Si necesitamos alguna de estas informaciones para poder construir la vista, obtendremos un error o bien la vista estará incompleta. El hecho de emitir un nuevo pedido es útil para cuando el usuario efectúa una acción con efectos colaterales y no queremos que la repita por error.

Por ejemplo, si transfirió una cantidad de dinero entre cuentas de un banco. En este caso, la acción fue "transferir X cantidad de dinero entre cuenta A y cuenta B", que, como resultado final, tuvo un ActionForward a una página que confirmó la operación. Si el usuario, accidentalmente o no, presionara el botón de recargar página en el navegador, estaría reenviando el pedido de transferir nuevamente el efectivo, algo que dudosamente sea lo que él quiera. Si el ActionForward de la operación de transferencia hacia la página que muestra que se efectuó correctamente emite un nuevo pedido, entonces el último pedido será "ir hasta la página de confirmación", y no habría problemas si el usuario recargara la página.

Veamos algunos ejemplos de ActionForward :

```
<forward name="pedido" path="/jsp/bla.jsp" />
<forward name="confirmacion" path="/jsp/confirm.jsp" redirect="true"/>
```

Actions para todos

El paquete org.apache.struts.actions provee varias subclases de Action con diversas funcionalidades comunes ya implementadas. Podemos descargar estos paquetes desde las direcciones web vistas en el capitulo anterior.

Como ya vimos, Struts invoca el método execute dentro de los objetos Action . Esto implica que debemos crear una clase por cada acción. Sería más conveniente poder tener varias acciones dentro de un mismo objeto Action , de modo de agrupar las acciones según lo que realicen. Las clases DispatchAction y sus subclases sirven exactamente para esto.

Para estos ejemplos, podemos descargar cualquier paquete que necesitemos desde esta excelente página de recursos: www.findjar.com

DispatchAction

DispatchAction llama al método según el valor de un parámetro del pedido, cuyo nombre es especificado por la propiedad parameter en el

elemento action del struts-config . ¿Confuso? Veámoslo en la práctica. Uno de los atributos del elemento action era parameter , cuya función era ser de uso general para pasar información extra al objeto Action . En este caso, vamos a hacer uso de ese parámetro. Declaramos una acción en struts-config .

```
<action
        path="/cuentas"
        type="capitulo6.CuentaAction"
        name="cuentaForm"
        scope="request"
        parameter="metodo"
/>
```

Definimos la clase:

```
package capitulo6;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.DispatchAction;

public class CuentaAction extends DispatchAction {

    public ActionForward depositar(ActionMapping map, ActionForm form,
            HttpServletRequest req, HttpServletResponse res) throws Exception {
        // Acciones de depósito
    }
```

```
    public ActionForward extraer(ActionMapping map, ActionForm form,
            HttpServletRequest req, HttpServletResponse res) throws Exception {
        // Acciones de extracción
    }
}
```

Y he aquí una página minimalista JSP que interactúa con esta acción. La llamaremos formulario.jps :

```
<a href="cuentas.do?metodo=depositar">Depositar</a>
<br/>
<a href="cuentas.do?metodo=extraer">Extraer</a>
<br/>
<a href="cuentas.do">Error!</a>
```

Al acceder a esta acción, Struts buscará el parámetro de nombre metodo en el pedido, obtendrá su valor y llamará al método de ese nombre. Si el parámetro no existe, o si el valor del parámetro no coincide con ningún método, obtendremos una excepción. Es importante destacar que no alcanza con que el método coincida con el nombre: también debe coincidir con los parámetros recibidos. Por eso, al implementar estos métodos, debemos asegurarnos de respetar la aridad del método execute .

MappingDispatchAction

En este caso, no necesitamos de un parámetro en el pedido; el método a ser llamado es el valor del atributo parameter . Utilicemos el ejemplo anterior. Usando MappingDispatchAction , sería:

ORG.APACHE.STRUTS.ACTION TIENE SUBCLASES DE ACTION IMPLEMENTADAS

```
<action
        path=”/extraccionCuentas”
        type=”capitulo6.CuentaAction”
        name=”cuentaForm”
        scope=”request”
        parameter=”extraer”
/>

<action
        path=”/depositoCuentas”
        type=”capitulo6.CuentaAction”
        name=”cuentaForm”
        scope=”request”
        parameter=”deposito”
/>
```

La clase no varía, tiene los dos métodos con los mismos nombres.

```
<a href=”depositoCuentas.do”>Depositar</a>
<br/>
<a href=”extraccionCuentas.do”>Extraer</a>
<br/>
```

ForwardAction, IncludeAction

Estas dos clases proveen la misma funcionalidad que los atributos forward e include en los elementos action . En ambos casos, el valor del atributo parameter indica el recurso a redirigir o incluir, respectivamente.

En concreto, los siguientes ejemplos son equivalentes:

```
<action
        path=”/verFormulario”
        forward=”/jsp/capitulo6/formulario.jsp”>

<action
path=”/verFormulario” type=”org.apache.struts.actions.
ForwardAction” parameter=”/jsp/capitulo6/formulario.jsp”/>
```

Igualmente para el caso de IncludeAction :

```
<action
        path=”/footer”
        include=”/jsp/comun/footer.jsp”>

<action path=”/footer” type=”org.apache.struts.actions.
IncludeAction” parameter=”/jsp/comun/footer.jsp”>
```

DownloadAction

Esta clase brinda la funcionalidad básica para manejar la descarga de archivos por parte del usuario. Usar Struts en vez de directamente crear un enlace al archivo a descargar nos sirve para no revelarle al usuario la ubicación del archivo y, fundamentalmente, para poder realizar comprobaciones (típicamente, permisos) antes de permitir la descarga, además del hecho de que un enlace es estático y, usando este enfoque, la ubicación del archivo (incluso el contenido mismo del archivo) se puede obtener dinámicamente. DownloadAction es una clase abstracta; nuestra subclase debe implementar el método abstracto getStreamInfo y, opcionalmente, getBufferSize .

- getBufferSize : simplemente devuelve un entero con el tamaño del buffer a ser usado por el servlet al transferir la información al cliente.
- getStreamInfo : devuelve un objeto con la información sobre los datos a transferir, del tipo DownloadAction.StreamInfo , que es una interfaz. Los dos métodos que declara esta interfaz son:

```
String getContentType()
```

Con este método le informamos al cliente (el navegador) el tipo de archivo que estamos enviando.

```
InputStream getInputStream()
```

Aquí definimos la fuente de los datos a enviar.

La clase DownloadAction provee dos clases estáticas que implementan esta interfaz, listas para ser usadas. Veamos sus constructores, que son todo lo que necesitamos de ellas:

```
DownloadAction.FileStreamInfo(String contentType, java.io.File file)

DownloadAction.ResourceStreamInfo(String contentType,
ServletContext context, String path)
```

En el primer caso, proveemos el contentType y, luego, un objeto File con el archivo que se descargará. Esto es útil para archivos fijos y no variables, ya que debemos proveer una dirección absoluta.

En el segundo caso, además del contentType , debemos proveer el contexto del servlet y una dirección relativa. Este caso es más útil para devolver recursos de la aplicación web, ya que estamos limitados en las direcciones a usar. Veamos tres ejemplos, uno usando FileStreamInfo , otro

con ResourceStreamInfo y un tercero en donde implementamos nosotros mismos la interfaz. Crearemos la clase DownloadFileAction :

```
package capitulo6.download;

import java.io.File;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.DownloadAction;

public class DownloadFileAction extends DownloadAction {

    protected StreamInfo getStreamInfo(ActionMapping map, ActionForm form,
            HttpServletRequest req, HttpServletResponse res) throws Exception {
        return new FileStreamInfo("image/gif", new File("/home/tulsi/logo.gif"));
    }
}
```

TIPOS DE CONTENIDO

Los diferentes tipos de contenido que existen son manejados por IANA (Internet Assigned Numbers Authority), una entidad que supervisa la asignación de direcciones IP, sistemas autónomos y otros recursos relativos a los protocolos de internet.

En su página web, www.iana.org/assignments/media-types , se pueden consultar todos los tipos existentes e incluso registrar nuevos.

Creamos la clase DownloadResourceAction :

```
package capitulo6.download;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.DownloadAction;

public class DownloadResourceAction extends DownloadAction {

    protected StreamInfo getStreamInfo(ActionMapping map, ActionForm form,
            HttpServletRequest req, HttpServletResponse res) throws Exception {

        return new ResourceStreamInfo("application/pdf",
                getServlet().getServletContext(), "/pdf/mapreduce.pdf");
    }

    // Este archivo es grande, agrandamos el buffer
    protected int getBufferSize() {
        return 8192;
    }
}
```

Creamos la clase DownloadCustomAction :

```
package capitulo6.download;

import java.io.ByteArrayInputStream;
import java.io.IOException;
```

```
import java.io.InputStream;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.DownloadAction;

public class DownloadCustomAction extends DownloadAction {

    protected StreamInfo getStreamInfo(ActionMapping map, ActionForm form,
            HttpServletRequest req, HttpServletResponse res) throws Exception {

// Una clase interna que implemente la interfaz
        return new StreamInfo() {
            public String getContentType() {
                return “text/plain”;
            }

            public InputStream getInputStream() throws IOException {
                String texto = “texto a devolver”;
                return new ByteArrayInputStream(texto.getBytes());
            }
        };
    }
}
```

Como vemos en estos ejemplos, con poco código podemos enviar al usuario directamente archivos que podrán ser guardados o manejados por el navegador y quizá mostrados directamente (muchos navegadores modernos tienen funcionalidades para manejar archivos PDF, por ejemplo).

En la práctica, las acciones de descarga de archivos van asociadas con restricciones al usuario y, además, por lo general, el archivo o contenido a descargar es dinámico.

LocaleAction

Esta acción nos permite cambiar el Locale del usuario. Por defecto, Struts configura un Locale basándose en propiedades enviadas por el navegador. En muchos casos, el usuario no configuró el navegador para elegir su idioma, o está en otra máquina, o simplemente quiere cambiar el idioma de navegación. Con esta acción podemos cambiar fácilmente el objeto Locale que Struts guarda internamente, y, si nuestra aplicación soporta internacionalización, cambiaremos el idioma en que el usuario navega.

Esta acción se define normalmente en el struts-config . Al ejecutarse, busca en el pedido los parámetros language , country y page .

Los valores de language y country definen el nuevo Locale . El parámetro page define la página adonde dirigirse luego del cambio de Locale . Si el parámetro page no existiera, se redirecciona el ActionForward de nombre success . Dado este funcionamiento, es altamente recomendable definir un ActionForward de salida con ese nombre si pensamos usar esta acción: aunque siempre pasemos el parámetro page , nos resguardará de posibles errores al hacer cambios o adaptaciones.

Un uso típico de esta clase es el que se muestra a continuación.

Definimos la acción en struts-config :

DEFINIR UN LOCALE

Los parámetros language y country definen un Locale en Java. La lista de los posibles lenguajes y países se encuentra disponible en www.ics.uci.edu/pub/ietf/http/related/iso639.txt y www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html respectivamente.

```
<action
        path=”/cambiarIdioma”
        type=”org.apache.struts.actions.LocaleAction”
        scope=”request”>
        <forward name=”success” path=”/index.jsp” />
</action>
```

Y la llamamos desde una página JSP, idioma.jsp :

```
<a href=”cambiarIdioma.do?language=es”>Espa&ntilde;ol</a>
<a href=”cambiarIdioma.do?language=en&country=
za”>South African English</a>
<a href=”cambiarIdioma.do?language=en&country=uk”>British English</a>
```

En este caso, no usamos el parámetro page , y el cambio de idioma siempre vuelve a la página de inicio. Si estas opciones estuvieran en varios lados, volver siempre a la página de inicio sería muy molesto para el usuario, y convendría mantener la página con el idioma cambiado.

RESUMEN

En este capítulo pudimos conocer a fondo los objetos Action , su funcionamiento, su función en el modelo MVC y su interacción con los demás componentes. Vimos también cómo prevenir los problemas más comunes.

Actividades

TEST DE AUTOEVALUACIÓN

1. ¿Cuál es la función de los objetos Action ?
2. ¿Qué representa un ActionForward ?
3. ¿En qué caso se utilizaría DispatchAction y en cuál MappingDispatchAction ?
4. ¿Por qué sólo puede haber una acción con el atributo unknown con valor true ?

EJERCICIOS PRÁCTICOS

1. Modifique el ejemplo del Capítulo 5 y escriba la validación en la acción, en vez de hacerlo en el form-bean .
2. Escriba una acción que permita al usuario descargar el archivo struts-config.xml de la aplicación.
3. Escriba una acción genérica que reenvíe el pedido a una dirección dada, pero antes guarde la marca en la sesión (sugerencia: escriba una subclase de ActionForward).

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: profesor@redusers.com .

*07

Vista

En este capítulo veremos cómo programar la vista. Para ello, en las próximas páginas estudiaremos cómo generar páginas JSP con tags de Struts y cuáles son sus principales utilidades.

Hasta la vista, baby

Llegó el momento de concentrarnos en el tercer pilar del modelo MVC: la vista . En este caso, consiste en generar páginas HTML mediante el uso de JSP. Una vez más, no debemos olvidar que, si bien Struts ofrece una gran variedad de herramientas y funcionalidades para facilitar la creación de la vista con JSP, no es la única forma de crear páginas HTML, ya que trae también soporte (aunque menor) para otras formas como Velocity , XSL, entre otras.

Tags de Struts

Struts define varios tags propios para la creación de la vista. Dichos tags agregan funcionalidad y nos permiten completar automáticamente campos al cargar una página, cambiar el estilo a un formulario si ocurrió algún error, cargar opciones de colecciones, etcétera.

En el Capítulo 4 definimos, en el archivo web.xml , un conjunto de descriptores de los tags de Struts, como vemos a continuación:

```
<taglib>
   <taglib-uri>/tags/struts-html</taglib-uri>
   <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
 </taglib>
```

Esta configuración asocia la ubicación de los descriptores de los tags (que definen los posibles parámetros de entrada y la clase que los implementa, entre otras cosas) con un identificador. Este identificador es el que debemos usar en las páginas JSP para indicarle al motor JSP (en el caso de Tomcat, será Jasper 2) dónde buscar las definiciones y clases de los tags. Por ejemplo:

```
<%@ taglib uri=”/tags/struts-html” prefix=”html”%>
```

En este caso, el atributo uri debe coincidir con algún elemento taglib-uri definido en el archivo web.xml . El atributo prefix indica el prefijo a usar dentro de la página.

Errores en los tags

La mayoría de los tags de Struts están definidos para arrojar una excepción del tipo JspException en caso de error. Por eso, es importante declarar una página de error (en la directiva <%@ page %>) de la página JSP para procesar estas excepciones cuando ocurran.

Bean tags

Estos tipos de tags son útiles para el manejo de JavaBeans. Si bien JSP provee por defecto tags para el manejo de beans, los tags de Struts brindan funcionalidad extra que es de mucha utilidad para el manejo de componentes JavaBean.

Acceso a propiedades de beans

Los tags de beans de struts poseen una sintaxis que permite el acceso a propiedades simples, anidadas e indexadas.

- Las propiedades simples son las que ya conocemos: si estamos usando un tag que accede a una propiedad xyz , se traducirá en getXyz() .
- En cambio, las propiedades anidadas permiten descender en las propiedades del bean. Supongamos que tenemos un bean Persona que tiene una propiedad mascota que, a su vez, tiene una propiedad raza , y esta, a su vez, posee la propiedad nombre ; en este caso, podemos hacer referencia al nombre de la raza de la mascota de la persona usando la propiedad mascota.raza.nombre , que se traducirá en getMascota().getRaza().getNombre() .
- En los casos en que estemos usando la propiedad para establecer valores (por ejemplo, un setter), solo el último método es llamado como

setter y los demás como getter . En este último ejemplo, el código generado será: getMascota().getRaza().setNombre(valor).

- Las propiedades indexadas se emplean cuando tenemos propiedades con múltiples valores, es decir, una propiedad que es un conjunto de elementos. En estos casos, los métodos getter y setter deben, además, suministrar un parámetro entero indicando a qué elemento de la colección se están refiriendo. Por ejemplo:

```
class Persona {
    private List mascotas;

    public Mascota getMascota(int pos) {
        return (Mascota) mascotas.get(pos);
    }

    public void setMascota(Mascota m, int pos) {
        mascotas.set(pos, m);
    }
}
```

En estos casos, podemos suministrar la propiedad indexada mascota[2] que se traducirá en getMascota(2) .

bean:write

Vamos a comenzar por ver en detalle el tag más simple del paquete de los tags de beans. Este tag simplemente devuelve el contenido del bean o una de sus propiedades. En esencia, este tag es igual a <jsp:getProperty> , pero con las mejoras que vimos previamente.

Los atributos que podemos proveer son los siguientes:

- name : especifica el nombre del bean al que estamos accediendo.
- property : la propiedad a la que estamos accediendo (soporta la sintaxis extendida de Struts, propiedades simples, anidadas e indexadas).

Si no especificamos propiedad, se devuelve el bean.

- scope : especifica el scope donde buscar el bean.
- ignore : si este atributo es true , y el bean que especificamos no existe en el scope indicado, entonces el tag no devuelve nada. Caso contrario, obtendremos una excepción.
- filter : este atributo con valor true indica que el resultado de este tag será filtrado por los caracteres sensitivos HTML que contenga, y estos serán reemplazados por su entidad correspondiente. Esto es: si la evaluación del tag –consistente en una propiedad de determinado bean– devuelve retoño , y si el atributo filter es verdadero, se filtrará la letra eñe y se reemplazará por &ntilde , de forma que el resultado sea compatible con HTML y visualizable en cualquier navegador.
- formatKey : especifica la clave de los mensajes de aplicación (MessageResources) en donde buscar la cadena de formato a aplicar al resultado del tag.
- format : especifica la cadena de formato que ha de aplicarse al resultado de este tag. El resultado de la evaluación de este tag siempre es, a posteriori , un objeto de tipo string. Sin embargo, una gran cantidad de veces estamos accediendo a objetos que no son de tipo string y debe efectuarse una conversión. Struts evalúa el tipo del objeto accedido y, en ciertos casos, puede aplicar una transformación que implique formateo de los datos. Struts aplica el formato a objetos enteros, decimales y fechas.

CREAR UN BEAN EN LAS PÁGINAS JSP

Sólo con los tags de Struts no tenemos forma de crear un bean en las páginas JSP. Según el paradigma MVC, esto no tendría que pasar nunca: todos los objetos (el modelo) son creados en las acciones y son accedidos luego en las páginas JSP para mostrar propiedades. Sin embargo, si necesitáramos crear un bean (no recomendado), podemos usar el tag jsp:useBean .

Veamos un ejemplo de uso de este tag. Para simplificarlo un poco, vamos a hacer algunas cosas no muy acordes con las buenas prácticas de programación web, como ser: usar código caduco y crear un objeto usando código en la página JSP. Pero, a los efectos del ejemplo, sirve, así no tenemos que crear una acción, definirla, etcétera.

```
<!-- Declaramos que vamos a usar la librería de tags
de beans y que la usaremos con el prefijo bean -->
<%@ taglib uri=”/tags/struts-bean” prefix=”bean”%>

<h2>Fechas</h2>

<!-- Declaramos un objeto Date que hará las veces de bean -->
<jsp:useBean id=”fecha” class=”java.util.Date” />

<!-- Imprimimos el objeto -->
Fecha: <bean:write name=”fecha” /><br/>

<!-- Imprimimos propiedades del objeto (¡este código es caduco!) -->
Dia: <bean:write name=”fecha” property=”date” /><br/>
Mes: <bean:write name=”fecha” property=”month” /><br/>
A&ntilde;o: <bean:write name=”fecha” property=”year” /><br/>

<!-- Imprimimos la fecha, pero dándole un formato -->
Fecha con formato: <bean:write name=”fecha” format=”dd/MM/yyyy” /><br/>

<!-- Con formato, usando formatKey -->
Fecha con formatKey: <bean:write name=”fecha”
formatKey=”fecha.formato” /><br/>

<h2>N&uacute;meros</h2><br/>

<%
        /* Declaramos un objeto de tipo Double
y lo agregamos al scope de página */
```

```
    Double pi = Math.PI;
    pageContext.setAttribute("pi", pi, PageContext.PAGE_SCOPE);
%>

<!-- Imprimimos el número -->
PI: <bean:write name="pi" /><br/>

<!-- El número, formateado -->
PI con 4 decimales: <bean:write name="pi" format="##.####" /><br/>

<!-- Uso del atributo ignore, el siguiente tag no arrojará
una excepción pese a que el bean no existe -->
<bean:write name="inexistente" property="prop" ignore="true" />
```

Al acceder a la página, obtendremos el siguiente resultado, con fecha y hora actuales, de manera muy clara:

Figura 1. Si usamos el tag de Struts podemos dar formato a nuestros objetos sin necesidad de escribir código extra.

Como podemos observar, mostramos la fecha dos veces y con formatos distintos. En el primer caso, especificamos explícitamente el tipo de formato que queríamos, usando el atributo format .

Si esta página fuera accedida por gente de distintas partes del mundo, deberíamos mostrarles las fechas en los formatos que ellos mejor entiendan. Por ejemplo, en Norteamérica es común poner primero el mes y luego el día (o sea, un formato MM/dd/yyyy). Usando formatKey , estamos diciéndole a Struts que, dada una clave, busque el tipo de formato en el archivo de recursos (en nuestro caso, MessageResources.properties).

Como vimos, que podemos tener varias versiones del archivo de recursos, una para cada Locale que queramos soportar. Usando este enfoque, simplemente guardamos los distintos tipos de formatos en los archivos, y el usuario verá los objetos en sus respectivos formatos.

En el ejemplo, escribimos el atributo formatKey con valor fecha.formato . Veamos ahora unos ejemplos de archivos de recursos para esta clave:

MessageResources.properties (el archivo por defecto):

```
fecha.formato=dd/MM/yyyy
```

MessageResources_en_us.properties (el archivo para el Locale inglés de Estados Unidos):

```
fecha.formato=MM/dd/yyyy
```

MessageResources_ko.properties (el archivo para el Locale coreano):

```
fecha.formato=yyyy|MM|dd
```

Podemos tener todos los archivos de recursos que queramos. Siempre debe haber un archivo de recursos por defecto, sin especificación de Locale. Los otros archivos se crean con el mismo nombre que definimos en el struts-config agregando antes de la extensión el Locale, como se ve en los ejemplos.

Usando esta configuración, si un usuario con un navegador configurado con lenguaje coreano entra en nuestra aplicación, verá la fecha con el formato acorde; por ejemplo: 2002|02|02.

Pero no olvidemos que tener que especificar un formato o clave de formato cada vez que accedamos a objetos o propiedades de tipo fecha o numéricos es monótono y propenso a errores.

Es por eso que Struts define unas claves por defecto para usar en el archivo de propiedades según el tipo de objeto que estemos accediendo en el tag. De esta forma, si no especificamos los atributos format o formatKey y estamos accediendo a, por ejemplo, un objeto de tipo java.util.Date , Struts busca en el archivo de recursos una clave por defecto, y, si existe, le aplica el formato.

En la Tabla 1 tenemos la lista de claves por defecto y los objetos a los que se aplica.

CLAVES POR DEFECTO

CLAVE	TIPOS DE OBJETO
org.apache.struts.taglib.bean.format.sql.timestamp	java.sql.Timestamp
org.apache.struts.taglib.bean.format.sql.date	java.sql.Date
org.apache.struts.taglib.bean.format.sql.time	java.sql.Time
org.apache.struts.taglib.bean.format.date	java.util.Date
org.apache.struts.taglib.bean.format.int	java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.BigInteger
org.apache.struts.taglib.bean.format.float	java.lang.Float, java.lang.Double, java.lang.BigDecimal

Tabla 1. Lista de claves por defecto.

Un ejemplo de archivo de recursos podría ser:

```
org.apache.struts.taglib.bean.format.date=dd/MM/yyyy
org.apache.struts.taglib.bean.format.sql.date= dd/MM/yyyy
org.apache.struts.taglib.bean.format.sql.time=hh:mm:ss
org.apache.struts.taglib.bean.format.float=##.###
```

Y, de esta forma, no necesitamos explicitar que deseamos una salida con formato para estos tipos de objetos al usar el tag write .

bean:message

Se trata de un tag usado para obtener una cadena de texto del archivo de recursos. Esto es útil para mostrar texto internacionalizado, ya que no escribimos el texto en sí a mostrar, sino que declaramos qué texto ha de mostrarse y, luego, en base a las preferencias de idioma del navegador del usuario, se obtiene la cadena de texto correspondiente. Podemos obtener un mensaje especificando directamente la clave del mensaje, o indirectamente, usando un bean que contenga el valor de dicha clave.

Los mensajes, además, pueden tener hasta cinco argumentos parametrizables. Veamos los atributos del tag:

- arg0, arg1, arg2, arg3, arg4 : valor del argumento.
- key: clave del archivo de recursos que contiene el valor del mensaje que queremos mostrar.
- name : nombre del bean que contiene el valor o la propiedad de la clave del mensaje.
- property : la propiedad dentro del bean que contiene la clave del mensaje.
- scope : el scope donde buscar el bean.

Si no especificamos directamente el atributo key, debemos especificar un bean (y, opcionalmente, una propiedad del bean) adonde ir a buscar la clave.

```
<%@ taglib uri=”/tags/struts-bean” prefix=”bean”%>

<%
        /* Declaramos un objeto de tipo String y lo agregamos al scope de página */
    String clave =”saludo.inicial”;
    pageContext.setAttribute(“clave”, clave, PageContext.PAGE_SCOPE);
%>

<!-- Llamado directo -->
Saludo: <bean:message key=”saludo.inicial”/><br/>

<!-- Llamado directo con parámetro -->
Saludo con parametro: <bean:message key=”
saludo.parametrico” arg0=”Signore Rigoberto” /><br/>

<!-- Llamado directo con parámetro -->
Saludo con din&aacute;mico: <bean:message key=”saludo.parametrico”
arg0=”<%= Double.toString(Math.PI * 10) %>” /><br/>

<!-- Llamado indirecto -->
Saludo indirecto: <bean:message name=”clave” /><br/>
```

Y el archivo MessageResources.properties :

```
saludo.inicial=Hola!
saludo.parametrico=Hola {0}!
```

Como se ve en saludo.parametrico , las cadenas {0} {1} {2} {3}y {4}serán reemplazadas por el valor del argumento: arg0, arg1, etcétera.

En la página JSP, vemos ejemplos de llamados directos e indirectos.
El parámetro puede ser fijo o el resultado de una expresión cualquiera.

La Figura 2 muestra el resultado:

Figura 2. El tag bean:message es ideal para internacionalizar aplicaciones web y mostrar al usuario contenidos en su idioma.

bean:define

Mediante este tag podemos definir un nuevo bean. Este tag es similar a <jsp:useBean> pero difiere en varios aspectos. El primero se usa para crear objetos invocando a su constructor vacío. El caso de Struts, en cambio, se usa para introducir referencias a objetos sobre la base de valores constantes o expresiones y copiar beans o alguna de sus propiedades. Veamos los atributos que podemos definir en el bean y, luego, los infaltables ejemplos.

- id: el nombre del bean que se creará.
- name : nombre del bean al que accedemos para copiarlo (o alguna de sus propiedades).
- property : propiedad del bean especificado en name que queremos copiar en el bean a crear.
- scope : scope donde buscar el bean a copiar.
- toScope : scope donde guardar el bean que se crea (por defecto, page).
- type : nombre completo de la clase que será el tipo del bean creado.
- value : valor con el que el nuevo bean será instanciado.

Fundamentalmente tenemos dos formas de crear un nuevo bean: pasando un valor (constante o una expresión) al atributo value o como cuerpo del tag –lo cual creará un bean de tipo string con el contenido que le pasamos–

o definiendo otro bean y, eventualmente, una propiedad de este, y el bean a crear será una referencia al bean (o propiedad). Estas dos formas son excluyentes, y, si definimos el atributo value y name a la vez, obtendremos una excepción. De todas formas, si definimos el atributo value y, además, el tag tiene cuerpo (ya que estamos creando un bean string), estaremos brindando dos veces el contenido. Pasemos ahora a los ejemplos:

```
<%@ taglib uri=”/tags/struts-bean” prefix=”bean”%>

<!-- Creamos un GregorianCalendar -->
<jsp:useBean id=”cal” class=”java.util.GregorianCalendar” />

<!-- Copiamos el bean cal en otro bean y definimos su tipo -->
<bean:define id=”cal2” name=”cal” type=”java.util.Calendar”/>

<!-- Creamos un bean en base a una propiedad de otro bean -->
<bean:define id=”tz” name=”cal” property=”timeZone” />

<!-- Diferentes formas de crear un bean de tipo String -->
<bean:define id=”str1” value=”Nuevo bean” toScope=”request” />

<bean:define id=”str2” value=”<%= “hola pianola”%>” />

<bean:define id=”str3”>Valor del tercer String,
 en el cuerpo del tag</bean:define>

<!-- Mostramos los beans o propiedades de los beans recientemente creados -->
Timezone: <bean:write name=”tz” property=”displayName” /><br/>
String 1: <bean:write name=”str1” /><br/>
String 2: <bean:write name=”str2” /><br/>
String 3: <bean:write name=”str3” /><br/>
```

bean:size

Este sencillo tag crea un bean con la cantidad de elementos de una colección dada. Si bien veremos luego tags para iterar sobre colecciones, sin este tag no habría forma (sin recurrir a código en el JSP) de obtener los

elementos de una colección. Su modo de uso es sencillo y fácil de entender.
Veamos los atributos:

- collection : expresión que evalúa a una colección.
- id: el nombre del bean de tipo Integer que se creará.
- name : nombre del bean que contiene la colección a ser evaluada.
- property : nombre de la propiedad (dentro del bean de nombre name) que devuelve la colección.
- scope : el scope donde buscar el bean de nombre name .

Nuevamente, tenemos dos formas: o bien usamos el atributo collection o bien definimos un bean o propiedad con los atributos name, property y scope :

```
<%@ taglib uri=”/tags/struts-bean” prefix=”bean”%>

<!-- Creamos una coleccion y la guardamos en el scope de página -->
<%
        java.util.Vector v = new java.util.Vector();
        v.addElement(“s”);
        v.addElement(“t”);
        v.addElement(“z”);
        pageContext.setAttribute(“v”, v, PageContext.PAGE_SCOPE);
%>

<!-- Diferentes formas de acceder al tamaño de la colección -->
<bean:size id=”s” collection=”<%= v %>” />
<bean:size id=”s2” name=”v” />
```

Es muy importante que recordemos que escribir código Java en las páginas JSP no es recomendable, y que, en estos casos, lo estamos haciendo solamente para mantener los ejemplos pequeños y focalizar en lo que queremos ver, que son los tags.

Logic tags

Este paquete contiene tags para acciones lógicas: condiciones, iteraciones, etcétera. La idea aquí es, nuevamente, no embeber código Java en las páginas sino usar los tags de Struts para no mezclar la presentación con el contenido.

logic:equal, logic:greaterEqual, logic:greaterThan, logic:lessEqual, logic:lessThan, logic:notEqual

Estos tags son tags de comparaciones . En todos los casos, sus atributos y funcionamiento son iguales, y solo cambia el tipo de comparación. La lógica de estos tags es proveer dos valores que luego serán comparados.
Si la evaluación es satisfactoria, se ejecuta el cuerpo del tag; caso contrario, se omite. Veamos primero los atributos posibles, para luego pasar a la explicación de su funcionamiento:

- value : primer valor a ser comparado (constante o expresión).
- name , property , scope : nombre, propiedad y scope del bean con el dato a ser comparado.
- cookie : nombre de la cookie con el valor a ser comparado.
- parameter : nombre del parámetro con el valor a ser comparado.
- header : nombre del encabezado con el valor a ser comparado.

Debemos proporcionar dos valores: uno es una constante o resultado de una expresión y se especifica con el atributo value . El otro valor puede ser especificado mediante un bean o una propiedad, el valor de una cookie, de un encabezado o un parámetro del pedido. Debemos especificar solo uno de estos atributos u obtendremos una excepción.

Notemos que los valores pasados siempre serán cadenas de texto. Al hacer la comparación, primeramente se evalúa si los valores pueden ser convertidos a double o long . De ser así, la comparación será numérica; caso contrario, se realizará una comparación de cadenas de caracteres. Si alguno de los valores es nulo, será reemplazado por una cadena vacía.

Veamos algunos ejemplos de uso:

```
<%-- Declaramos que vamos a usar la librería de tags
 logic y que la usaremos con el prefijo logic --%>
<%@ taglib uri=”/tags/struts-logic” prefix=”logic”%>
<%@ taglib uri=”/tags/struts-bean” prefix=”bean”%>

<%-- Definimos dos variables --%>
<bean:define id=”dos” value=”2” />
<bean:define id=”cadena” value=”java” />

<%-- Comparación por igualdad, numérica --%>
<logic:equal name=”dos” value=”2”>
        2 = 2
        <br/>
</logic:equal>

<%-- Comparación por desigualdad, numérica --%>
<logic:notEqual name=”dos” value=”<%= Double.toString(Math.E) %>”>
        2 != e
        <br/>
</logic:notEqual>

<%-- Comparación por menor, numérica --%>
<logic:lessThan name=”dos” value=”<%= Double.toString(Math.E) %>”>
        2 < e
        <br/>
</logic:lessThan>

<%-- Comparación por mayor, texto --%>
<logic:greaterThan name=”cadena” value=”a”>
        java > a
        <br/>
</logic:greaterThan>

<%-- Comparación por mayor, texto --%>
<logic:greaterThan name=”cadena” value=”2”>
        java > 2
        <br/>
</logic:greaterThan>
```

logic:empty, logic:notEmpty

Con estos tags podemos evaluar una variable para comprobar si tiene algún valor. Los atributos son los clásicos name , property y scope que ya tanto conocemos y usamos para determinar un objeto sobre la base de un bean o una propiedad de este. Según el objeto que determinemos usando estos atributos, la evaluación determinará si su contenido es vacío o no (empty y notEmpty , respectivamente).

El objeto será vacío si es nulo, si es una cadena de caracteres de tamaño cero o si es una colección, mapa o arreglo sin elementos. De manera análoga podremos constatar su condición de no vacío .

logic:iterate

Mediante este tag podemos iterar sobre una colección y repetir el contenido del cuerpo del tag tantas veces como elementos tenga la colección. Los atributos del tag para definir la iteración son:

- collection : expresión que evalúa a una colección.
- name , property , scope : atributos para determinar la collection (ya son amigos, no necesitan más explicación).
- id: nombre que le daremos al elemento actual de iteración.
- type : nombre de clase del elemento actual de iteración (el definido por el parámetro id). Si no especificamos este atributo, no se realizarán conversiones de tipo.
- indexId : nombre de la variable que contiene el índice de iteración actual.
- length : máximo número de iteraciones que haremos sobre esta colección. Puede ser un número directo o referencia a un bean de tipo Integer .
- offset : índice de inicio de iteración.

Tenemos atributos para determinar el objeto a iterar y otros propios de la iteración. Pueden parecer confusos, pero son similares a una iteración común de Java. Veamos una iteración sobre una colección usando un ciclo for , y comprobaremos la analogía con los atributos de Struts.

```
// La colección
Collection col = ...

// Iteramos
for (int indexId=offset; indexId < length; indexId++) {
        [type] id = col.elementAt(indexId)
        ...
}
```

Este pseudocódigo muestra la función de los atributos del tag iterate . Los atributos offset y length son usados para paginación, cuando tenemos una colección muy grande y no queremos que el usuario reciba una página enorme de respuesta. En ese caso, usamos variables para ir mostrando páginas acotadas con algunos resultados. No pueden faltar los ejemplos clarificadores:

```
<%-- Importamos clases --%>
<%@ page import="java.util.*, java.net.URL"%>

<%-- Declaramos que vamos a usar la librería de tags
logic y que la usaremos con el prefijo logic --%>
<%@ taglib uri="/tags/struts-logic" prefix="logic"%>
<%@ taglib uri="/tags/struts-bean" prefix="bean"%>

<%
        // Una colección de direcciones web
        Collection urls = new Vector();
        urls.add(new URL("http://www.google.com"));
        urls.add(new URL("http://onweb.tectimes.com"));
        urls.add(new URL("http://java.sun.com"));
        urls.add(new URL("http://struts.apache.org"));
        urls.add(new URL( "http://java.sun.com/products/jsp/"));
        pageContext.setAttribute("colUrls", urls, PageContext.PAGE_SCOPE);
%>
```

```
<%-- Iteramos y mostramos todas las direcciones web --%>
<h3>Toda la colecci&oacute;n:</h3>
<logic:iterate id="url" name="colUrls">
        <bean:write name="url"/><br/>
</logic:iterate>

<%-- Iteramos y mostramos la propiedad host de las primeros
dos elementos de la colección --%>
<h3>Los primeros dos hosts:</h3>
<logic:iterate id="url" name="colUrls" length="2">
        <bean:write name="url" property="host" /><br/>
</logic:iterate>

<%-- Iteramos y mostramos la propiedad protocol de las primeros
dos elementos de la colección y su posición --%>
<h3>Las direcciones 3 y 4:</h3>
<logic:iterate indexId="i" id="url" name="colUrls" offset="2" length="2">
        <bean:write name="url" />
        (posici&oacute;n <bean:write name="i" />)
        <br/>
</logic:iterate>
```

Notaremos que, en el último caso, estamos accediendo al tercer y cuarto elemento de la colección y, sin embargo, al imprimir su posición usando el bean indexId, nos devuelve 2 y 3 respectivamente. La explicación queda como ejercicio para el lector (alcanza con recordar que el índice empieza en cero).

Html tags

Este paquete de tags se utiliza para crear páginas HTML con más facilidad y, sobre todo, que puedan interactuar con componentes de Struts (en especial, formularios) para ahorrarnos trabajo. Muchos de estos tags tienen una gran cantidad de atributos para ser compatibles con los tags estándar de HTML (atributos para métodos JavaScript, estilos CSS, entre otros).

html:html

Este tag es sencillo de usar y provee una funcionalidad muy poderosa. Su función es generar un tag <html> . Sus atributos:

- lang : si su valor es verdadero (true), se generará el atributo lang , con el valor del idioma y país (el Locale) definido en la sesión, o el brindado por el navegador o el Locale por defecto (en ese orden de prioridades). Este atributo es útil para el navegador para poder mostrar el contenido apropiadamente o para los motores buscadores que puedan catalogar el sitio con más precisión.
- xhtml : estableciendo este valor como verdadero, todos los tags del paquete HTML generarán código compatible con XHTML.

html:form

Mediante este tag podemos generar un tag <form> de HTML. Todos los componentes del formulario deben estar encerrados dentro de un tag form. Con los atributos del formulario podremos aplicar funcionalidad a todos sus componentes. Vamos a ver sus atributos:

- action : la dirección adonde es enviado este form. Este valor debe existir como atributo path en alguno de los elementos action del struts-config .
- acceptCharset : una lista de los posibles charsets (conjuntos de caracteres) que este formulario aceptará.
- disabled : si es true , deshabilita todos los campos de este formulario.
- enctype : la codificación del formulario cuando sea enviado. Generalmente no debemos modificar este valor, a menos que queramos incluir funcionalidad para que el usuario envíe archivos, en cuyo caso debemos especificar este atributo con valor multipart/form-data .
- focus : nombre del campo perteneciente a este form al que queremos darle foco cuando la página se cargue.
- focusIndex : si especificamos un campo en el atributo focus y este campo contiene múltiples valores (una lista de opciones, por ejemplo), podemos

establecer el elemento específico dentro del campo al que queramos dar foco mediante este atributo.

- method : el método HTTP que deseemos usar para enviar el formulario. Podemos especificar GET o POST (por defecto, este último).
- onreset : código JavaScript a ejecutarse cuando el form es limpiado.
- onsubmit : código JavaScript a ejecutarse cuando el form es enviado.
- readonly : si es true , establece que todos los campos del formulario sean solo de lectura, inmodificables por el usuario.
- style : estilos CSS a aplicar al elemento.
- styleClass : clase CSS a aplicar al elemento.
- styleId : identificador a asignar al elemento.
- target : ventana destino adonde el formulario es enviado. Útil para aplicaciones que usan frames.

A medida que vayamos viendo campos de formularios, veremos también ejemplos de configuraciones y usos de este tag.

html:text

Este tag genera un campo de texto (en HTML un elemento <input type="text">). Este tag contiene una gran cantidad de atributos posibles. Nos concentraremos en algunos y otros simplemente los enunciaremos. Veamos algunos de ellos:

- disabled : si es true , deshabilita este campo.
- style : estilos CSS a aplicar al elemento.
- styleClass: clase CSS a aplicar al elemento.
- errorStyle : estilo CSS a aplicar al campo si un error existe para la propiedad que define este campo.
- errorStyleClass : clase CSS a aplicar al campo si un error existe para la propiedad que define este campo.
- maxlength : número máximo de caracteres a aceptar en el campo.
- property : propiedad del form que se populará al enviar el formulario.

- size: tamaño del campo (en caracteres).
- value : valor inicial del campo.

El tag soporta, además, una gran cantidad de atributos compatibles con acciones JavaScript (onclick, onfocus, onmousedown , etcétera). Algunos ejemplos:

```
<%-- Declaramos el uso del paquete de tags HTML --%>
<%@ taglib uri=”/tags/struts-html” prefix=”html”%>

<html:html xhtml=”true”>

<%-- Foco en el campo edad --%>
<html:form action=”procesarFormulario” focus=”edad”>

<%-- Un campo con estilo que cambia al haber un error --%>
Nombre:
<html:text property=”nombre”
        errorStyle=”font-family:Verdana;font-size:12px;”
        style=”font-family:Verdana;font-size:12px;
background-color:#FFFF00;color:#FF0000” />
<br/>

<%-- Un campo acotado en tamaño --%>
Edad: <html:text property=”edad” size=”3” maxlength=”3”/>
<br/>

<%-- Campo deshabilitado --%>
Campo deshabilitado: <html:text property=”deshabilitado” disabled=”true” />
<br/>

<%-- Campo de solo lectura y con valor por defecto --%>
Campo de s&oacute;lo lectura: <html:text property=”readonly”
readonly=”true” value=”No soy modificable” />
<br/>

</html:form>

</html:html>
```

html:password

Casi igual a html:text , este tag genera un campo propicio para el ingreso de contraseñas y datos confidenciales. Comparte todos los atributos con html:text y agrega uno más:

- redisplay: por defecto, cuando una página con valores es mostrada, ya sea por valores de inicialización, por la vuelta a la página luego de una validación fallida, etcétera, los valores de los campos son completados automáticamente. En el caso de un campo de contraseña, esto no es conveniente, porque, por más que aparezcan asteriscos, el valor de la contraseña será visible si accedemos al código de la página HTML. Si establecemos este atributo como false, el campo no se repopulará automáticamente.

html:textarea

En esencia, similar a html:text , solo que este tag genera un campo de texto de más de una línea. Tiene prácticamente los mismos atributos de html:text , a excepción de los atributos maxlength y size. Este último es reemplazado por:

- rows: número de filas del campo.
- cols : número de columnas del campo.

html:select

Este tag genera una lista de valores que puede ser múltiple o simple, permitiendo al usuario seleccionar muchos o solo un valor de la lista.
Veamos sus atributos:

- disabled , style , styleClass , errorStyle , errorStyleClass , property , value : comparte las mismas funciones que html:text .
- multiple : si es true , el usuario puede elegir varios elementos de la lista; si no, solo uno.
- size : cantidad de elementos a mostrar a la vez.

Este tag genera el elemento HTML <select> , pero todavía tenemos que generar las opciones (los elementos <option>). Para ello, podemos usar cualquier combinación de los tags html:option , html:options y html:optionsCollection , solamente válidos dentro del cuerpo de un html:select .

html:option

Genera una opción dentro de una lista. Para esta opción debemos especificar dos valores: el que se ha de enviar si esta opción es seleccionada y el que se muestra al usuario.

- disabled, style, styleId, styleClass : misma funcionalidad que la descripta antes.
- key: si especificamos este atributo, determina la clave dentro del archivo de recursos que se usará para el valor a mostrar para esta opción; caso contrario, dicho valor se obtendrá del cuerpo del tag.
- value : el valor a enviar si esta opción es la elegida.

html:options

Genera un conjunto de opciones, basándose en una colección. Este tag tiene dos formas de funcionamiento distintas en base a si el atributo collection está seteado o no. Vamos a ver solamente un funcionamiento, ya que ambos tornarán confuso el asunto y, además, para el segundo caso, el tag html:optionsCollection es más simple de usar.

Veamos los atributos y su uso:

- style , styleClass : estilo y clase CSS del elemento.
- collection : asumiremos que este atributo no se especifica.
- filter : por defecto, filtra los caracteres sensitivos HTML. Si no queremos que esto pase, debemos establecerle valor false .
- name, property : determinan un bean conteniendo una colección de valores.
- labelName , labelProperty : determinan un bean que contiene una colección de etiquetas.

Una explicación es necesaria sobre el funcionamiento de este tag: recordemos que cada opción tiene un valor, que es el enviado cuando la opción es elegida, y una etiqueta, que es la que se muestra al usuario en la lista.
Este tag genera un conjunto de opciones y toma los valores y las etiquetas de dos colecciones separadas. La primera opción será el valor del primer elemento de la colección de valores y la etiqueta del primer elemento de la colección de etiquetas, y así sucesivamente. Con los atributos name y property , estamos determinando la colección de valores (property es opcional), y con labelName y labelProperty determinamos la colección de etiquetas. En este caso, ambos atributos son opcionales. Si especificamos solo labelProperty , la colección de etiquetas será buscada como una propiedad del form. Si no especificamos ninguno de los dos, la etiqueta será el valor.

> LOS HTML TAGS SON PAQUETES DE TAGS PARA CREAR PÁGINAS HTML CON MÁS FACILIDAD

¿Qué pasa si las dos colecciones tienen distinto número de elementos?

Si tenemos más elementos en la colección de etiquetas, estos simplemente serán ignorados. Si tenemos más elementos en la colección de valores, estos serán agregados a la lista, y su etiqueta será el mismo valor.

html:optionsCollection

Este tag genera también una lista de opciones. A diferencia de html:options , este tag funciona con una única colección de beans donde cada uno contiene el valor y, opcionalmente, la etiqueta. Veamos los atributos antes de la explicación pertinente:

- filter , style , styleClass : poseen la misma funcionalidad que html:options .
- name , property : determinan la colección de beans que usaremos para generar las opciones.
- label : propiedad del bean que devuelve la etiqueta.
- value : propiedad del bean que devuelve el valor.

Veamos un ejemplo integrador. Es importante destacar que, dentro de un elemento html:select , podemos incluir y mezclar cualquier cantidad de estos tags de opciones, y en cualquier orden. En el ejemplo, creamos un elemento select y lo populamos con los tres tags posibles.

Un simple bean que creamos ad hoc :

```
package capitulo7.model;

public class Pais {
    private String codigo;

    private String nombre;

    public Pais(String codigo, String nombre) {
        this.codigo = codigo;
        this.nombre = nombre;
    }

    public String getCodigo() {
        return codigo;
    }

    public String getNombre() {
        return nombre;
    }
}
```

ONWEB: EJEMPLO INTEGRADOR

Enredusers.com/premium podemos acceder a un ejemplo integrador de todo lo visto en este capítulo. Después de tantos tags, atributos, configuraciones y modos de uso, es menester mostrar un ejemplo que use todos los tags y ver cómo acceder a los datos desde el servidor.

Ahora la página JSP que muestra la lista y sus opciones:

```
<%-- Importamos clases --%>
<%@ page import="java.util.*, capitulo7.model.Pais"%>

<%@ taglib uri="/tags/struts-html" prefix="html"%>

<%
        // Una colección de países
        Collection paises = new Vector();
        paises.add(new Pais("bo", "Bolivia"));
        paises.add(new Pais("co", "Colombia"));
        paises.add(new Pais("cl", "Chile"));
        paises.add(new Pais("pe", "Peru"));
        paises.add(new Pais("py", "Paraguay"));
        paises.add(new Pais("ve", "Venezuela"));
        paises.add(new Pais("uy", "Uruguay"));
        pageContext.setAttribute("paises", paises, PageContext.PAGE_SCOPE);

        // Una colección de códigos
        Collection codigos = new Vector();
        codigos.add("ec");
        codigos.add("br");
        codigos.add("mx");
        codigos.add("bb");  /* Este código será etiqueta también */

        pageContext.setAttribute("codigos", codigos, PageContext.PAGE_SCOPE);

        // Una colección de nombres
        Collection nombres = new Vector();
        nombres.add("Ecuador");
        nombres.add("Brasil");
        nombres.add("Mexico");
        pageContext.setAttribute("nombres", nombres,
PageContext.PAGE_SCOPE);
%>
```

```
<html:html xhtml=”true”>

<html:form action=”procesarFormulario”>

<%-- Una lista de elementos, mostramos 4 ítems por
vez y es una lista de selección simple --%>
<html:select property=”pepe” size=”4” multiple=”false”>

        <%-- Primera opción, fija --%>
        <html:option value=”ar”>Argentina</html:option>

        <%-- Accedemos a distintas colecciones
para obtener valores y etiquetas --%>
        <html:options name=”codigos” labelName=”nombres”/>

        <%-- Accedemos a una colección de beans y especificamos las
propiedades para obtener el valor y la etiqueta de cada elemento --%>
        <html:optionsCollection name=”paises” label=”nombre” value=”codigo” />
</html:select>

</html:form>

</html:html>
```

html:radio

Con este tag generamos un botón de exclusión mutua, también conocido como radio button . Al igual que en la lista, el usuario elije una y solo una opción (de ahí su nombre, exclusión mutua). Si hay una opción ya elegida y el usuario marca otra, se desmarcará la primera y se marcará la segunda, obligando a elegir solo un valor. Veamos los atributos fundamentales del tag:

- property : propiedad del form correspondiente a este componente.
- idName : nombre del bean que contiene el valor que ha de devolver este componente. Por lo general, se usa en conjunción con un iterador.

- value: valor constante a devolver o, si existe el atributo idName, nombre de la propiedad del bean determinado por idName que contiene el valor.

Vemos que este tag tiene dos formas de funcionamiento, dependiendo de si especificamos o no el atributo idName. Si no especificamos tal atributo, en value definimos el valor que se enviará al servidor si el botón actual es el elegido por el usuario. Puede ser una constante o el resultado de una expresión.

Caso contrario, idName determina un bean que contiene el valor a usar y podemos opcionalmente especificar, con el atributo value, la propiedad de dicho bean que devolverá el valor. Una vez más, es preciso un ejemplo:

```
<%-- Importamos clases --%>
<%@ page import="java.util.*, capitulo7.model.Pais"%>

<%-- En este ejemplo, usamos tres paquetes de tags --%>
<%@ taglib uri="/tags/struts-html" prefix="html"%>
<%@ taglib uri="/tags/struts-bean" prefix="bean"%>
<%@ taglib uri="/tags/struts-logic" prefix="logic"%>

<%
        // Una colección de países
        Collection paises = new Vector();
        paises.add(new Pais("bo", "Bolivia"));
        paises.add(new Pais("co", "Colombia"));
        paises.add(new Pais("cl", "Chile"));
        paises.add(new Pais("pe", "Peru"));
        paises.add(new Pais("py", "Paraguay"));
        paises.add(new Pais("ve", "Venezuela"));
        paises.add(new Pais("uy", "Uruguay"));
        pageContext.setAttribute("paises", paises, PageContext.PAGE_SCOPE);
%>

<html:html xhtml="true">

<%-- Le pasamos el foco al tercer componente del grupo de botones pais --%>
```

```
<html:form action=”procesarFormulario” focus=”pais” focusIndex=”2”>

Sexo: <br/>

<%-- Dos radios con valor constante --%>
<html:radio property=”sexo” value=”M” />Masculino<br/>
<html:radio property=”sexo” value=”F” />Femenino<br/>

<br/>

Pa&iacute;s:<br/>

<%-- Iteramos sobre el conjunto de países para generar los radios --%>
<logic:iterate id=”itPais” name=”paises”>

<%-- Cada radio obtiene su valor de la propiedad codigo
del bean itPais, el bean actual de la iteración --%>
        <html:radio property=”pais” idName=”itPais” value=”codigo” />

<%-- Escribimos el nombre del país para el usuario --%>
        <bean:write name=”itPais” property=”nombre” />
        <br/>
</logic:iterate>

</html:form>

</html:html>
```

LISTA CON PRIMERA OPCIÓN

Una buena idea es agregar una primera opción fija en las listas de selección que contenga una etiqueta del estilo ELIJA OPCIÓN y valor nulo para facilitar su validación. Usando html:option podremos agregar este valor fácilmente a la lista de valores.

Este código en acción, en la Figura 3 .

Figura 3. Los botones radio cumplen la misma función que las listas de selección simple, pero son más claros cuando hay pocas opciones.

En el primer caso, simplemente creamos dos opciones con un valor fijo. En el segundo caso, más complicado, nos valemos de un iterador para crear tantos botones radio como elementos tengamos en una colección dada, y especificamos los atributos idName y value para obtener el valor de una propiedad del objeto iterado. Notemos cómo en este sencillo ejemplo hemos usado tags del paquete bean , html y logic .

html:checkbox

Este tag sirve para generar un checkbox (‘casilla de verificación’). Cada uno de estos componentes, a diferencia del botón radio, puede tomar dos valores

LOGIC:ELSE

El paquete logic no tiene ningún tag para manejar una condición que falló, como sería else en Java. Debemos crear dos tags condicionales; el segundo, con la condición complementaria.

(marcado o desmarcado, implicando verdadero y falso respectivamente), independientemente de los otros componentes checkbox que existan en la página. En el Capítulo 5 , en uno de los ejemplos que programamos, usamos un checkbox que el usuario podía marcar según su condición de fumador.
Los atributos principales del tag son muy sencillos:

- property : propiedad del form a enviar.
- value : valor a enviar si el checkbox es marcado. Este atributo, a diferencia de casi todos los otros tags, es opcional; si no se especifica, el valor por defecto es on .
-

Antes de pasar a un ejemplo, veamos primero otro tag.

html:multibox

Si tenemos una larga lista de opciones que el usuario puede marcar, tendremos que crear una propiedad en el form-bean para cada una de ellas y, luego, en la acción, preguntar una por una por su estado.
Si la lista es larga, este proceso es tedioso. Más importante aún: si no sabemos a priori la cantidad de elementos que tiene la lista, no podremos realizar esta funcionalidad. El tag html:multibox nos sirve en estas ocasiones. Este tag genera el mismo código HTML que html:checkbox , pero su forma de obtener los elementos y enviarlos al servidor es distinta. El tag html:checkbox genera un checkbox y está asociado a una propiedad.

OTROS TAGS HTML

Struts tiene varios tags en el paquete HTML; no hemos visto aquí todos los atributos posibles, sino solo los más significativos. En la dirección http://struts.apache.org/struts-doc-1.2.8/userGuide/struts-html.html hay documentación sobre los restantes tags.

Cuando el formulario se envía, esa propiedad tendrá el valor especificado en el atributo value o el valor por defecto on. Con html:multibox se envía al servidor un arreglo (por lo general, de objetos de tipo string) con los valores seleccionados. Veamos la forma de utilizar el tag y, luego, un ejemplo.

- property : propiedad del form a enviar.
- value : valor a enviar si el checkbox es marcado. Si este atributo no se especifica, el valor es tomado del cuerpo del tag.

Vamos a aprovechar y ver un ejemplo completo, donde usemos estos dos últimos tags. Definimos un form en el archivo struts-config.xml :

```
<form-bean
   name="checkboxForm"
type="org.apache.struts.validator.LazyValidatorForm">
   <form-property name="paisesVisitados" type="java.lang.String[]" />
</form-bean>
```

Notemos que, si bien el tipo del form-bean es LazyValidatorForm y a priori no necesitamos definir sus propiedades, sí es necesario en este caso explicitar que paisesVisitados es un arreglo de tipo string. Si no lo hiciéramos, al enviar el formulario con varios ítems seleccionados, el form lo tomaría como varios valores de la misma propiedad y sobrescribiría el valor anterior, en definitiva, obteniendo a lo sumo un único valor seleccionado, que, claramente, no es lo que necesitamos.

Definimos dos acciones: la primera proveerá los datos a la página para mostrar el formulario, y la segunda procesará el formulario enviado.

```
<action path="/verCheckboxForm"
                  type="capitulo7.action.VerCheckboxAction"
                  name="checkboxForm" scope="request">
      <forward name="ok" path="/jsp/capitulo7/html_checkbox.jsp" />
</action>
```

```
<action path=”/procesarCheckboxForm”
type=”capitulo7.action.ProcesarCheckboxAction”
                    name=”checkboxForm” scope=”request”/>
```

Veamos, ahora, la clase que implementa la primera acción:

```
package capitulo7.action;

import java.util.Collection;
import java.util.Vector;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.validator.LazyValidatorForm;

import capitulo7.model.Pais;

public class VerCheckboxAction extends Action {

    public ActionForward execute(ActionMapping map, ActionForm form,
            HttpServletRequest req, HttpServletResponse res) throws Exception {

        // Cast al tipo específico de objeto
        LazyValidatorForm dyna = (LazyValidatorForm) form;

        // Una colección de países
        Collection paises = new Vector();
```

```
        paises.add(new Pais("bo", "Bolivia"));
        paises.add(new Pais("co", "Colombia"));
        paises.add(new Pais("cl", "Chile"));
        paises.add(new Pais("pe", "Peru"));
        paises.add(new Pais("py", "Paraguay"));
        paises.add(new Pais("ve", "Venezuela"));
        paises.add(new Pais("uy", "Uruguay"));

        // Agregamos los países al form
        dyna.set("paises", paises);

        // Queremos ir a la vista de nombre "ok"
        return map.findForward("ok");
    }
}
```

Vemos que simplemente se trata de crear una colección de objetos de tipo Pais y agregarla al form. ¡Finalmente, estamos haciendo esta parte bien! No más inserciones en las páginas JSP.

La página del formulario:

```
<%@ taglib uri="/tags/struts-html" prefix="html"%>
<%@ taglib uri="/tags/struts-bean" prefix="bean"%>
<%@ taglib uri="/tags/struts-logic" prefix="logic"%>

<html:html>

<%-- Le pasamos el foco al tercer componente del grupo de botones pais --%>
<html:form action="procesarCheckboxForm">

Condimentos: <br/>

<%-- Objetos checkbox que envían un valor que definimos nosotros --%>
```

```
<html:checkbox property=”aceite” value=”si” />Aceite<br/>
<html:checkbox property=”vinagre” value=”si” />Vinagre<br/>
<html:checkbox property=”sal” value=”si” />Sal<br/>

Pa&iacute;ses visitados:<br/>

<%-- Iteramos sobre el conjunto de países para generar los otros checkbox --%>
<logic:iterate id=”itPais” name=”checkboxForm” property=”paises”>
        <html:multibox property=”paisesVisitados”>

<%-- El cuerpo del tag es el valor que enviará --%>
                <bean:write name=”itPais” property=”codigo” />
        </html:multibox>

        <%-- Escribimos el nombre del país para el usuario --%>
        <bean:write name=”itPais” property=”nombre” />
        <br/>
</logic:iterate>

<br/>
<html:submit value=”Enviar” />

</html:form>

</html:html>
```

FORM COMO BEAN

Notemos que, en el ejemplo, la lista de opciones está guardada en el formulario, por lo que debemos especificar el nombre del formulario como objeto bean (mediante el atributo `name`) y, además, una propiedad, que es la que nos devuelve la colección sobre la que iteraremos.

El primer grupo de checkboxes tiene tres elementos fijos; el segundo es una lista de la que, en principio, no sabemos qué cantidad de elementos tiene.

Figura 4. Distintos tags generan el mismo resultado.

Por último, veamos un bosquejo de la acción que procesa el formulario una vez enviado:

```
package capitulo7.action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.validator.LazyValidatorForm;

public class ProcesarCheckboxAction extends Action {
```

```
public ActionForward execute(ActionMapping map, ActionForm form,
        HttpServletRequest req, HttpServletResponse res) throws Exception {

    LazyValidatorForm dyna = (LazyValidatorForm) form;

    // Obtenemos los resultados de los checkbox
    boolean aceite = "si".equals(dyna.get("aceite"));
    boolean vinagre = "si".equals(dyna.get("vinagre"));
    boolean sal = "si".equals(dyna.get("sal"));

    // Obtenemos los países
    String[] paisesVisitados = (String[]) dyna.get("paisesVisitados");

    return null;
  }
}
```

Esta acción es solo un esqueleto para que veamos las diferentes formas de acceder a los valores, tanto de los checkboxes como de los multiboxes (que, en definitiva, también son checkboxes).

CONJUNTOS DE CARACTERES

Si establecemos conjuntos de caracteres válidos impedimos, por ejemplo, que un usuario ingrese datos que contengan caracteres incompatibles con nuestra base de datos y que, finalmente, terminarán mostrándose incorrectamente en los navegadores. La organización IANA mantiene la lista de todos los conjuntos de caracteres y su descripción en: www.iana.org/assignments/character-sets .

html:submit

Antes de terminar el capítulo, un tag sencillo pero fundamental en la creación de formularios: html:submit . Este tag genera un botón de envío de formulario. Cuando este botón es presionado, el formulario y sus datos serán enviados a la dirección especificada por el atributo action del tag html:form . El uso de este tag es muy sencillo, pero debemos recordar siempre agregar una forma de enviar los datos en cada formulario. El único atributo significativo es value , que determina la etiqueta a aplicar al botón. En el ejemplo de checkboxes usamos este tag.

RESUMEN

Este capítulo fue largo pero fructífero, ya que analizamos un aspecto fundamental del desarrollo web: la vista. Usando JSP junto con tags propios de Struts, vimos cómo crear formularios fácilmente, sobre la base de colecciones y atributos de aplicación, y cómo usar los tags para evitar la necesidad de embeber código Java.

Actividades

TEST DE AUTOEVALUACIÓN

1 ¿Qué diferencia existe entre los tres tags que generan opciones para la lista (html:option , html:options y html:optionsCollection)?

2 ¿Por qué es necesario contar con el tag html:multibox cuando ya existe html:checkbox ?

EJERCICIOS PRÁCTICOS

1 Pruebe el resultado de no definir el arreglo en el ejemplo de checkboxes y vea qué obtiene cuando elige más de un checkbox.

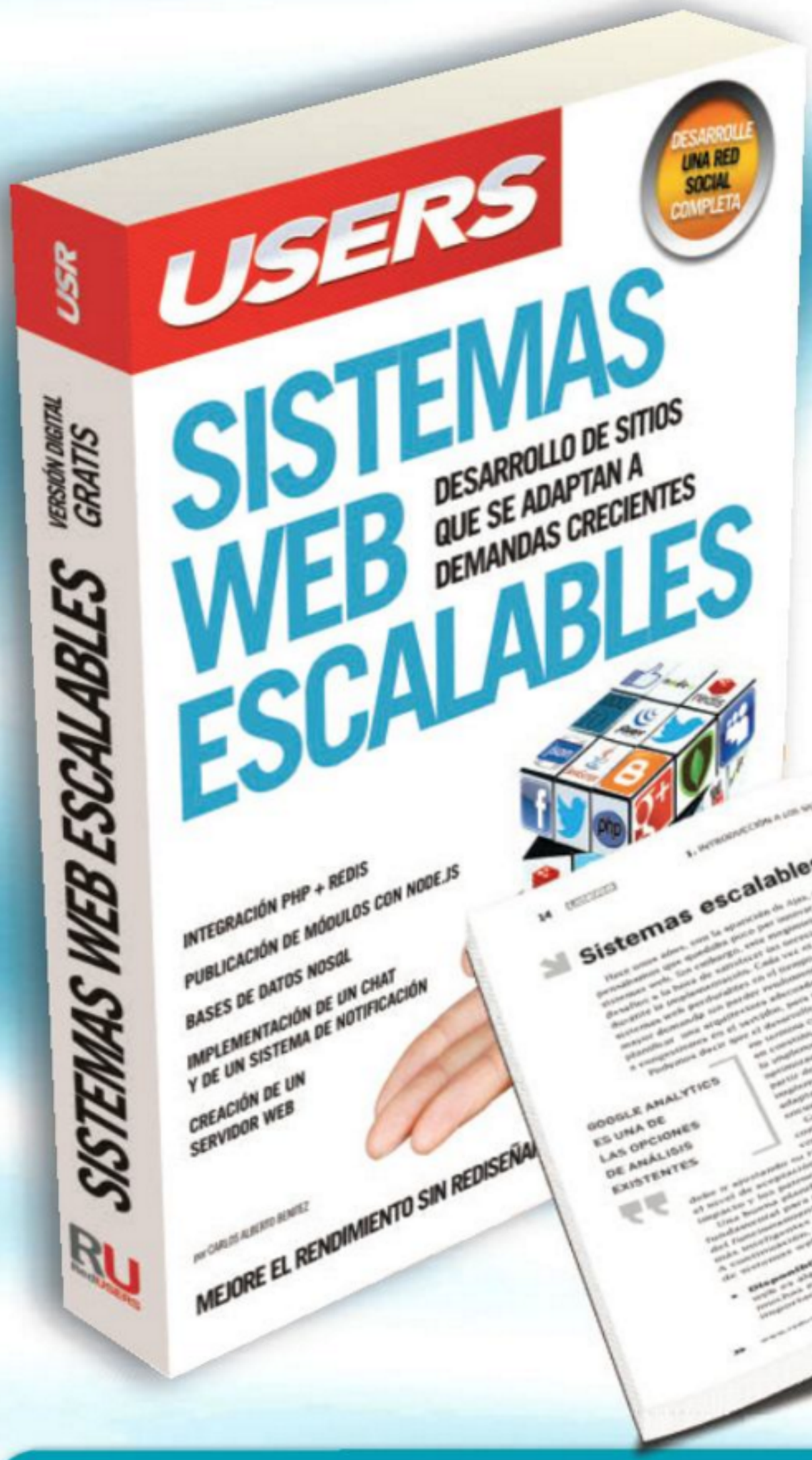
2 Complete el ejemplo de checkbox haciendo algo con la acción y creando una página de salida.

3 Traduzca el juego del Ahorcado para que funcione con Struts.

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: profesor@redusers.com .

CONÉCTESE CON LOS MEJORES LIBROS DE COMPUTACIÓN



Cree su propia red social e implemente un sistema capaz de evolucionar en el tiempo y responder al crecimiento del tráfico.

>>DESARROLLO / INTERNET
>>320 PÁGINAS
>>ISBN 9978-987-1949-20-5

LLEGAMOS A TODO EL MUNDO VÍA OCA* Y DHL**

MÁS INFORMACIÓN / CONTÁCTENOS

usershop.redusers.com +54 (011) 4110-8700 usershop@redusers.com

*SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // **VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

Desarrollo web con Java desde cero

Complemento ideal de Java desde cero , esta obra reúne todas las herramientas necesarias para convertirse en un verdadero experto en el desarrollo de aplicaciones web con Java. En cada capítulo se presenta un ejemplo integrador, real y concreto, realizado con herramientas open source.

Dentro del libro encontrará*:

Introducción a la programación web / Configuración de un servidor / Trabajo con JSP / Programación de Struts y ActionForms / Programación de vistas / Validación automática de páginas

* Parte del contenido de este libro fue publicado previamente en Programación web Java , de esta misma editorial.

Otros títulos de la colección:

ISBN 978-987-1949-74-8

RU

REDUSERS.com

En nuestro sitio podrá encontrar noticias relacionadas y también participar de la comunidad de tecnología más importante de América Latina.

PROFESOR EN LÍNEA

Ante cualquier consulta técnica relacionada con el libro, puede contactarse con nuestros expertos: profesor@redusers.com.