

Introducción

Resumen

Capítulo 1

Capítulo 2

Capítulo 3

Capítulo 4

Capítulo 5

Capítulo 6

Capítulo 7

Capítulo 8

Capítulo 9

Capítulo 10

Capítulo 11

Capítulo 12

Capítulo 13

Capítulo 14

⑧ 7

El patrón Controlador frontal

En el capítulo 5 se habló acerca de la idea de la arquitectura MVC (Model-View-Controller). Este capítulo dará algunos pasos más lejos introduciendo los diversos modos de centralizar el procesamiento dentro de sus aplicaciones Web. Al igual que en el capítulo anterior, comenzaré por definir el patrón, después recorreremos las distintas estrategias para implementar dicho patrón y, finalmente, presentaré algunos ejemplos concretos del patrón. El objetivo de este capítulo es crear un sólido armazón de administración de solicitudes mediante el patrón *Front Controller* (Controlador frontal).

Definir el patrón

La mayoría de las aplicaciones Web comienzan como un simple conjunto de páginas que realizan una recolección de datos e informan acerca de la funcionalidad. Una página realiza un poco de procesamiento y entonces muestra la siguiente página. Esta página realiza algo más de procesamiento y muestra otra página con más resultados. Cada página contiene la autorización de seguridad necesaria, el código de administración de solicitudes, la lógica empresarial, la lógica de presentaciones y el código de navegación. La mayor parte de este código se duplica en cada página siguiente, pero en el caso de una aplicación sencilla que nunca cambiará, esto no debería suponer un problema, ¿no?

Desgraciadamente, estas aplicaciones sencillas suelen evolucionar en sistemas más sofisticados en los que se basa en gran medida el negocio. El resultado de esto

es un grupo de páginas JSP, o Java servlets, que contienen código de lógica de navegación dentro de cada página, duplican el código insertado dentro de cada página y no hay un modo sencillo de integrar otros servicios en la aplicación.

El patrón *Front Controller* (Controlador frontal) define el modo de implementar un patrón MVC dentro de sus aplicaciones utilizando páginas JSP y Java servlets. Lo que quizás es más importante es que el patrón *Front Controller* fomenta la creación de un armazón estándar para administrar solicitudes. Esto permitirá una más sencilla integración de la nueva funcionalidad mientras la aplicación crece más allá de su intención original. Este patrón está formado por un controlador central, ya sea un servlet o JSP, que sirve como único punto de entrada a la aplicación (véase figura 7.1). Este controlador puede administrar recursos compartidos como conexiones de base de datos y sesiones HTTP. También puede administrar el código normalmente ejecutado a través de múltiples solicitudes como una autenticación de usuario. Otro beneficio que se obtiene al definir un único punto de entrada es que podemos conectar servicios comunes como filtros en una cadena de filtrado, según se indicó en el capítulo anterior.

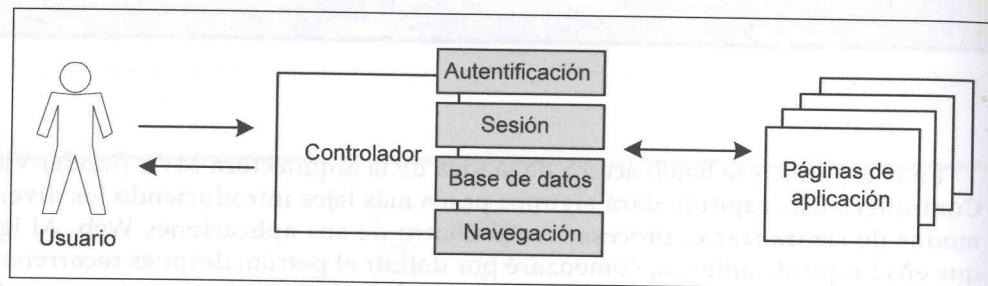


Figura 7.1. El patrón *Front Controller*

Desarrollar estrategias

El patrón *Front Controller* por sí mismo no es más que el patrón MVC aplicado a páginas JSP y servlets. Lo destacable radica en las diversas estrategias utilizadas para crear un sólido armazón de administración de solicitudes dentro del patrón *Front Controller*. Una vez creado este armazón, no sólo ayuda a que su aplicación actual sea más eficaz y extensible, sino que también puede volver a ser utilizado en otras aplicaciones. Recuerde que la ventaja de utilizar patrones de diseño es que suponen un paso adelante en cuanto al desarrollo. Un armazón reutilizable de administración de solicitudes puede ser una parte importante de la aplicación que no requiere pruebas y modificaciones adicionales, permitiendo al usuario enfocar sus esfuerzos en el código específico de la aplicación en lugar de imaginar cómo administrar las solicitudes entrantes y dirigirlas adecuadamente.

Un buen armazón de administración de solicitudes define una ruta estándar la cual debe seguir la solicitud mientras que se mueve por la aplicación. Precisamente

con este propósito, definiré el **ciclo vital de una solicitud**, que consiste en lo siguiente:

- El usuario emite una solicitud al controlador de la aplicación (servlet o página JSP).
- La solicitud se divide en parámetros de formulario.
- La solicitud se autentifica.
- La acción solicitada se mapea a un recurso específico.
- Se crea un objeto de comandos para procesar la solicitud.
- La solicitud se envía a la página adecuada.
- La salida de la acción solicitada se muestra al usuario.

Este ciclo de vida da unas sólidas bases para definir el armazón. Hay diversas estrategias para implementar el patrón *Front Controller* que dirige cada paso en su ciclo de vida. El objetivo del capítulo es usar estas estrategias para crear un sólido armazón de administración de solicitudes que puede volver a usarse en otros proyectos. Se basará en cada estrategia hasta que llegue al armazón final. Al acabar, tendrá un mecanismo estándar de aceptación de solicitudes, activación, procesamiento y envío de las mismas a un recurso apropiado para visualizar sus resultados (véase figura 7.2). En este armazón puede insertar tareas comunes en puntos específicos dentro del ciclo de vida solicitado. Por ejemplo, la autenticación del usuario puede ocurrir usando el objeto de ayuda solicitado antes de crear el objeto de comandos.

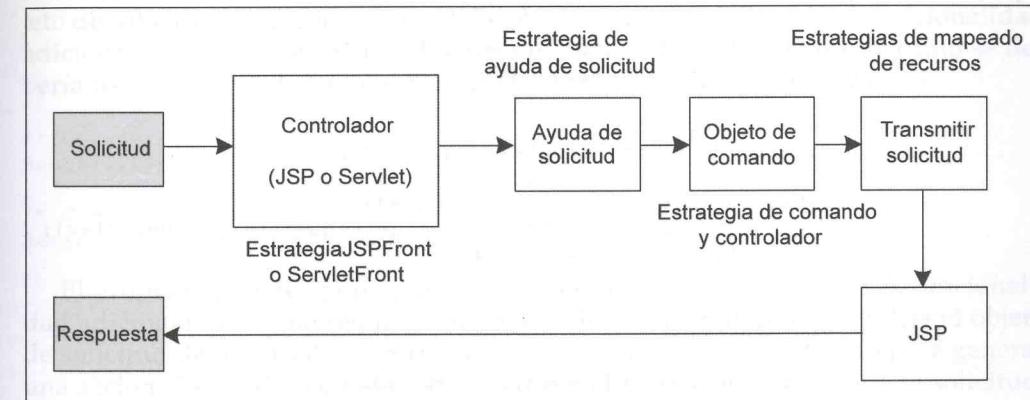


Figura 7.2. Armazón de administración de solicitudes dentro del patrón *Front Controller*

Frontal JSP contra estrategia frontal del servlet

Puede insertar el controlador como una página JSP o un servlet Java. El hecho de implementar un controlador como página JSP es una opción viable; sin embargo,

no es preferible a utilizar un servlet como su controlador. Crear un controlador como una página JSP es un hecho un tanto intuitivo. El código del controlador no tiene nada que ver con la visualización del contenido y es más adecuado para un servidor que para una página de marcado. La mayoría de los programadores preferirían trabajar con un servlet en lugar de con una página JSP al ocuparse del código del controlador. Las páginas JSP resultan más adecuadas para la adaptación y presentación de los datos.

Aquí tiene un concepto básico de cómo sería el aspecto de un servlet controlador (excepto la administración de excepciones necesaria):

```
public class ControllerServlet extends HttpServlet {
    public void init () {
        /* Realizar la primera inicialización de los recursos compartidos
           como conexiones a bases de datos. */
    }
    public void doGet(HttpServletRequest _req, HttpServletResponse _res)
        throws ServletException, IOException
    {
        /* Pasar al método doPost */
        doPost(_req, _res);
    }
    public void doPost(HttpServletRequest _req, HttpServletResponse _res)
        throws ServletException, IOException
    {
        /* Crear solicitud para el objeto de ayuda */
        ReqUtil reqUtil = new ReqUtil(_req);

        /* Crear objeto de acción (comando) */
        Action action = reqUtil.getAction();

        /* Ejecutar acción */
        String view = action.execute(_req, _res);

        /* Tratar por la visualización adecuada */
        RequestDispatcher dispatcher = _req.getRequestDispatcher(view);
        dispatcher.forward(_req, _res);
    }
    public void destroy()
    {
        /* Realizar aquí operaciones de limpieza
           como cerrar bases de datos. */
    }
}
```

Los servicios de aplicación
se ejecutan dentro de un contenedor de aplicación.

El controlador del servlet le otorga la oportunidad de inicializar recursos compartidos antes de la primera solicitud y después cerrarlos antes de que el servidor quede fuera de servicio. Estos recursos compartidos suelen ser conexiones a bases de datos, pero también podrían ser cosas como colas de mensajes entrantes o EJBs.

(Enterprise JavaBeans). Existen dos maneras distintas de que una solicitud acceda a nuestro controlador. Una solicitud puede venir mediante los métodos `doGet` y `doPost`. En la mayoría de los casos, puede elegir un método sobre el otro anulando uno de ellos e indicándole que llame al otro, pasando al mismo la solicitud y los objetos de respuesta. También podría crear un método diferente para procesar la solicitud y hacer que los métodos `doGet` y `doPost` opten por procesar en el nuevo método.

Dentro del método de procesamiento de la solicitud (`doGet` o `doPost`), puede crear un objeto de ayuda que obtenga el objeto de solicitud original. Este objeto de ayuda obtiene los parámetros de la solicitud y extrae la acción requerida del objeto de solicitud. Esta acción indica al controlador qué debería hacer con la solicitud. Se crea entonces un objeto de comando por el objeto de ayuda de la solicitud, quien ejecuta su lógica empresarial. Cualquier dato generado como resultado de la acción se suele almacenar como EJB o simplemente como JavaBean. La visualización correspondiente a la acción recibe la solicitud y utilizará el EJB o el JavaBean para adaptar los datos al código de presentación adecuado. Las secciones siguientes se ocupan de los detalles de este conjunto de operaciones.

Estrategia de ayuda de solicitud

Para implementar un armazón de administración de solicitudes, es importante crear un mecanismo estándar que complete la solicitud con una funcionalidad adicional que quizás pueda ser necesaria para procesar la solicitud y para obtener los parámetros de solicitud o determinar la acción que debería tomar el servlet. La estrategia de ayuda de solicitud ayuda a crear un objeto de ayuda que acepta el objeto de solicitud en su constructor. Este objeto de ayuda, junto con su funcionalidad adicional, está entonces disponible para el resto del método. Observe cómo se debería usar el objeto dentro del método `doPost` que vimos anteriormente:

```
/* Obtener un objeto de solicitud mediante la ayuda */
ReqUtility reqUtil = new ReqUtility(_req);

/* Crear un objeto Action basándose en los parámetros de solicitud */
Action action = reqUtil.getAction();
```

El propio objeto de ayuda puede contener por sí mismo mucha más funcionalidad adicional de lo que requiere su aplicación. Como mínimo, obtendría el objeto de solicitud de dentro de su constructor y proporcionaría un método para generar una acción al servidor con el fin de basarse en los parámetros dados a la solicitud. Observe el aspecto de una sencilla ayuda de solicitud:

```
public class ReqUtility
{
    HttpServletRequest request;

    public ReqUtility(HttpServletRequest _req)
        throws ServletException, IOException
```

```

    {
        request = _req;
    }

    public Action getAction()
    {
        /* Utilice factory para crear una acción basada en parámetros de
           solicitud */
        String action = (String) request.getParameter("action");
        return ActionFactory.createAction(action);
    }
}

```

La estrategia de ayuda de solicitud le proporciona un objeto de solicitud, pero para procesar la solicitud necesita proporcionar objetos de comando, o acciones, con el fin de ejecutar la lógica empresarial y obtener el modelo y la visualización necesarios para completar la solicitud. En las siguientes secciones completará la imagen y echará un vistazo al resto del armazón de administración de solicitudes.

Estrategia de controlador y comando

Un buen armazón de administración de solicitudes facilita la extensión de la aplicación sin interferir con el controlador o el propio código de administración de solicitudes. Para conseguir este nivel de distinción (de separación), necesita un mecanismo flexible para añadir nuevos comportamientos a su código de administración de solicitudes. Puede realizar esto mediante un patrón *Factory*. Este patrón tiene sus raíces en el mundo del diseño orientado a objetos. Toma instantáneas de los nuevos objetos cuando no se conoce su clase hasta el tiempo de ejecución. Recuerde que en la sección anterior comprobó cómo se usaba un *factory* dentro de la ayuda de solicitud:

```

String action = (String) request.getParameter("action");
return ActionFactory.createAction(action);

```

Esto le permite añadir nuevos comportamientos agregándolos en el *factory* en lugar de modificar el código de procesamiento de solicitud del servlet. Pero para hacer que esto funcione, necesita definir una interfaz para nuestros objetos de comando. Me gustaría referirme a los objetos de comando como acciones, de modo que crearé una interfaz llamada *Action*. Codificando para una interfaz en lugar de a una implementación de clase específica, su código de administración de solicitudes nunca deberá cambiar cuando se añada un nuevo comportamiento a la aplicación cuando cualquier nuevo comportamiento implemente la interfaz *Action*. Aquí tiene los métodos básicos que define la interfaz *Action*:

```

/* Ejecutar lógica empresarial */
public boolean execute(HttpServletRequest _req, HttpServletResponse _res)
    throws ServletException, IOException;

```

Ponle la
escalabilidad →

Action

```

    /* Devolver en nombre de página (y la ruta de acceso) para mostrar la
       visualización */
    public String getView();

    /* Devolver un JavaBean que contiene el modelo (datos) */
    public Object getModel();

```

Estos métodos, al ser implementados, ejecutarán la lógica empresarial, devolverán el nombre de la página JSP para ser usada como visualización y devolverán un modelo para ser usado por JSP para generar su salida final. Para añadir un nuevo comportamiento a la aplicación, simplemente necesita crear una clase que implemente la interfaz *Action* y proporcione implementaciones para cada uno de los métodos definidos por la interfaz.

El propio *factory* se implementa como una clase abstracta. Haga esto ya que *factory* nunca debería ser instanciado. Su único propósito es devolver objetos que implementen la interfaz *Action*. El único método que puede definir en *factory* es un método estático que toma un parámetro *String* y le indica qué acción debe devolver. Entonces, el método instancia el objeto apropiado y lo devuelve al programa ejecutable (en este caso, la ayuda de solicitud). Observe el aspecto que tendría *factory* en este caso:

Solo permite ejecución de métodos específicos

```

public abstract class ActionFactory {
    public static Action createAction(String _action) {
        /* Return Action object */
        if (_action.equals("x")) {
            return new xAction();
        }
        if (_action.equals("y")) {
            return new yAction();
        }
        return null;
    }
}

```

XAction extends Action

public class AccionRef extends Action

El único problema para generar objetos de acción es que las nuevas acciones necesitan el código para volver a ser compiladas. Un modo más indicado sería obtener una lista de acciones de un archivo XML o pasar al método un nombre de clase completamente certificado y entonces utilizar *Java Reflection* para instanciar el objeto. En el capítulo 11 nos ocuparemos de ello.

Estrategias de mapeado de recursos

Una vez está en su sitio el armazón de administración de solicitudes, su aplicación necesita una estrategia para asignar acciones a las solicitudes. Hay diversas estrategias que cumplen esta necesidad. *J2EE Patterns Catalog* define la estrategia

Physical Resource Mapping, la estrategia *Logical Resource Mapping* y la estrategia *Multiplexed Resource Mapping*. La estrategia *Physical Resource Mapping* se ocupa de realizar solicitudes a un nombre de recurso físico específico como `http://myserver/myapp/servlet/controller`. La estrategia *Logical Resource Mapping*, por otra parte, mapea nombres lógicos hacia recursos físicos utilizando un archivo de configuración. Un ejemplo de esto sería `http://myserver/myapp/controller`. Esto apuntaría hacia un recurso específico en el archivo de configuración. Si deseaba modificar el recurso actuando como su controlador, simplemente volvería a mapear el nombre lógico en el archivo de configuración. Puede configurar esos mapeados en el archivo `web.xml` de su aplicación Web.

Todavía necesita una manera de asociar una acción específica con la solicitud. Puede hacer esto usando la estrategia *Multiplexed Resource Mapping*. Se trata de una extensión de la estrategia *Logical Resource Mapping* dado que utiliza un nombre lógico para definir el controlador, pero añade otro componente para asociar una acción a la solicitud. El mapeado lógico puede ser un nombre lógico específico o bien un patrón en particular que se asocie a un recurso físico. Por ejemplo, `http://myserver/myapp/login.c` podría ser mapeado hacia un controlador especificando en el archivo de configuración que todas las solicitudes con la extensión `.c` fueran al servlet controlador. En este caso, la parte izquierda de la solicitud, `login`, podría reservarse para especificar la acción.

O bien, su solicitud sencillamente podría ser `http://myserver/myapp/controller?action=login`. De este modo se asociaría el controlador del nombre lógico con el controlador del servlet y se pasaría a un parámetro llamado `action` indicando al servlet que genere una acción `login`. Se trata del modo más común y sencillo de mapear solicitudes hacia acciones.

Aplicar el patrón Front Controller

Para aplicar este patrón, vamos a rellenar las partes en que se divide un armazón de administración de solicitudes que comenzó a diseñar en las secciones anteriores. Entonces volverá a la aplicación de encuesta creada en el capítulo 5 y la volverá a diseñar utilizando su nuevo controlador y el armazón de administración de solicitudes.

Volver a visitar MVC: un ejemplo

En el capítulo 5, creó una aplicación de encuesta para ilustrar el armazón MVC. Aunque se trató de una gran mejora respecto a una solución JSP estándar, hay muchas oportunidades más de mejora. Específicamente, necesita ser un tanto más extensible. Si la aplicación fuera a crecer con el paso del tiempo, como suele pasar, se volvería complejo y difícil añadir un nuevo comportamiento. Es aquí donde puede resultar de gran ayuda el armazón de administración de solicitudes. Como debería

recordar, el ejemplo del capítulo 5 trató con una sencilla pantalla de conexión, después un formulario de entrada de datos y, finalmente, una pantalla de confirmación. Para este ejemplo, primero regresará a cada uno de los pasos del armazón de administración de solicitudes, después añadirá los comportamientos específicos necesarios para la aplicación y terminará modificando las páginas JSP para usar el nuevo controlador.

Crear la ayuda de la solicitud

Comenzaremos creando el objeto de ayuda de la solicitud. Con este propósito, implementará la mínima funcionalidad necesaria por un ayudante de solicitud. El método `getAction` utilizará el `ActionFactory` que creará posteriormente para obtener una instancia del objeto de acción adecuado. El listado 7.1 muestra el aspecto del ayudante de solicitud.

Listado 7.1. `ReqUtil.java`

```
package jspbook.ch7; package controlador

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

public class ReqUtility {

    HttpServletRequest request;

    public ReqUtility(HttpServletRequest _req)
        throws ServletException, IOException
    {
        request = _req;
    }

    public Action getAction()
    {
        /* Utilice factory para crear una acción basándose en parámetros de
         * solicitud */
        String action = (String) request.getParameter("action");
        return ActionFactory.createAction(action);
    }
}
```

AyudaSolicitud.java

Definir la interfaz Action

Antes de crear `factory`, echemos un vistazo a los objetos de acción que generará. Cada objeto de acción implementará la interfaz `Action`. Esto permite al armazón de administración de solicitudes permanecer invariable mientras el usuario añade nuevos comportamientos, pues trata con la interfaz `Action` en lugar de con cualquier objeto específico.

La interfaz Action define tres métodos centrales y un método de ayuda. Los métodos centrales son execute, getView y getModel. Además de todo ello, añadió un método de ayuda, setDatabase, para permitirle transmitir una conexión a base de datos existente al objeto. Esto le permite compartir de un modo sencillo recursos de base de datos. El método execute, al ser implementado, realizará cualquier lógica empresarial necesaria para tramitar la solicitud. Los métodos getView y getModel se usan para devolver la página y los datos necesarios para presentar los resultados de la acción. El listado 7.2 muestra el aspecto de esta interfaz.

Listado 7.2. Action.java

```
package jspbook.ch7;

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

public interface Action {

    /* Establecer conexión a base de datos */
    public void setDatabase(Connection _db);

    /* Ejecutar lógica empresarial */
    public boolean execute(HttpServletRequest _req, HttpServletResponse _res)
        throws ServletException, IOException;

    /* Devolver el nombre de página (y su ruta de acceso) para mostrar la
       visualización */
    public String getView();

    /* Devolver un JavaBean que contiene el modelo (datos) */
    public Object getModel();
}
```

Crear factory action

El factory utilizado para instanciar sus objetos Action es una clase abstracta con un único método estático. Esta clase es abstracta debido a que el usuario nunca desea que sea instanciada. El único método estático, createAction, comprueba el nombre de la acción que se le ha asignado y después instancia el objeto adecuado, devolviendo un objeto de tipo Action. Vea el aspecto del factory en el listado 7.3.

Listado 7.3. ActionFactory.java

```
package jspbook.ch7;

abstract class ActionFactory {

    public static Action createAction(String _action)
    {
```

```
    /* Devuelve un objeto Action */
    if (_action.equals("login")) {
        return new LoginAction();
    }
    if (_action.equals("submit")) {
        return new SubmitAction();
    }
    return null;
}
```

Implementar los comportamientos específicos de la aplicación

La gran diferencia entre este ejemplo y el del capítulo 5 es que el usuario implementa la lógica de la aplicación como acciones diferentes en lugar de dentro del código del controlador. Existen dos acciones que necesita crear, una acción login y una submit. Esencialmente tratará el mismo código del ejemplo anterior dentro del método execute de cada acción. El método execute realiza su procesamiento y actualiza las variables de clase con los datos necesarios. En caso de la acción login, almacene el identificador de usuario y el estado de la acción login. El método getModel ensambla un objeto CustomerBean y actualiza esos campos. Este bean será etiquetado en los atributos de solicitud y se utilizará para la visualización de la página siguiente. El método getView devuelve el nombre de la página que se va a mostrar a continuación. Esto se determina dentro del método execute y se almacena localmente. El listado 7.4 muestra el aspecto que presenta la acción login.

Listado 7.4. LoginAction.java

```
package jspbook.ch7;

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

import jspbook.ch7.CustomerBean;

public class LoginAction implements Action {

    private String view;
    private Connection dbCon;
    private String status;
    private String uid, pwd;

    public LoginAction() {}
```

```

/* Establecer conexión a la base de datos */
public void setDatabase(Connection _db)
{
    dbCon = _db;
}

/* Ejecutar lógica empresarial */
public boolean execute(HttpServletRequest _req, HttpServletResponse _res)
throws ServletException, IOException
{
    uid = (_String) _req.getParameter("UID");
    pwd = (_String) _req.getParameter("PWD");

    /* Validar usuario */
    if (authenticate(uid, pwd)) {
        status = "success";
        view = "/WEB-INF/jsp/ch7/census.jsp";
    }
    else {
        status = "failed";
        view = "/ch7/login.jsp";
    }
    return true;
}

/* Devolver el nombre de página (y su ruta de acceso) para mostrar la
   visualización */
public String getView()
{
    return view;
}

/* Devolver un JavaBean que contiene el modelo (datos) */
public Object getModel()
{
    /* Utilizar el JavaBean para devolver el estado de la conexión */

    CustomerBean cBean = new CustomerBean();
    cBean.setUid(uid);
    cBean.setLoginStatus(status);
    return cBean;
}

/* Comprobar si es válido el usuario */
private boolean authenticate(String _uid, String _pwd)
{
    ResultSet rs = null;
    try {
        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());
    }
    return false;
}
}

```

La acción submit es similar a la acción login. La única diferencia está en el código de dentro del método execute. Graba los datos de la encuesta y actualiza un campo de estado situado dentro de la clase. El CustomerBean se actualiza dentro del método getModel y la siguiente página se define en el método getView. El listado 7.5 muestra el aspecto de la acción submit.

Listado 7.5. SubmitAction.java

```

package jspbook.ch7;

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

import jspbook.ch7.CustomerBean;

public class SubmitAction implements Action {

    private String view;
    private Connection dbCon;
    private String status;

    public SubmitAction() {}

    /* Establecer conexión a la base de datos */
    public void setDatabase(Connection _db)
    {
        dbCon = _db;
    }

    /* Ejecutar lógica empresarial */
    public boolean execute(HttpServletRequest _req, HttpServletResponse _res)
        throws ServletException, IOException
    {
        /* Enviar datos de encuesta */
        if (recordSurvey(_req)) {
            status = "success";
            view = "/WEB-INF/jsp/ch7/thankyou.jsp";
        }
        else {
            status = "failed";
            view = "/WEB-INF/jsp/ch7/census.jsp";
        }
    }
}

```

```

        return true;
    }

    /* Devolver el nombre de página (y su ruta de acceso) para mostrar la
       visualización */
    public String getView()
    {
        return view;
    }

    /* Devolver un JavaBean que contiene el modelo (datos) */
    public Object getModel()
    {
        /* Return the status of the action */
        CustomerBean cBean = new CustomerBean();
        cBean.setSubmitStatus(status);
        return cBean;
    }

    /* Utilizar CustomerBean, grabar los datos */
    public boolean recordSurvey(HttpServletRequest _req)
    {
        CustomerBean cBean = new CustomerBean();
        cBean.populateFromParms(_req);
        return cBean.submit(dbCon);
    }
}

```

Ahora necesita realizar algunos cambios en el CustomerBean que usó anteriormente en el capítulo 5 para adaptarse al nuevo armazón. Estos cambios tienen que ver con el modo en que recupera el identificador de usuario y se definen dentro del bean. También necesita añadir algunos campos de estado. Este bean se recupera dentro de las páginas JSP, y sus campos de estado se comprueban antes de mostrar cada pantalla. El listado 7.6 nos muestra el aspecto que presenta una vez realizados esos cambios.

Listado 7.6. CustomerBean.java

```

package jspbook.ch7;

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

public class CustomerBean implements java.io.Serializable {

    /* Variables de miembro */
    private String lname, fname, sex;
    private int age, children;
    private boolean spouse, smoker;

    /* Variables de ayuda */;
    private String uid;
}

```

```

private String loginStatus, submitStatus;
/* Constructor */
public CustomerBean() {
    /* Initialize properties */
    setLname("");
    setFname("");
    setSex("");
    setAge(0);
    setChildren(0);
    setSpouse(false);
    setSmoker(false);
}

public void populateFromParms(HttpServletRequest _req) {
    // Publicar las propiedades del bean desde los parámetros de solicitud
    setLname(_req.getParameter("lname"));
    setFname(_req.getParameter("fname"));
    setSex(_req.getParameter("sex"));
    setAge(Integer.parseInt(_req.getParameter("age")));
    setChildren(Integer.parseInt(_req.getParameter("children")));
    setSpouse(_req.getParameter("married").equals("Y")) ? true : false;
    setSmoker(_req.getParameter("smoker").equals("Y")) ? true : false;
    setUid(_req.getParameter("uid"));
}

/* Métodos Accessor */

/* Nombre */
public void setLname(String _lname) {lname = _lname;}
public String getLname() {return lname;}

/* Apellido */
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;}

```

```

/* Variables de ayuda */
public void setUid(String _uid) {uid = _uid;}
public String getUid() {return uid;}

public void setLoginStatus(String _status) {loginStatus = _status;}
public String getLoginStatus() {return loginStatus;}

public void setSubmitStatus(String _status) {submitStatus = _status;}
public String getSubmitStatus() {return submitStatus;}

public boolean submit(Connection _dbCon) {

    Statement s = null;
    ResultSet rs = null;
    String custId = "";
    StringBuffer sql = new StringBuffer(256);

    try {
        // Comprobar si existe el cliente (utilice UID para obtener el 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("lname='").append(lname).append("', ");
            sql.append("fname='").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("where id='").append(custId).append("'");
        } else {
            // Insertar registro
            sql.append("INSERT INTO customer VALUES(");
            sql.append(custId).append(",");
            sql.append(lname).append(",");
            sql.append(fname).append(",");
            sql.append(age).append(",");
            sql.append(sex).append(",");
            sql.append((spouse) ? "Y" : "N").append(",");
            sql.append(children).append(",");
            sql.append((smoker) ? "Y" : "N").append(")");
        }
        s.executeUpdate(sql.toString());
    }
    catch (SQLException e) {
        System.out.println("Error saving customer: " + custId + " : "
                           + e.toString());
    }
}

```

Puedes implementar cambios simples

```

        return false;
    }
    return true;
}
}

```

Crear el controlador

Finalmente, creará el controlador, que se implementa como servlet. Dentro del método doPost, la solicitud se agrega al ayudante de solicitud (ReqUtil). Entonces se crea un objeto action (de acción), se ejecuta la misma y finalmente el modelo se adjunta a la solicitud. Para terminar, la solicitud se envía a la página JSP para su visualización. Observe el listado 7.7, que contiene el código de nuestro controlador.

Listado 7.7. Controller.java

```

package jspbook.ch7;

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

public class Controller extends HttpServlet {

    /* Compartir conexión a base de datos */
    private Connection dbCon;

    public void init()
    {
        /* Inicializar recursos compartidos */

        try {
            Context initCtx = new InitialContext();
            Context envCtx = (Context) initCtx.lookup("java:comp/env");
            DataSource 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 connecting to the
                               database.");
            System.out.println(e.toString());
        }
    }
}

```

```

    }

    public void doGet(HttpServletRequest _req, HttpServletResponse _res)
        throws ServletException, IOException
    {
        /* Enviar al método doPost */
        doPost(_req, _res);
    }

    public void doPost(HttpServletRequest _req, HttpServletResponse _res)
        throws ServletException, IOException
    {
        /* Asignar el objeto de solicitud al ayudante */
        ReqUtility reqUtil = new ReqUtility(_req);

        /* Crear un objeto Action basándose en los parámetros de solicitud */
        Action action = reqUtil.getAction();

        /* Transferir la conexión a base de datos a la acción */
        action.setDatabase(dbCon);

        /* Ejecutar lógica empresarial */
        if (action.execute(_req, _res)) {

            /* Obtener la visualización adecuada para la acción */
            String view = action.getView();

            /* Agregar el modelo a los atributos requeridos */
            _req.setAttribute("model", action.getModel());

            /* Enviar la solicitud a la visualización dada */
            RequestDispatcher dispatcher = _req.getRequestDispatcher(view);
            dispatcher.forward(_req, _res);
        }
    }

    public void destroy()
    {
        /* Limpiar recursos compartidos */

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

```

Modificar las páginas JSP

Las páginas JSP son esencialmente lo mismo de lo que se habló en el capítulo 5. La única diferencia estriba en cómo recuperan el estado de la acción `login` o de la acción `submit`. Ahora, recuperan el `CustomerBean` de los atributos de solicitud y entonces asumen el estado del bean. Los listados 7.8, 7.9 y 7.10 muestran el aspecto de las páginas JSP una vez realizados estos cambios.

Listado 7.8. login.jsp

```

<%@ page
    import="jspbook.ch7.CustomerBean"
    errorPage="myError.jsp?from=login.jsp"
%>

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

<body bgcolor="#FFFF99">

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

<form method="post" action="Controller?action=login">

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

<p>&ampnbsp</p>

<%    CustomerBean custBean = (CustomerBean) request.getAttribute("model");
    if (custBean != null) {
        String status = custBean.getLoginStatus();
        if (status != null && status.equals("failed")) {
%>
<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>
        <td><input type="text" name="UID"></td>
    </tr>
    <tr>

```

```

<td><font face="Arial, Helvetica, sans-serif" size="2">Password:</font></td>
<td><input type="password" name="PWD"></td>
</tr>
<tr align="center">
    <td colspan="2"><input type="submit" name="Submit" value="Login"></td>
</tr>
</table>

</form>

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

</body>
</html>

```

Listado 7.9. census.jsp

```

<!-- Directivas JSP --&gt;
&lt;%@ page
    import="jspbook.ch7.CustomerBean"
    errorPage="myError.jsp?from=census.jsp"
%&gt;

&lt;html&gt;
&lt;head&gt;
    &lt;title&gt;Insurance Quoting System&lt;/title&gt;
&lt;/head&gt;
&lt;body bgcolor="#FFFF99"&gt;

&lt;basefont face="Arial"&gt;

&lt;%@ include file="/ch7/myHeader.html" %&gt;

&lt;form action="Controller?action=submit" method="post"&gt;
&lt;br&gt;&lt;br&gt;

&lt;%    CustomerBean custBean = (CustomerBean) request.getAttribute("model");
    String uid = "";
    if (custBean != null) {
        String status = custBean.getSubmitStatus();
        uid = custBean.getUid();
        if (status != null &amp;&amp; status.equals("failed")) {
%&gt;
&lt;center&gt;
    &lt;font color="#ff0000"&gt;Error recording survey data, please try again
        &lt;/font&gt;
&lt;/center&gt;
&lt;br&gt;&lt;br&gt;
&lt;%    }
%&gt;
</pre>

```

```

<center><b>Enter personal information:</b></center>
<br><br>
<input type='hidden' name='uid' value='<%= uid %>'>
<table cellspacing="2" cellpadding="2" border="0" align="center">
<tr>
    <td align="right">First Name:</td>
    <td><input type="Text" name="fname" size="10"></td>
</tr>
<tr>
    <td align="right">Last Name:</td>
    <td><input type="Text" name="lname" size="10"></td>
</tr>
<tr>
    <td align="right">Age:</td>
    <td><input type="Text" name="age" size="2"></td>
</tr>
<tr>
    <td align="right">Sex:</td>
    <td>
        <input type="radio" name="sex" value="M" checked>Male</input>
        <input type="radio" name="sex" value="F">Female</input>
    </td>
</tr>
<tr>
    <td align="right">Married:</td>
    <td><input type="Text" name="married" size="2"></td>
</tr>
<tr>
    <td align="right">Children:</td>
    <td><input type="Text" name="children" size="2"></td>
</tr>
<tr>
    <td align="right">Smoker:</td>
    <td><input type="Text" name="smoker" size="2"></td>
</tr>
<tr>
    <td colspan="2" align="center"><input type="Submit" value="Submit">
        </td>
</tr>
</table>

<br><br>

</form>
<%@ include file="/ch7/myFooter.html" %>

</body>
</html>

```

Listado 7.10. thankyou.jsp

```

<!-- Directivas JSP --&gt;
&lt;%@ page
    errorPage="myError.jsp?from=thankyou.jsp"
%&gt;
</pre>

```

```

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

<body bgcolor="#FFFF99">

<basefont face="Arial">

<%@ include file="/ch7/myHeader.html" %>

<br><br>

<center>
Your survey answers have been recorded. Thank you for participating in this
survey.
</center>

<br><br>

<%@ include file="/ch7/myFooter.html" %>

</body>
</html>

```

Utilizar filtros con un Front Controller

Una gran ventaja de implementar el patrón *Front Controller* radica en que centraliza la administración de solicitudes. Esto crea la posibilidad de realizar el procesamiento común en la solicitud y la respuesta. Esta tarea puede realizarse dentro del propio controlador, pero en muchos casos es mejor utilizar filtros (repase el capítulo 6 si desea una completa información sobre los filtros). Los filtros le permiten conectar y desconectar funcionalidad adicional sin interferir con el armazón de administración de solicitudes del mismo modo que su *ActionFactory* le permitía añadir un nuevo comportamiento a la aplicación sin necesidad de realizar modificaciones al servlet controlador.

Más adelante en este libro, ensamblará todas las partes que hemos creado y creará un armazón completo. Entonces desarrollará una aplicación completa utilizando el armazón. Dentro del mismo, usará filtros para administrar la autenticación de usuarios.

Resumen

Probablemente este capítulo ha sido uno de los más importantes de este libro. Contiene las bases necesarias para poder construir un sólido armazón de aplicación. En los capítulos siguientes comprobará la manera de completar el mecanismo

de administración de solicitudes que se ha creado en este capítulo con el fin de dar lugar a un robusto y extensible armazón para generar aplicaciones Web. En el capítulo 11 mejorará este armazón de administración de solicitudes al incluir también características tales como activación de conexiones, administración de errores y mantenimiento de bases de datos.