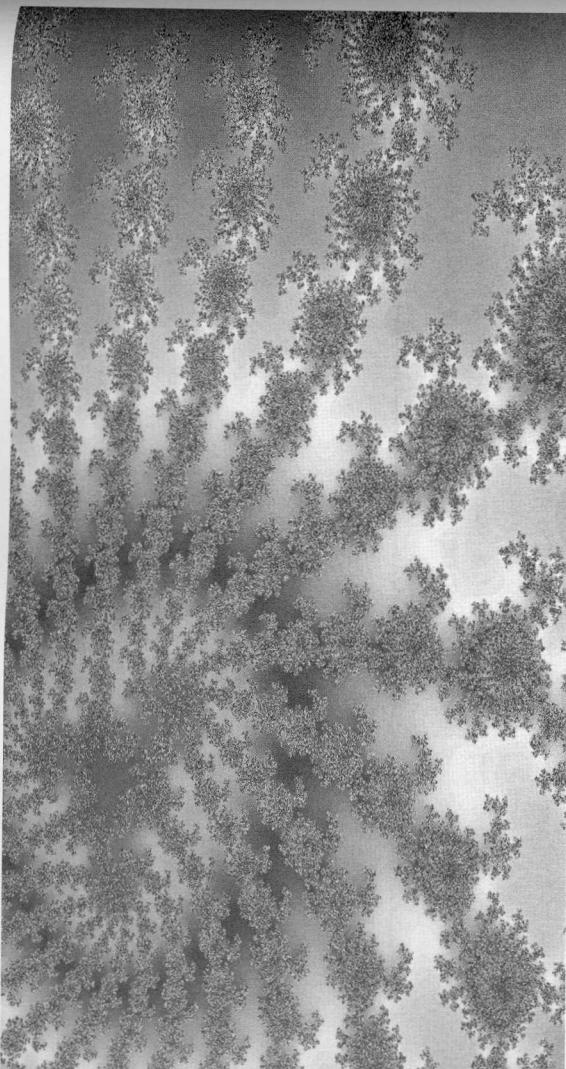


Una ventaja fundamental a la hora de utilizar la tecnología Java es que se trata de un lenguaje orientado a objetos (OO). Esto le permite escribir código reutilizable y altamente escalable. Según se vaya acostumbrando al desarrollo OO, podría reconocer algunas prácticas *recomendables* a seguir a la hora de desarrollar soluciones de una clase en particular. Por ejemplo, podría encontrarse con que en cada aplicación de entrada de datos en la que esté trabajando, tiende a codificar las rutinas de validación de datos de un modo similar. Si fuera a formalizar esta práctica recomendable y abstraer parte de los detalles de implementación, cabe la posibilidad de que otros pudieran utilizarla como mapa de carreteras para comenzar con su propio esfuerzo de desarrollo implementando una técnica ya probada en la validación de datos. Esto elimina un montón de esfuerzo de diseño además de numerosas pruebas.

Las prácticas recomendables publicadas se han dado en llamar *patrones de diseño*. Esto se originó en el mundo OO y se ha ido publicando en varios formatos específicos en implementaciones como C++ y Java, además de en diversos estudios generales. En particular, el libro *Design Patterns* de *Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides* (*Addison-Wesley, 1995*) se ha convertido en la guía definitiva para los patrones de diseño OO (orientado a objetos). Recientemente, el concepto de patrones de diseño ha influido bastante a la comunidad de desarrolladores en J2EE, motivando que Sun publicase *J2EE Patterns Catalog* ([HTTP:// developer.java.sun.com/developer/technicalArticles/J2EE/patterns/](http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/)). Estos patrones van dirigidos a temas de diseño típicos, ocupándose del modo en que podemos aplicar las distintas tecnologías J2EE para resolver tales temas.

## 5

# Desarrollar utilizando patrones



Este capítulo es el primero, entre varios, que se ocupará de los patrones de diseño empresarial. Trataremos sobre los beneficios de utilizar patrones, revisaremos el libro *J2EE Patterns Catalog*. Seleccionaremos los patrones más relevantes para el caso (el tema que nos ocupa en este libro) y finalizaremos con una discusión acerca del patrón MVC (en inglés, *Model-View-Controller* o controlador visual de modelos) en el que se basan la mayoría de patrones J2EE.

## ¿Por qué utilizar patrones?

Comenzaré a tratar los patrones ocupándome de responder a esta cuestión: ¿Por qué deberíamos pensar en los patrones de diseño para ayudarnos en nuestros esfuerzos de desarrollo? La respuesta a esta pregunta es muy sencilla. Los patrones de diseño son técnicas contrastadas que pueden volver a usarse y aplicarse a muchos problemas similares. También proporcionan un vocabulario común a la hora de discutir sobre diseño de aplicaciones.

## Son técnicas contrastadas

Al diseñar una aplicación, hay que resolver muchos problemas. En muchos casos, estos problemas no son específicos de la aplicación que estamos diseñando. Si deseáramos diseñar e implementar una solución personalizada para cada problema, entonces esa parte de código necesitaría ser experimentada en diversas iteraciones de pruebas y código subsiguiente hasta que se tratara exactamente de lo que necesitamos para nuestra aplicación específica.

Si fuéramos a tomar el escenario anterior y utilizar un patrón de diseño en lugar de una solución personalizada, entonces reduciríamos ampliamente el tiempo de desarrollo y de prueba. El patrón de diseño ya ha experimentado muchas iteraciones de pruebas y desarrollo para dar lugar a una práctica recomendable extendida por la industria. Obviamente, todavía necesitaremos realizar algo de desarrollo personalizado para implementar el patrón, pero en este momento únicamente necesitamos probar el código específico de la implementación y no una compleja parte funcional.

## Es un vocabulario común

Al hablar de diseño de aplicación, resulta útil tener un vocabulario común para comunicar nuestras opciones al resto del equipo de desarrollo. Por ejemplo, un patrón de diseño OO común es el patrón empresarial (*factory pattern*). Este patrón demuestra su utilidad cuando un objeto necesita ser instanciado en tiempo de ejecución, pero la clase de dicho objeto no se conoce en tiempo de compilación. De modo que, al hablar sobre opciones de diseño, podríamos decir algo como "Bueno, si implementamos un patrón empresarial en el módulo de informes, en el futuro podemos añadir nuevos informes sin modificar el armazón de la aplicación". Si todo el mundo en el equipo entiende el patrón empresarial, todos ellos pueden imaginar la solución basándose en ese comentario.

## Introducción al catálogo de patrones J2EE

Los arquitectos de Sun han compilado una serie de patrones de diseño y los han publicado como *J2EE Patterns Catalog*, disponible en el sitio *Java Developer Connection* (<http://developer.java.sun.com>). Estos patrones están dirigidos a problemas de aplicaciones comunes a lo largo de la aplicación de las tecnologías J2EE. El catálogo de patrones agrupa los diversos patrones en tres grupos:

- **Grupo de presentaciones:** Todo lo necesario para presentar los datos de la aplicación y los elementos de la interfaz de usuario está dentro del grupo de presentaciones de la aplicación. Las tecnologías fundamentales en uso son *JavaServer Pages (JSP)* y los *Java Servlet*.
- **Grupo de negocios:** El grupo de negocios es donde tiene lugar todo el proceso de los negocios. Las principales tecnologías J2EE de este grupo son los *Enterprise JavaBean (EJBs)*.
- **Grupo de integración:** El grupo de integración proporciona conexiones al grupo de recursos. El grupo de recursos incluye cosas como las colas de mensajes, bases de datos y sistemas heredados. Las tecnologías J2EE más destinadas son *Java Message Service (JMS)*, *Java Database Connectivity (JDBC)* y *Java Connector Architecture (JCA)*.

Puesto que éste es un libro sobre JSP, principalmente hablaremos de aquellos patrones que tienen alguna relación con el grupo de presentaciones. No intentaré describir cada patrón en detalle; el catálogo de patrones ya se ocupa muy bien de ello. El objetivo de este libro es proporcionar la mejor práctica y los mejores ejemplos. A tal fin, facilitaré la suficiente definición como para permitirle aplicar esos patrones a tareas de desarrollo comunes utilizando sólo unas mínimas modificaciones.

## Son reutilizables

En el espíritu del diseño OO, los patrones de diseño empresarial están pensados para ser reutilizados en varios proyectos. Cada patrón proporciona una solución experimentada para una clase específica de problemas. Estos problemas tienden a aparecer en muchas aplicaciones distintas. En lugar de reinventar la rueda cada vez, tiene más sentido aplicar un patrón de diseño utilizando sólo unas mínimas modificaciones.

## Ejemplos Prácticos

## Un vistazo a los patrones de diseño de presentaciones

Los patrones de los que hablaremos en este libro son *Decorating Filter*, *Front Controller*, *Dispatcher View* y *View Helper*. Existen algunos patrones de presentaciones más en el catálogo J2EE de los que no hablaré. Estos cuatro patrones son suficientes para ilustrar los ejemplos y prácticas recomendables que trataré.

Cada uno de estos patrones cubre una capa distinta de la lógica de la presentación. Cuando llega la solicitud, puede pasar a través de un filtro antes de ser procesada (patrón *Decorating Filter*). entonces podría ir a un servlet centralizado para ser procesada (patrón *Front Controller*). Una vez que ha sido procesada, entonces el servlet podría enviar los resultados a una página JSP específica (patrón *Dispatcher View*). Finalmente, la página JSP podría hacer uso de etiquetas personalizadas o JavaBeans para ayudar a incluir los datos en la salida HTML (patrón *View Helper*). La figura 5.1 muestra la relación entre estos patrones.

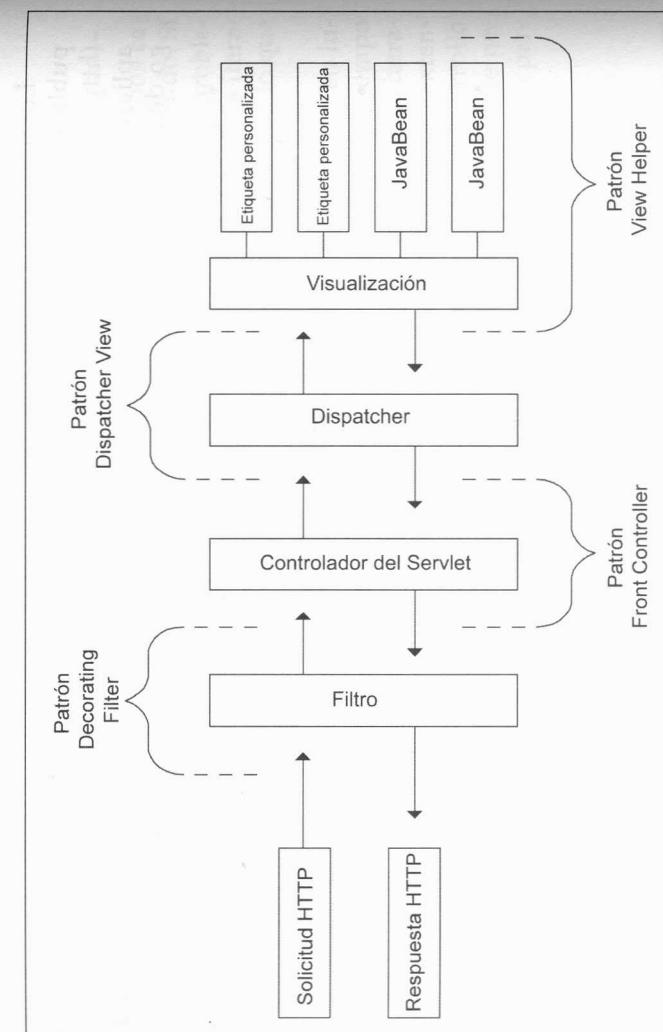


Figura 5.1. Patrones de presentaciones

Observe una introducción sobre cada patrón de los que hablaremos:

- **Decorating filter:** Este patrón aplica una clase de filtro a la solicitud o el objeto de respuesta mientras entra y sale del contenedor Web. Puede usar los

filtros como un lugar común donde registrar transacciones, autenticar usuarios e, incluso, asignar formato a los datos.

- **Patrón Front Controller:** Este patrón está desarrollado sobre el concepto de patrón MVC (lea la sección siguiente). Además supone la utilización de un servlet para administrar cada una de las solicitudes en lugar de insertar código de control dentro de cada página JSP.
- **Dispatcher View:** Dentro del controlador, existe una parte de código que determina si la solicitud procesada debería ser visualizada. En otras palabras, aplica una especie de estrategia para imaginar qué vista o página JSP utilizar para mostrar los datos actuales.
- **View Helper:** Una vez se ha elegido una visualización específica, JSP hace uso de diversas "ayudas" para adaptar los datos al contenido de salida final. Estas ayudas consisten en etiquetas personalizadas o JavaBeans.

## Entender MVC

Los patrones de presentaciones del catálogo J2EE están basados en la arquitectura MVC (*Model-View-Controller*). MVC se aplica en proyectos de desarrollo de software en un esfuerzo por separar los datos de la aplicación de la presentación de los mismos. Esta separación permite a la interfaz, o visualización, adoptar diferentes formas con una ligera modificación al código de la aplicación. Por ejemplo, utilizando un patrón MVC, una interfaz de usuario puede presentarse tanto como una página HTML (para navegadores Web) como una página WML (para dispositivos móviles), dependiendo del dispositivo que solicite la página. El controlador reorganizaría la fuente de la solicitud y plasmaría la visualización adecuada a los datos de la aplicación (véase figura 5.2).

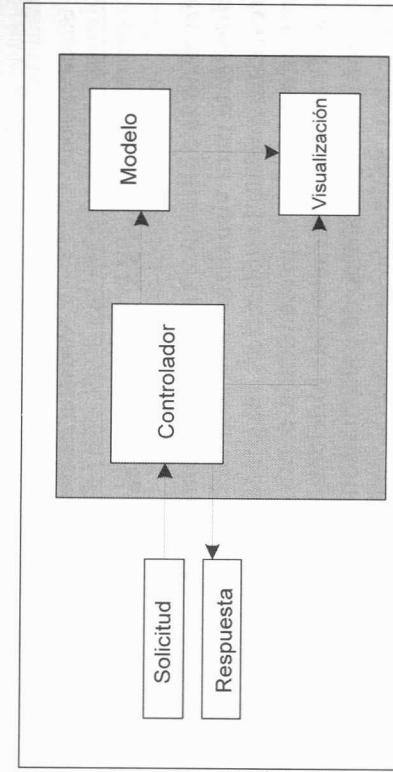


Figura 5.2. Arquitectura MVC

La idea de separar la lógica de la presentación de los datos y administrar todo ello mediante un controlador tiene sus orígenes en el desarrollo de la interfaz gráfica de usuario (GUI). Por ejemplo, imagine una interfaz que esté formada por muchos controladores de usuario distintos. Estos controladores contienen los datos, las instrucciones en cuanto al formato y el código que lanza un evento cuando se activa el controlador. Esto genera una interfaz para plataforma específica acoplada al propio código de la aplicación. Aplicando un patrón MVC y separando cada uno de sus componentes, la interfaz de usuario se convierte en algo reducido, fácilmente conectable y totalmente transferible entre plataformas. *Java Swing API* es un buen ejemplo de ello.

Puede aplicar el patrón MVC a otras áreas de desarrollo de software además de interfaces gráficas de usuario cliente / servidor. El desarrollo Web se ha beneficiado de esta idea separando claramente el código de la presentación de los datos de la aplicación y el código de controlador aparece unido a ambos. Así, imaginemos una sencilla aplicación Web que muestra las páginas de un catálogo. Normalmente, hablaremos de una búsqueda de página, una página de resultados y después la página que muestra en detalle un elemento. Cada página tiene la responsabilidad de autenticar el usuario, recuperar las preferencias del mismo, obtener los datos requeridos y, finalmente, controlar la navegación por la página (vea la figura 5.3).

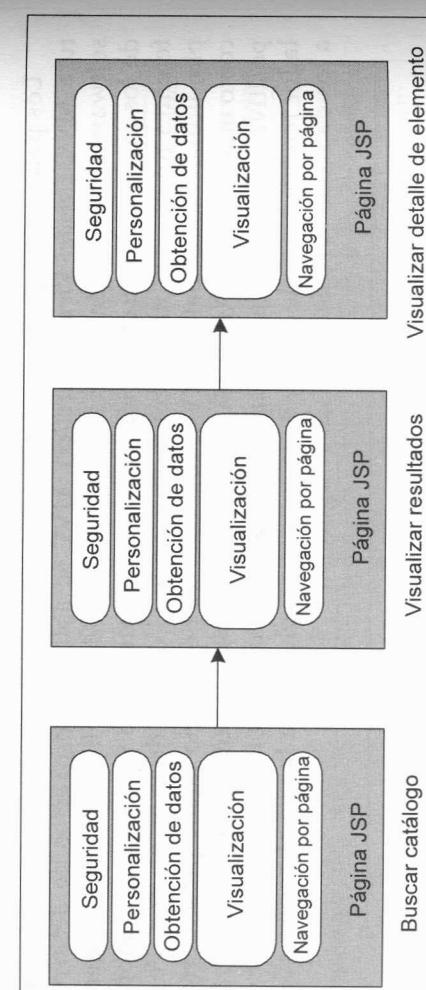


Figura 5.3. Aplicación de catálogo (sin MVC)

Observando la aplicación, resulta fácil comprobar la existencia de bastante código redundante para visualizar cada página. Esto no sólo introduce la posibilidad de errores, sino que ata la aplicación a la presentación incluyendo muchas funciones no relacionadas con la presentación dentro del código de la misma. Al aplicar un patrón MVC a esta aplicación, las funciones más comunes van a un controlador en el servidor. Ahora, el código de presentación es el único responsable de representar los datos de la aplicación en el formato más adecuado para un dispositivo en particular (normalmente un navegador Web). Observe la figura 5.4 para ver un ejemplo de MVC para esta aplicación.

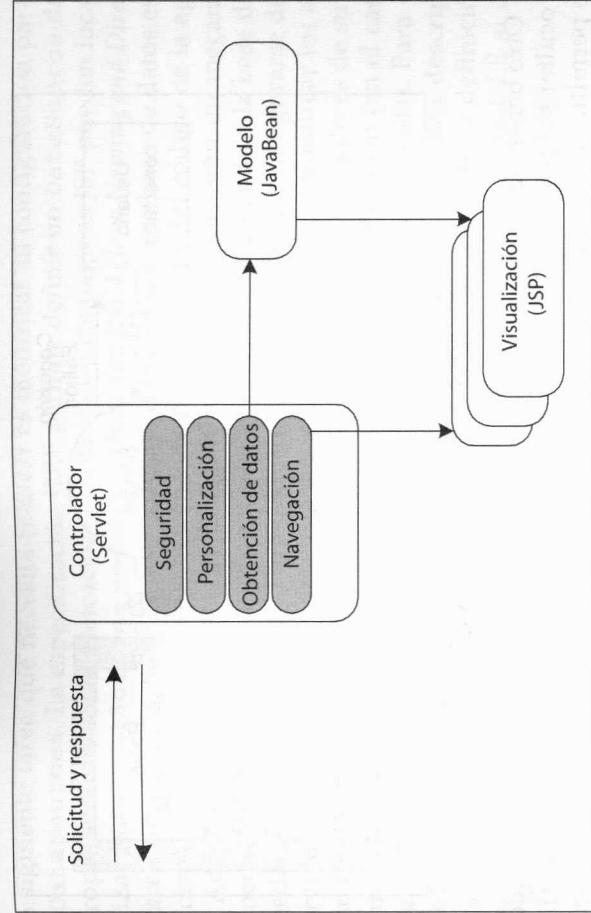


Figura 5.4. Aplicación de catálogo (con MVC)

## Ver MVC en acción

Como ejemplo del patrón MVC, vamos a crear una sencilla aplicación Web que recoge datos y los almacena en una base de datos. Los datos que vamos a obtener versan sobre información sanitaria que se guardará en la tabla de clientes de nuestra base de datos presupuestaria. Además de recoger los datos, la aplicación solicitará al usuario conectarse al sistema antes de acceder a cualquiera de sus páginas.

Este ejemplo es muy sencillo, pero sirve para ilustrar el beneficio que se obtiene al usar la arquitectura MVC. Comprobará cómo centralizar la seguridad de la aplicación otorgando al usuario un único punto de acceso a la aplicación. También estandarizará y compartirá la conexión a la base de datos usando un mecanismo de asociación de conexiones en el servidor de la aplicación. En los capítulos siguientes, utilizaré este ejemplo (entre otros) para presentar nuevos patrones. Teniendo esto en cuenta, este ejemplo es muy importante en estos momentos. Más tarde añadiremos características tales como la validación de campos y una mejor administración de los errores.

La aplicación comienza con una página de conexión y entonces pasa a un servlet principal que actúa como controlador (véase figura 5.5). El servlet determinará si proceder con la página siguiente basándose en el éxito del procedimiento de conexión. Una vez se haya conectado el usuario, irá a una página de encuestas donde añadirá su información y la enviará. Una vez más, el servlet requerirá la solicitud y llevará al usuario a la página siguiente. Si los datos se han grabado satisfactoriamente, se lleva al usuario a una página de confirmación.

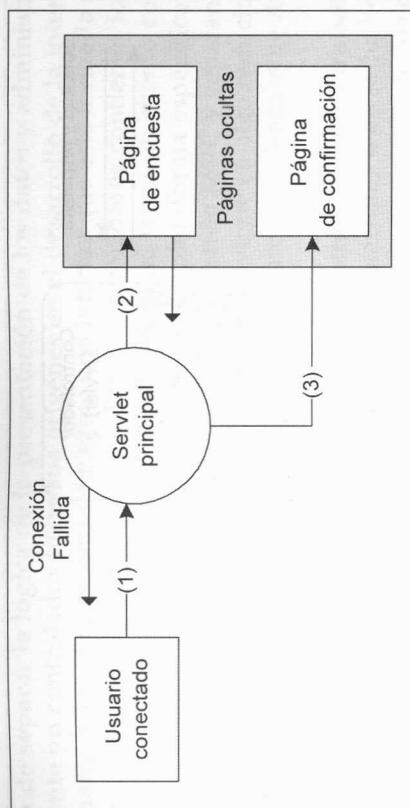


Figura 5.5. Aplicación de encuesta

Otro beneficio de utilizar un servlet como único punto de entrada es que permite ocultar sus páginas JSP al mundo exterior. Esto le permite asegurar el sistema al no permitir el acceso directo a su aplicación. La única página a la que puede acceder el usuario es la página de conexión. Si se fuera a escribir el nombre de otra página, el servidor devolvería un error 404 ("no encontrada"). Puede conseguir esto "ocultando" sus páginas JSP dentro del directorio \WEB-INF. Por definición, todo lo ubicado en este directorio es inaccesible directamente al usuario. Sin embargo, el servlet que actúa como nuestro controlador puede acceder a este directorio y, por lo tanto, se le permite enviar solicitudes a páginas que residan allí. Observe el aspecto de la estructura de su directorio cuando haya terminado este ejemplo:

```

webapps\jspbook\ch5\vlogin.jsp
webapps\jspbook\ch5\myError.jsp
webapps\jspbook\ch5\myHeader.htm
webapps\jspbook\ch5\myFooter.htm
webapps\jspbook\ch5\images\logo.gif
webapps\jspbook\ch5\WEB-INF\jsp\census.jsp
webapps\jspbook\ch5\WEB-INF\jsp\thankyou.jsp
  
```

### Configurar la aplicación

Antes de empezar a codificar, necesita añadir una tabla a la base de datos y después modificar la configuración de su servidor de aplicación para acomodarlo al uso de *DataSources*. La tabla que necesita añadir es una tabla de usuario que contenga el ID de usuario, la contraseña y un ID de cliente. El ID de cliente crea un registro de cliente que se corresponde con el cliente. Idealmente, este campo debería generarse dinámicamente, pero en nuestro caso vamos a limitarnos a codificar este valor. Observe el script para actualizar la base de datos:

```

DROP TABLE IF EXISTS user;
CREATE TABLE user (id varchar(10) not null, pwd varchar(10), cust_id int);
INSERT INTO user VALUES ('apatz', 'apress', 6);
  
```

La siguiente tarea que necesita realizar es modificar su configuración para utilizar *DataSources*. La especificación J2EE permite definir un *DataSource* dentro del propio servidor de la aplicación. Los servlet y las páginas JSP pueden localizar y utilizar estos *DataSources* mediante la interfaz *JNDI (Java Naming and Directory Interface)*. Una ventaja clave para acceder de este modo a una base de datos es que la información de conexión se almacena en un lugar fuera del código de la aplicación. Además, la mayoría de servidores de aplicación incorporan un mecanismo de asociación de conexiones que podemos aprovechar accediendo a la base de datos mediante un *DataSource*. Para que esto funcione, necesitará asegurarse de que su servidor de aplicación soporta esta característica. Antes de modificar los archivos de configuración necesarios, asegúrese de agregar los controladores de su base de datos a un directorio accesible para el servidor de la aplicación (en el caso de *Tomcat 4.0.1*, coloque los controladores en el directorio *\common\lib*). Para crear un *DataSource* en su servidor de aplicación, necesitará añadir una descripción del mismo al archivo *server.xml*. Esta descripción va dentro de su definición de contexto según se observa en el listado 5.1 (revise la especificación J2EE si desea obtener más detalles).

Listado 5.1. server.xml

```

<Context path="/jspBook"
  docBase="jspBook"
  crossContext="false"
  debug="0"
  reloadable="true" >

  <Logger className="org.apache.catalina.logger.FileLogger"
    prefix="localhost_jspBook_log." suffix=".txt"
    timestamp="true"/>

  <Resource name="jdbc/QuotingDB" auth="SERVLET"
    type="javax.sql.DataSource" />

  <ResourceParams name="jdbc/QuotingDB">
    <parameter>
      <name>driverClassName</name>
      <value>org.gjt.mm.mysql.Driver</value>
    </parameter>
    <parameter>
      <name>driverName</name>
      <value>jdbc:mysql://localhost:3306/quoting</value>
    </parameter>
    <ResourceParams name="gjt.mm.mysql">
      <parameter>
        <name>url</name>
        <value>jdbc:mysql://localhost:3306/quoting</value>
      </parameter>
    </ResourceParams>
  </ResourceParams>
</Context>
  
```

Ahora que ha descrito el *DataSource* al servidor de aplicación, necesita informar de ello a su aplicación. Hágalo añadiendo una entrada de recurso en su archivo *web.xml*. El listado 5.2 contiene lo que debería aparecer en este archivo (dentro de sus etiquetas <web-app>).

**Listado 5.2. web.xml**

```

<resource-ref>
  <description>Definición del recurso a una empresa para instancias java.sql.Connection que deben emplearse para comunicarse con una base de datos específica que está configurada en el archivo server.xml.</description>
  <res-ref-name>jdbc/QuotingDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>SERVLET</res-auth>
</resource-ref>
```

Finalmente, para utilizar DataSource, necesita reemplazar cualquier código que posea una conexión a base de datos mediante el siguiente código (en el servlet de este ejemplo debe ejecutar una vez este código dentro del método init):

```

try {
    Context initCtx = new InitialContext();
    Context envCtx = (Context) initCtx.lookup("java:comp/env");
    ds = (DataSource) envCtx.lookup("jdbc/QuotingDB");
    dbCon = ds.getConnection();
}
catch (javax.naming.NamingException e) {
    System.out.println("A problem occurred while retrieving a DataSource object");
    System.out.println(e.toString());
}
catch (java.sql.SQLException e) {
    System.out.println("A problem occurred while accessing the database.");
    System.out.println(e.toString());
}
```

### Definir el modelo

Antes de adentrarnos en el controlador o las visualizaciones, necesita definir el modelo. El modelo es responsable de almacenar los datos que se mostrarán por una o más visualizaciones. Típicamente, un modelo existe como EJB (Enterprise JavaBean) o simplemente un JavaBean. Para este ejemplo, usaremos un sencillo JavaBean. Recordará del capítulo 3 que utilizó un JavaBean para modelar los datos del cliente. Utilizará parte de dicho JavaBean y añadirá algunos métodos adicionales para seguir nuestro propósito.

Aparte de eliminar el código innecesario, necesitará añadir dos nuevos métodos al CustomerBean que se ha creado en el capítulo 3. El primer método que añadirá

**Listado 5.3. CustomerBean.java**

```

package jspbook.ch5;
import java.util.*;
import java.sql.*;
import javax.servlet.http.*;

public class CustomerBean implements Serializable {
    /* Variables de miembro */
    private String lname, fname, sex;
    private int age, children;
    private boolean spouse, smoker;
    /* Variables de ayuda */
    private String uid;

    /* Constructor */
    public CustomerBean() {
        /* Inicializar propiedades */
        setLname("");
        setFname("");
        setSex("");
        setAge(0);
        setChildren(0);
        setSpouse(false);
        setSmoker(false);
    }

    public void populateFromParms(HttpServletRequest _req) {
        /* Publicar las propiedades del bean para los parámetros solicitados */
        setLname(_req.getParameter("lname"));
        setFname(_req.getParameter("fname"));
        setSex(_req.getParameter("sex"));
        setAge(_req.getIntParameter("age"));
        setChildren(_req.getParameter("children"));
        setSpouse(_req.getBooleanParameter("married"));
        setSmoker(_req.getParameter("smoker").equals("Y") ? true : false);
        // Obtener una sesión y publicar identificador de usuario uid
        HttpSession session = _req.getSession();
        uid = (String) session.getAttribute("uid");
    }

    /* Métodos Accesoror */
    /* Apellido */
    public void setLname(String _lname) { lname = _lname; }
    public String getLname() { return lname; }
```

es populateFromParameters. Este método toma un objeto HttpServletRequest como parámetro. El método es responsable de leer los campos de entrada del objeto solicitado y publicar las propiedades del bean con sus valores. Además, el ID (identificador) de usuario se extrae de la sesión del usuario y se almacena en el bean para un uso posterior. El listado 5.3 muestra el código actualizado del CustomerBean.

```

    /* Nombre */
    public void setFname(String fname) { fname = _fname; }
    public String getFname() { return fname; }

    /* Sexo */
    public void setSex(String sex) { sex = _sex; }
    public String getSex() { return sex; }

    /* Edad */
    public void setAge(int age) { age = _age; }
    public int getAge() { return age; }

    /* Número de hijos */
    public void setChildren(int children) { children = _children; }
    public int getChildren() { return children; }

    /* ¿Casado/a? */
    public void setSpouse(boolean spouse) { spouse = _spouse; }
    public boolean getSpouse() { return spouse; }

    /* ¿Fumador/a? */
    public void setSmoker(boolean smoker) { smoker = _smoker; }
    public boolean getSmoker() { return smoker; }
}

public boolean submit(URLConnection dbCon) {
    Statement s = null;
    ResultSet rs = null;
    String custId = "";
    StringBuffer sql = new StringBuffer(256);

    try {
        // Comprobar si existe el cliente (utilice uid para obtener custID)
        s = _dbCon.createStatement();
        rs = s.executeQuery("select * from user where id = '" + uid + "'");
        if (rs.next()) {
            custId = rs.getString("cust_id");
        }
    }
    rs = s.executeQuery("select * from customer where id = " + custId);
    if (rs.next()) {
        // Actualizar registro
        sql.append("UPDATE customer SET ");
        sql.append("name=").append(fname).append(",");
        sql.append("age=").append(age).append(",");
        sql.append("sex=").append(sex).append(",");
        sql.append("married=").append((spouse) ? "Y" : "N").append(",");
        sql.append("children=").append(children).append(",");
        sql.append("smoker=").append((smoker) ? "Y" : "N").append(",");
        sql.append("id=").append(custId).append(",");
    }
    else {
        // Insertar registro
        sql.append("INSERT INTO customer VALUES (");
    }
}

```

### Ajustar la visualización

La presentación se almacena en tres archivos JSP. El primero, login.jsp, es accesible al público, mientras los otros dos sólo son accesibles desde el servlet del controlador. La página de conexión es una sencilla pantalla de entrada de usuario y contraseña que envía sus datos al servlet Main. Observará que añade un parámetro al servidor llamado action. Este indica al servidor qué debe hacer. En este caso, la acción es login. Si aparece un error mientras intenta conectarse, el servidor agrega un atributo a la sesión que indica que existe un problema y después devuelve al usuario a la página de conexión. Debido a esto, la página de conexión comprueba la sesión para encontrar el atributo adecuado y muestra el correspondiente mensaje de error si lo encuentra. Observe en el listado 4.5 la lista completa de la página de conexión.



**Listado 5.4.** login.jsp (`\webapps\jspBook\ch5\login.jsp`)

```

<%@ page errorPage="myError.jsp?from=login.jsp" %>

<html>
    <head>
        <title>Quoting System Login</title>
    </head>
    <body bgcolor="#FFFF99">
        <form method="post" action="Main?action=login">
            <%@ include file="myHeader.html" %> => Tiempo de respuesta
            <input type="text" name="uid" value="uid" />
            <input type="password" name="pwd" value="password" />
            <input type="submit" value="Login" />
        </form>
    </body>
</html>

```

```

<p align="center">
  <font face="Arial, Helvetica, sans-serif" size="6" color="#003300">
    <b><i>Login to Quoting System</i></b>
  </font>
</p>

<p>&nbsp;</p>

<% String loginError = (String) session.getAttribute("loginError");
  if (loginError != null && loginError.equals("y")) { %>
<center>
  <font color="#ff0000">Invalid login, please try again.</font>
</center>
<% } %>

<table width="199" border="0" align="center" cellpadding="5">
<tr>
  <td>
    <font face="Arial, Helvetica, sans-serif" size="2">User ID:</font>
    <td><input type="text" name="UID"></td>
  </tr>
  <tr>
    <td><font face="Arial, Helvetica, sans-serif" size="2">Password:</font>
    <td><input type="password" name="PWD"></td>
  </tr>
  <tr align="center">
    <td colspan="2" value="Submit" name="submit" type="submit"></td>
  </tr>
</table>

```

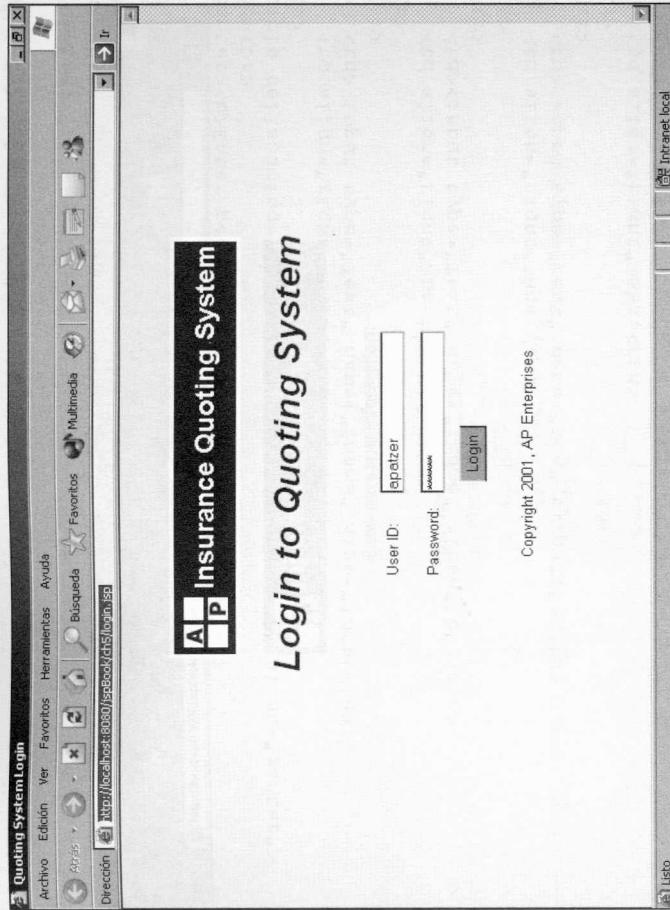


Figura 5.6. Página de conexión

**Figura 5.6.** Página de conexión

```

Listado 5.5. census.jsp (\\webapps\jspBook\ch5\census.jsp)

<!-- Directivas JSP -->
<%@ page errorPage="myError.jsp?from=census.jsp" %>

<html>
  <head>
    <title>Insurance Quoting System</title>
  </head>
  <body bgcolor="#FFFFFF">

```

&lt;%&gt;

```

    <form action="Main?action=submit" method="post">
      <br><br>

```

```

        <input type="text" name="UID" value="Login">

```

```

        <input type="password" name="PWD" value="Login">

```

```

      </form>

```

```

    <%@ include file="myFooter.html" %>

```

```

  </body>
</html>

```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

```


```

<font color="#ffff00">Error recording survey data, please try again.

En la figura 5.7 podemos comprobar el aspecto que presenta la página de encuesta.

A continuación presentamos partes

Archivo Edición Ver Favoritos Herramientas Ayuda

Atres > Busqueda FAVORITOS Multimedia

Dirección http://localhost:8080/jspBook/ch5/Main?action=login

**Insurance Quoting System**

Enter personal information:

First Name:	Andrew
Last Name:	Pater
Age:	30
Sex	<input checked="" type="radio"/> Male <input type="radio"/> Female
Married:	<input checked="" type="checkbox"/> Yes
Children:	2
Smoker:	N
<b>Submit</b>	

Úneto

Figura 5.7. Página de encuesta

Finalmente, una vez se han enviado los datos, se remite la solicitud a una página de confirmación (thankyou.jsp). Es una sencilla página que confirma que los datos han sido aceptados. Si existiera un error al intentar enviar los datos, se devolvería el control a la página de encuesta (census.jsp) y aparecería un mensaje de error en la parte superior (similar al que aparecía en la página de conexión). Observe el listado 5.6, donde aparece el código de la página de confirmación.

**Listado 5.6.** `thankyou.jsp` (`\websites\jspBook\ch5\thankyou.jsp`)

```

<!-- Directivas JSP -->
<%@ page errorPage="myError.jsp?from=thankyou.jsp" %>

<html>
<head>
<title>Insurance Quoting System</title>
</head>

<body bgcolor="#FFFF99">

```

```

<basefont face="Arial">
<%@ include file="/ch5/myHeader.html" %>
<br><br>
<center>
Your survey answers have been recorded. Thank you for participating in this
survey.
</center>
<br><br>
<%@ include file="/ch5/myFooter.html" %>
</body>
</html>

```

La figura 5.8 muestra la página de confirmación que se muestra después de una correcta grabación de los datos de la encuesta.

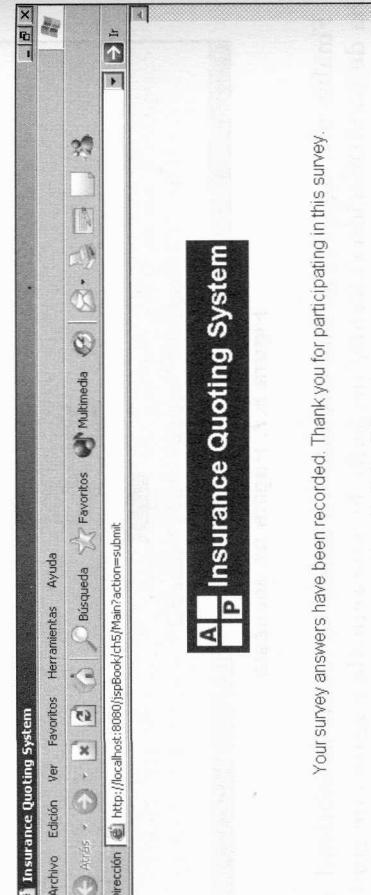


Figura 5.8. Página de confirmación

```

<servlet>
  <servlet-name>
    Main
  </servlet-name>
  <servlet-class>
    jspbook.ch5.Main
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>
    Main
  </servlet-name>
  <url-pattern>
    /ch5/Main
  </url-pattern>
</servlet-mapping>
<taglib>

```

El método init obtiene la conexión a la base de datos mediante el Data Source que creamos anteriormente. Esta conexión a base de datos se encuentra cerrada en el método destroy al final del ciclo de vida del servlet. Cada solicitud se tramita por el método doPost. Allí dentro, la acción se determina activando el parámetro action. La primera vez que esto ocurre, la acción login dirige el servlet al método authenticate. Si se establece la conexión, entonces se lleva al usuario a la página census.jsp.

Lo más importante a reseñar, en este caso, es que toda la seguridad, la conectividad a bases de datos y el control de navegación están centralizados dentro de este único servlet. Puede volver a utilizar el código en diversos sitios. Por ejemplo, el código de navegación para el método goToPage. Si necesita cambiar esta funcionalidad, solamente necesita hacerlo en un sitio. Mientras explora otros patrones comprobará lo útil que resulta esta arquitectura. El objetivo de este ejemplo es simplemente ilustrar el concepto básico de un patrón MVC. Revise el listado 5.7 para el servlet controlador.

Listado 5.7. Main.java

```

package jspbook.ch5;

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

import jspbook.ch5.CustomerBean;

public class Main extends HttpServlet {
  //Conexión dbCon;
  DataSource ds;
  HttpSession session;
}

```

*Servicio controlador*

### Crear el controlador

Utilizará un servlet como controlador (Main). Para que esté accesible, añada la siguiente entrada a su archivo web.xml (dentro de las etiquetas <web-app>):

Figura 5.9. Entrada para el controlador en web.xml

```

    /* Inicializar servlet. Utilice JNDI para buscar un DataSource */
    public void init() {
        try {
            Context initCtx = new InitialContext();
            Context envCtx = (Context) initCtx.lookup("java:comp/env");
            ds = (DataSource) envCtx.lookup("jdbc/QuotingDB");
            dbCon = ds.getConnection(); Se necesita conexión
        }
        catch (javax.naming.NamingException e) {
            System.out.println("A problem occurred while retrieving a DataSource
object");
            System.out.println(e.toString());
        }
    }

    public void doPost(HttpServletRequest _req, HttpServletResponse _res)
throws ServletException, IOException {
    /* Actualizar atributos de sesión */
    session = _req.getSession();
    session.removeAttribute("loginError");
    session.removeAttribute("submitError");

    String action = _req.getParameter("action");
    /* Autentificar usuario si la solicitud proviene de una página de
    conexión */
    if (action.equals("login")) {
        String uid = _req.getParameter("UID");
        String pwd = _req.getParameter("PWD");
        if (authenticate(uid, pwd)) {
            session.setAttribute("validUser", "y");
            session.setAttribute("LoginError", "n");
            session.setAttribute("uid", uid);
            gotoPage("/WEB-INF/jsp/ch5/census.jsp", _req, _res);
        }
        /* Si falla la conexión del usuario, devolver atributos a la página
        de conexión para volver a intentar */
    }
    else {
        loginError(_req, _res);
    }
}

/* Grabar los datos de encuesta si la solicitud proviene del formulario
de encuesta */
else if (action.equals("submit")) {
    /* Asegurarse de que el usuario se ha conectado antes de grabar los
    datos */
    String validUser = (String) session.getAttribute("validUser");
    if (validUser.equals("y")) {
        if (recordSurvey(_req)) {
            /* Restablecer el marcador validUser y enviarlo a la página
            ThankYou */
        }
    }
}

```

```

    /* Iniciar sesión */
    session.removeAttribute("validUser");
    gotoPage("/WEB-INF/jsp/ch5/thankyou.jsp", _req, _res);
}
else {
    session.setAttribute("submitError", "y");
    gotoPage("/ch5/login.jsp", _req, _res);
}
/* Si el usuario no se ha conectado, enviarlos a la página de
conexión */
else {
    loginError(_req, _res);
}
}

/* Enviar solicitud a otra página */
private void gotoPage(String _page, HttpServletRequest _req,
HttpServletResponse _res)
throws IOException, ServletException {
RequestDispatcher dispatcher = _req.getRequestDispatcher(_page);
if (dispatcher != null)
    dispatcher.forward(_req, _res);
}

/* Definir los atributos de error en la sesión y volver a Login page
(página de conexión) */
private void loginError(HttpServletRequest _req, HttpServletResponse
_res)
throws IOException, ServletException {
session.setAttribute("validUser", "n");
session.setAttribute("LoginError", "y");
gotoPage("/ch5/login.jsp", _req, _res);
}

/* Verificar la validez del usuario */
private boolean authenticate(String _uid, String _pwd) {
Connection dbCon = null;
ResultSet rs = null;
try {
    dbCon = ds.getConnection();
    Statement s = dbCon.createStatement();
    rs = s.executeQuery("select * from user where id = '"
    + _uid + "' and pwd = '" + _pwd + "'");
    return (rs.next());
}
catch (java.sql.SQLException e) {
    System.out.println("A problem occurred while accessing the
database.");
    System.out.println(e.toString());
}
}

```

```

finally {
    try {
        dbCon.close();
    }
    catch (SQLException e) {
        System.out.println("A problem occurred while closing the
                           database.");
        System.out.println(e.toString());
    }
    return false;
}

/* Utilizar CustomerBean, grabar los datos*/
public boolean recordSurvey(HttpServletRequest _req) {
    Connection dbCon = null;

    try {
        dbCon = ds.getConnection();
        CustomerBean cBean = new CustomerBean();
        cBean.populateFromParms(_req);
        return cBean.submit(dbCon);
    }
    catch (java.sql.SQLException e) {
        System.out.println("A problem occurred while accessing the
                           database.");
        System.out.println(e.toString());
    }
    finally {
        try {
            dbCon.close();
        }
        catch (SQLException e) {
            System.out.println("A problem occurred while closing the
                               database.");
            System.out.println(e.toString());
        }
    }
    return false;
}

public void destroy() {
}
}

```

muchos desarrolladores distintos en la industria. El catálogo *Patterns Catalog* contiene diversos patrones de diseño para el desarrollo Java empresarial. Este libro trata sobre cuatro patrones de presentación específicos que ayudan a describir las más recomendadas prácticas para el desarrollo en JSP.

Cada patrón soporta una arquitectura MVC, que organiza nuestras aplicaciones Web en tres partes lógicas. El modelo almacena los datos de la aplicación, la visualización muestra dichos datos y el controlador administra las solicitudes y la navegación por la aplicación. Los capítulos siguientes explorarán los patrones J2EE que extienden cada una de estas áreas y aplican estrategias para maximizar la eficacia del desarrollo de aplicaciones Web basadas en MVC.

*My first JSP!*

Este capítulo le ha enseñado el concepto de utilizar patrones para diseñar sus aplicaciones. Los patrones son prácticas recomendables que han sido probadas por

## Resumen