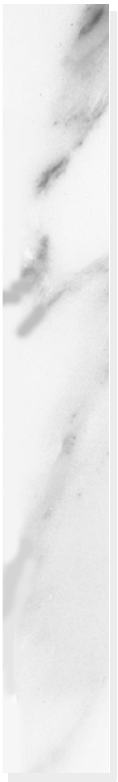
INTEGRATING SERVLETS AND JSP

The MVC Architecture



Topics in This Chapter

- Obtaining a `RequestDispatcher`
- Forwarding requests from servlets to dynamic resources
- Forwarding requests from servlets to static resources
- Using servlets to set up beans for use by JSP pages
- An on-line travel agency combining servlets and JSP
- Including JSP output in servlets
- A servlet that shows the raw HTML output of JSP pages
- Using `jsp:forward` to forward requests from JSP pages

Chapter 15

Servlets are great when your application requires a lot of real programming to accomplish its task. As you've seen elsewhere in the book, servlets can manipulate HTTP status codes and headers, use cookies, track sessions, save information between requests, compress pages, access databases, generate GIF images on-the-fly, and perform many other tasks flexibly and efficiently. But, generating HTML with servlets can be tedious and can yield a result that is hard to modify. That's where JSP comes in; it lets you separate much of the presentation from the dynamic content. That way, you can write the HTML in the normal manner, even using HTML-specific tools and putting your Web content developers to work on your JSP documents. JSP expressions, scriptlets, and declarations let you insert simple Java code into the servlet that results from the JSP page, and directives let you control the overall layout of the page. For more complex requirements, you can wrap up Java code inside beans or define your own JSP tags.

Great. We have everything we need, right? Well, no, not quite. The assumption behind a JSP document is that it provides a *single* overall presentation. What if you want to give totally different results depending on the data that you receive? Beans and custom tags, although extremely powerful and flexible, don't overcome the limitation that the JSP page defines a relatively fixed top-level page appearance. The solution is to use *both* servlets and JavaServer Pages. If you have a complicated application that may require several substantially different presentations, a servlet can handle the initial request,

partially process the data, set up beans, then forward the results to one of a number of different JSP pages, depending on the circumstances. In early JSP specifications, this approach was known as the *model 2* approach to JSP. Rather than completely forwarding the request, the servlet can generate part of the output itself, then include the output of one or more JSP pages to obtain the final result.

15.1 Forwarding Requests

The key to letting servlets forward requests or include external content is to use a `RequestDispatcher`. You obtain a `RequestDispatcher` by calling the `getRequestDispatcher` method of `ServletContext`, supplying a URL relative to the server root. For example, to obtain a `RequestDispatcher` associated with `http://yourhost/presentations/presentation1.jsp`, you would do the following:

```
String url = "/presentations/presentation1.jsp";
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher(url);
```

Once you have a `RequestDispatcher`, you use `forward` to completely transfer control to the associated URL and use `include` to output the associated URL's content. In both cases, you supply the `HttpServletRequest` and `HttpServletResponse` as arguments. Both methods throw `ServletException` and `IOException`. For example, Listing 15.1 shows a portion of a servlet that forwards the request to one of three different JSP pages, depending on the value of the `operation` parameter. To avoid repeating the `getRequestDispatcher` call, I use a utility method called `gotoPage` that takes the URL, the `HttpServletRequest` and the `HttpServletResponse`; gets a `RequestDispatcher`; and then calls `forward` on it.

Using Static Resources

In most cases, you forward requests to a JSP page or another servlet. In some cases, however, you might want to send the request to a static HTML page. In an e-commerce site, for example, requests that indicate that the user does not have a valid account name might be forwarded to an account application page that uses HTML forms to gather the requisite information. With `GET` requests, forwarding requests to a static HTML page is perfectly legal and requires no special syntax; just supply the address of the HTML page as the

Listing 15.1 Request Forwarding Example

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
  String operation = request.getParameter("operation");
  if (operation == null) {
    operation = "unknown";
  }
  if (operation.equals("operation1")) {
    gotoPage("/operations/presentation1.jsp",
             request, response);
  } else if (operation.equals("operation2")) {
    gotoPage("/operations/presentation2.jsp",
             request, response);
  } else {
    gotoPage("/operations/unknownRequestHandler.jsp",
             request, response);
  }
}

private void gotoPage(String address,
                      HttpServletRequest request,
                      HttpServletResponse response)
    throws ServletException, IOException {
  RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher(address);
  dispatcher.forward(request, response);
}
```

argument to `getRequestDispatcher`. However, since forwarded requests use the same request method as the original request, `POST` requests cannot be forwarded to normal HTML pages. The solution to this problem is to simply rename the HTML page to have a `.jsp` extension. Renaming `somefile.html` to `somefile.jsp` does not change its output for `GET` requests, but `somefile.html` cannot handle `POST` requests, whereas `somefile.jsp` gives an identical response for both `GET` and `POST`.

Supplying Information to the Destination Pages

To forward the request to a JSP page, a servlet merely needs to obtain a `RequestDispatcher` by calling the `getRequestDispatcher` method of `ServletContext`, then call `forward` on the result, supplying the `HttpServletRequest` and `HttpServletResponse` as arguments. That's fine as far as it goes, but this approach requires the destination page to read the

information it needs out of the `HttpServletRequest`. There are two reasons why it might not be a good idea to have the destination page look up and process all the data itself. First, complicated programming is easier in a servlet than in a JSP page. Second, multiple JSP pages may require the same data, so it would be wasteful for each JSP page to have to set up the same data. A better approach is for the original servlet to set up the information that the destination pages need, then store it somewhere that the destination pages can easily access.

There are two main places for the servlet to store the data that the JSP pages will use: in the `HttpServletRequest` and as a bean in the location specific to the `scope` attribute of `jsp:useBean` (see Section 13.4, "Sharing Beans").

The originating servlet would store arbitrary objects in the `HttpServletRequest` by using

```
request.setAttribute("key1", value1);
```

The destination page would access the value by using a JSP scripting element to call

```
Type1 value1 = (Type1)request.getAttribute("key1");
```

For complex values, an even better approach is to represent the value as a bean and store it in the location used by `jsp:useBean` for shared beans. For example, a `scope` of `application` means that the value is stored in the `ServletContext`, and `ServletContext` uses `setAttribute` to store values. Thus, to make a bean accessible to all servlets or JSP pages in the server or Web application, the originating servlet would do the following:

```
Type1 value1 = computeValueFromRequest(request);
getServletContext().setAttribute("key1", value1);
```

The destination JSP page would normally access the previously stored value by using `jsp:useBean` as follows:

```
<jsp:useBean id="key1" class="Type1" scope="application" />
```

Alternatively, the destination page could use a scripting element to explicitly call `application.getAttribute("key1")` and cast the result to `Type1`.

For a servlet to make data specific to a user session rather than globally accessible, the servlet would store the value in the `HttpSession` in the normal manner, as below:

```
Type1 value1 = computeValueFromRequest(request);
HttpSession session = request.getSession(true);
session.putValue("key1", value1);
```

The destination page would then access the value by means of

```
<jsp:useBean id="key1" class="Type1" scope="session" />
```

The Servlet 2.2 specification adds a third way to send data to the destination page when using GET requests: simply append the query data to the URL. For example,

```
String address = "/path/resource.jsp?newParam=value";
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher(address);
dispatcher.forward(request, response);
```

This technique results in an *additional* request parameter of newParam (with a value of value) being added to whatever request parameters already existed. The new parameter is added to the beginning of the query data so that it will replace existing values if the destination page uses getParameter (use the first occurrence of the named parameter) rather than getParameterValues (use all occurrences of the named parameter).

Interpreting Relative URLs in the Destination Page

Although a servlet can forward the request to arbitrary locations on the same server, the process is quite different from that of using the sendRedirect method of HttpServletResponse (see Section 6.1). First, sendRedirect requires the client to reconnect to the new resource, whereas the forward method of RequestDispatcher is handled completely on the server. Second, sendRedirect does not automatically preserve all of the request data; forward does. Third, sendRedirect results in a different final URL, whereas with forward, the URL of the original servlet is maintained.

This final point means that, if the destination page uses relative URLs for images or style sheets, it needs to make them relative to the server root, not to the destination page's actual location. For example, consider the following style sheet entry:

```
<LINK REL=STYLESHEET
      HREF="my-styles.css"
      TYPE="text/css">
```

If the JSP page containing this entry is accessed by means of a forwarded request, my-styles.css will be interpreted relative to the URL of the originating servlet, not relative to the JSP page itself, almost certainly resulting in an error. The solution is to give the full server path to the style sheet file, as follows:

```
<LINK REL=STYLESHEET
      HREF="/path/my-styles.css"
      TYPE="text/css">
```

The same approach is required for addresses used in `<IMG SRC=...>` and `<A HREF=...>`.

Alternative Means of Getting a RequestDispatcher

In servers that support version 2.2 of the servlet specification, there are two additional ways of obtaining a `RequestDispatcher` besides the `getRequestDispatcher` method of `ServletContext`.

First, since most servers let you give explicit names to servlets or JSP pages, it makes sense to access them by name rather than by path. Use the `getNamedDispatcher` method of `ServletContext` for this task.

Second, you might want to access a resource by a path relative to the current servlet's location, rather than relative to the server root. This approach is not common when servlets are accessed in the standard manner (`http://host/servlet/ServletName`), because JSP files would not be accessible by means of `http://host/servlet/...` since that URL is reserved especially for servlets. However, it is common to register servlets under another path, and in such a case you can use the `getRequestDispatcher` method of `HttpServletRequest` rather than the one from `ServletContext`. For example, if the originating servlet is at `http://host/travel/TopLevel`,

```
getServletContext().getRequestDispatcher("/travel/cruises.jsp")
```

could be replaced by

```
request.getRequestDispatcher("cruises.jsp");
```

15.2 Example: An On-Line Travel Agent

Consider the case of an on-line travel agent that has a quick-search page, as shown in Figure 15–1 and Listing 15.2. Users need to enter their e-mail address and password to associate the request with their previously established customer account. Each request also includes a trip origin, trip destination, start date, and end date. However, the action that will result will vary substan-

tially based upon the action requested. For example, pressing the "Book Flights" button should show a list of available flights on the dates specified, ordered by price (see Figure 15–1). The user's real name, frequent flyer information, and credit card number should be used to generate the page. On the other hand, selecting "Edit Account" should show any previously entered customer information, letting the user modify values or add entries. Likewise, the actions resulting from choosing "Rent Cars" or "Find Hotels" will share much of the same customer data but will have a totally different presentation.

To accomplish the desired behavior, the front end (Listing 15.2) submits the request to the top-level travel servlet shown in Listing 15.3. This servlet looks up the customer information (see Listings 15.5 through 15.9), puts it in the `HttpSession` object associating the value (of type `coreservlets.TravelCustomer`) with the name `customer`, and then forwards the request to a different JSP page corresponding to each of the possible actions. The destination page (see Listing 15.4 and the result in Figure 15–2) looks up the customer information by means of

```
<jsp:useBean id="customer"
             class="coreservlets.TravelCustomer"
             scope="session" />
```

then uses `jsp:getProperty` to insert customer information into various parts of the page. You should note two things about the `TravelCustomer` class (Listing 15.5).

First, the class spends a considerable amount of effort making the customer information accessible as plain strings or even HTML-formatted strings through simple properties. Almost every task that requires any substantial amount of programming at all is spun off into the bean, rather than being performed in the JSP page itself. This is typical of servlet/JSP integration—the use of JSP does not *entirely* obviate the need to format data as strings or HTML in Java code. Significant up-front effort to make the data conveniently available to JSP more than pays for itself when multiple JSP pages access the same type of data.

Second, remember that many servers that automatically reload servlets when their class files change do not allow bean classes used by JSP to be in the auto-reloading directories. Thus, with the Java Web Server for example, `TravelCustomer` and its supporting classes must be in *`install_dir`*`/classes/coreservlets/`, not *`install_dir`*`/servlets/coreservlets/`. Tomcat 3.0 and the JSWDK 1.0.1 do not support auto-reloading servlets, so `TravelCustomer` can be installed in the normal location.

Figure 15–1 Front end to travel servlet (see Listing 15.2).

Figure 15–2 Result of travel servlet (Listing 15.3) dispatching request to `BookFlights.jsp` (Listing 15.4).

Listing 15.2 `/travel/quick-search.html`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Online Travel Quick Search</TITLE>
  <LINK REL=STYLESHEET
        HREF="travel-styles.css"
        TYPE="text/css">
</HEAD>
```

Listing 15.2 /travel/quick-search.html (continued)

```
<BODY>
<BR>
<H1>Online Travel Quick Search</H1>

<FORM ACTION="/servlet/coreservlets.Travel" METHOD="POST">
<CENTER>
Email address: <INPUT TYPE="TEXT" NAME="emailAddress"><BR>
Password: <INPUT TYPE="PASSWORD" NAME="password" SIZE=10><BR>
Origin: <INPUT TYPE="TEXT" NAME="origin"><BR>
Destination: <INPUT TYPE="TEXT" NAME="destination"><BR>
Start date (MM/DD/YY):
  <INPUT TYPE="TEXT" NAME="startDate" SIZE=8><BR>
End date (MM/DD/YY):
  <INPUT TYPE="TEXT" NAME="endDate" SIZE=8><BR>
<P>
<TABLE CELLSPACING=1>
<TR>
  <TH> <IMG SRC="airplane.gif" WIDTH=100 HEIGHT=29
                 ALIGN="TOP" ALT="Book Flight"> 
  <TH> <IMG SRC="car.gif" WIDTH=100 HEIGHT=31
                 ALIGN="MIDDLE" ALT="Rent Car"> 
  <TH> <IMG SRC="bed.gif" WIDTH=100 HEIGHT=85
                 ALIGN="MIDDLE" ALT="Find Hotel"> 
  <TH> <IMG SRC="passport.gif" WIDTH=71 HEIGHT=100
                 ALIGN="MIDDLE" ALT="Edit Account"> 
<TR>
  <TH><SMALL>
      <INPUT TYPE="SUBMIT" NAME="flights" VALUE="Book Flight">
      </SMALL>
  <TH><SMALL>
      <INPUT TYPE="SUBMIT" NAME="cars" VALUE="Rent Car">
      </SMALL>
  <TH><SMALL>
      <INPUT TYPE="SUBMIT" NAME="hotels" VALUE="Find Hotel">
      </SMALL>
  <TH><SMALL>
      <INPUT TYPE="SUBMIT" NAME="account" VALUE="Edit Account">
      </SMALL>
</TABLE>
</CENTER>
</FORM>
<BR>
<P ALIGN="CENTER">
<B>Not yet a member? Get a free account
<A HREF="accounts.jsp">here</A>.</B></P>
</BODY>
</HTML>
```

Listing 15.3 Travel.java

```
package coreservlets;

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

/** Top-level travel-processing servlet. This servlet sets up
 *  the customer data as a bean, then forwards the request
 *  to the airline booking page, the rental car reservation
 *  page, the hotel page, the existing account modification
 *  page, or the new account page.
 */

public class Travel extends HttpServlet {
  private TravelCustomer[] travelData;

  public void init() {
    travelData = TravelData.getTravelData();
  }

  /** Since password is being sent, use POST only. However,
   *  the use of POST means that you cannot forward
   *  the request to a static HTML page, since the forwarded
   *  request uses the same request method as the original
   *  one, and static pages cannot handle POST. Solution:
   *  have the "static" page be a JSP file that contains
   *  HTML only. That's what accounts.jsp is. The other
   *  JSP files really need to be dynamically generated,
   *  since they make use of the customer data.
   */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    String emailAddress = request.getParameter("emailAddress");
    String password = request.getParameter("password");
    TravelCustomer customer =
      TravelCustomer.findCustomer(emailAddress, travelData);
    if ((customer == null) || (password == null) ||
        (!password.equals(customer.getPassword()))) {
      gotoPage("/travel/accounts.jsp", request, response);
    }
    // The methods that use the following parameters will
    // check for missing or malformed values.
    customer.setStartDate(request.getParameter("startDate"));
    customer.setEndDate(request.getParameter("endDate"));
    customer.setOrigin(request.getParameter("origin"));
```

Listing 15.3 `Travel.java` (continued)

```
    customer.setDestination(request.getParameter
                            ("destination"));
    HttpSession session = request.getSession(true);
    session.putValue("customer", customer);
    if (request.getParameter("flights") != null) {
      gotoPage("/travel/BookFlights.jsp",
               request, response);
    } else if (request.getParameter("cars") != null) {
      gotoPage("/travel/RentCars.jsp",
               request, response);
    } else if (request.getParameter("hotels") != null) {
      gotoPage("/travel/FindHotels.jsp",
               request, response);
    } else if (request.getParameter("cars") != null) {
      gotoPage("/travel/EditAccounts.jsp",
               request, response);
    } else {
      gotoPage("/travel/IllegalRequest.jsp",
               request, response);
    }
  }

  private void gotoPage(String address,
                        HttpServletRequest request,
                        HttpServletResponse response)
      throws ServletException, IOException {
    RequestDispatcher dispatcher =
      getServletContext().getRequestDispatcher(address);
    dispatcher.forward(request, response);
  }
}
```

Listing 15.4 BookFlights.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Best Available Flights</TITLE>
  <LINK REL=STYLESHEET
        HREF="/travel/travel-styles.css"
        TYPE="text/css">
</HEAD>

<BODY>
<H1>Best Available Flights</H1>
<CENTER>
<jsp:useBean id="customer"
             class="coreservlets.TravelCustomer"
             scope="session" />
Finding flights for
<jsp:getProperty name="customer" property="fullName" />
<P>
<jsp:getProperty name="customer" property="flights" />

<P>
<BR>
<HR>
<BR>
<FORM ACTION="/servlet/BookFlight">
<jsp:getProperty name="customer"
                 property="frequentFlyerTable" />
<P>
<B>Credit Card:</B>
<jsp:getProperty name="customer" property="creditCard" />
<P>
<INPUT TYPE="SUBMIT" NAME="holdButton" VALUE="Hold for 24 Hours">
<P>
<INPUT TYPE="SUBMIT" NAME="bookItButton" VALUE="Book It!">
</FORM>
</CENTER>

</BODY>
</HTML>
```

Listing 15.5 TravelCustomer.java

```
package coreservlets;

import java.util.*;
import java.text.*;

/** Describes a travel services customer. Implemented
 *  as a bean with some methods that return data in HTML
 *  format, suitable for access from JSP.
 */

public class TravelCustomer {
  private String emailAddress, password, firstName, lastName;
  private String creditCardName, creditCardNumber;
  private String phoneNumber, homeAddress;
  private String startDate, endDate;
  private String origin, destination;
  private FrequentFlyerInfo[] frequentFlyerData;
  private RentalCarInfo[] rentalCarData;
  private HotelInfo[] hotelData;

  public TravelCustomer(String emailAddress,
                        String password,
                        String firstName,
                        String lastName,
                        String creditCardName,
                        String creditCardNumber,
                        String phoneNumber,
                        String homeAddress,
                        FrequentFlyerInfo[] frequentFlyerData,
                        RentalCarInfo[] rentalCarData,
                        HotelInfo[] hotelData) {
    setEmailAddress(emailAddress);
    setPassword(password);
    setFirstName(firstName);
    setLastName(lastName);
    setCreditCardName(creditCardName);
    setCreditCardNumber(creditCardNumber);
    setPhoneNumber(phoneNumber);
    setHomeAddress(homeAddress);
    setStartDate(startDate);
    setEndDate(endDate);
    setFrequentFlyerData(frequentFlyerData);
    setRentalCarData(rentalCarData);
    setHotelData(hotelData);
  }
```

Listing 15.5 `TravelCustomer.java` (continued)

```
public String getEmailAddress() {
  return(emailAddress);
}

public void setEmailAddress(String emailAddress) {
  this.emailAddress = emailAddress;
}

public String getPassword() {
  return(password);
}

public void setPassword(String password) {
  this.password = password;
}

public String getFirstName() {
  return(firstName);
}

public void setFirstName(String firstName) {
  this.firstName = firstName;
}

public String getLastName() {
  return(lastName);
}

public void setLastName(String lastName) {
  this.lastName = lastName;
}

public String getFullName() {
  return(getFirstName() + " " + getLastName());
}

public String getCreditCardName() {
  return(creditCardName);
}

public void setCreditCardName(String creditCardName) {
  this.creditCardName = creditCardName;
}

public String getCreditCardNumber() {
  return(creditCardNumber);
}
```

Listing 15.5 `TravelCustomer.java` (continued)

```
public void setCreditCardNumber(String creditCardNumber) {
  this.creditCardNumber = creditCardNumber;
}

public String getCreditCard() {
  String cardName = getCreditCardName();
  String cardNum = getCreditCardNumber();
  cardNum = cardNum.substring(cardNum.length() - 4);
  return(cardName + " (XXXX-XXXX-XXXX-" + cardNum + ")");
}

public String getPhoneNumber() {
  return(phoneNumber);
}

public void setPhoneNumber(String phoneNumber) {
  this.phoneNumber = phoneNumber;
}

public String getHomeAddress() {
  return(homeAddress);
}

public void setHomeAddress(String homeAddress) {
  this.homeAddress = homeAddress;
}

public String getStartDate() {
  return(startDate);
}

public void setStartDate(String startDate) {
  this.startDate = startDate;
}

public String getEndDate() {
  return(endDate);
}

public void setEndDate(String endDate) {
  this.endDate = endDate;
}

public String getOrigin() {
  return(origin);
}
```

Listing 15.5 `TravelCustomer.java` (continued)

```
public void setOrigin(String origin) {
  this.origin = origin;
}

public String getDestination() {
  return(destination);
}

public void setDestination(String destination) {
  this.destination = destination;
}

public FrequentFlyerInfo[] getFrequentFlyerData() {
  return(frequentFlyerData);
}

public void setFrequentFlyerData(FrequentFlyerInfo[]
                                 frequentFlyerData) {
  this.frequentFlyerData = frequentFlyerData;
}

public String getFrequentFlyerTable() {
  FrequentFlyerInfo[] frequentFlyerData =
    getFrequentFlyerData();
  if (frequentFlyerData.length == 0) {
    return("<I>No frequent flyer data recorded.</I>");
  } else {
    String table =
      "<TABLE>\n" +
      "  <TR><TH>Airline<TH>Frequent Flyer Number\n";
    for(int i=0; i<frequentFlyerData.length; i++) {
      FrequentFlyerInfo info = frequentFlyerData[i];
      table = table +
              "<TR ALIGN=\"CENTER\">" +
              "<TD>" + info.getAirlineName() +
              "<TD>" + info.getFrequentFlyerNumber() + "\n";
    }
    table = table + "</TABLE>\n";
    return(table);
  }
}

public RentalCarInfo[] getRentalCarData() {
  return(rentalCarData);
}
```

Listing 15.5 `TravelCustomer.java` (continued)

```
public void setRentalCarData(RentalCarInfo[] rentalCarData) {
  this.rentalCarData = rentalCarData;
}

public HotelInfo[] getHotelData() {
  return(hotelData);
}

public void setHotelData(HotelInfo[] hotelData) {
  this.hotelData = hotelData;
}

// This would be replaced by a database lookup
// in a real application.

public String getFlights() {
  String flightOrigin =
    replaceIfMissing(getOrigin(), "Nowhere");
  String flightDestination =
    replaceIfMissing(getDestination(), "Nowhere");
  Date today = new Date();
  DateFormat formatter =
    DateFormat.getDateInstance(DateFormat.MEDIUM);
  String dateString = formatter.format(today);
  String flightStartDate =
    replaceIfMissing(getStartDate(), dateString);
  String flightEndDate =
    replaceIfMissing(getEndDate(), dateString);
  String [][] flights =
    { { "Java Airways", "1522", "455.95", "Java, Indonesia",
        "Sun Microsystems", "9:00", "3:15" },
      { "Servlet Express", "2622", "505.95", "New Atlanta",
        "New Atlanta", "9:30", "4:15" },
      { "Geek Airlines", "3.14159", "675.00", "JHU",
        "MIT", "10:02:37", "2:22:19" } };
  String flightString = "";
  for(int i=0; i<flights.length; i++) {
    String[] flightInfo = flights[i];
```

Listing 15.5 `TravelCustomer.java` (continued)

```
      flightString =
        flightString + getFlightDescription(flightInfo[0],
                                             flightInfo[1],
                                             flightInfo[2],
                                             flightInfo[3],
                                             flightInfo[4],
                                             flightInfo[5],
                                             flightInfo[6],
                                             flightOrigin,
                                             flightDestination,
                                             flightStartDate,
                                             flightEndDate);
    }
    return(flightString);
  }

  private String getFlightDescription(String airline,
                                      String flightNum,
                                      String price,
                                      String stop1,
                                      String stop2,
                                      String time1,
                                      String time2,
                                      String flightOrigin,
                                      String flightDestination,
                                      String flightStartDate,
                                      String flightEndDate) {
    String flight =
      "<P><BR>\n" +
      "<TABLE WIDTH=\"100%\"><TR><TH CLASS=\"COLORED\">\n" +
      "<B>" + airline + " Flight " + flightNum +
      " ($" + price + ")</B></TABLE><BR>\n" +
      "<B>Outgoing:</B> Leaves " + flightOrigin +
      " at " + time1 + " AM on " + flightStartDate +
      ", arriving in " + flightDestination +
      " at " + time2 + " PM (1 stop -- " + stop1 + ").\n" +
      "<BR>\n" +
      "<B>Return:</B> Leaves " + flightDestination +
      " at " + time1 + " AM on " + flightEndDate +
      ", arriving in " + flightOrigin +
      " at " + time2 + " PM (1 stop -- " + stop2 + ").\n";
    return(flight);
  }
```

Listing 15.5 TravelCustomer.java (continued)

```
  private String replaceIfMissing(String value,
                                  String defaultValue) {
    if ((value != null) && (value.length() > 0)) {
      return(value);
    } else {
      return(defaultValue);
    }
  }

  public static TravelCustomer findCustomer
                                 (String emailAddress,
                                  TravelCustomer[] customers) {
    if (emailAddress == null) {
      return(null);
    }
    for(int i=0; i<customers.length; i++) {
      String custEmail = customers[i].getEmailAddress();
      if (emailAddress.equalsIgnoreCase(custEmail)) {
        return(customers[i]);
      }
    }
    return(null);
  }
}
```

Listing 15.6 TravelData.java

```
package coreservlets;

/** This class simply sets up some static data to
 *  describe some supposed preexisting customers.
 *  Use a database call in a real application. See
 *  CSAJSP Chapter 18 for many examples of the use
 *  of JDBC from servlets.
 */

public class TravelData {
  private static FrequentFlyerInfo[] janeFrequentFlyerData =
    { new FrequentFlyerInfo("Java Airways", "123-4567-J"),
      new FrequentFlyerInfo("Delta", "234-6578-D") };
  private static RentalCarInfo[] janeRentalCarData =
    { new RentalCarInfo("Alamo", "345-AA"),
      new RentalCarInfo("Hertz", "456-QQ-H"),
      new RentalCarInfo("Avis", "V84-N8699") };
```

Listing 15.6 `TravelData.java` (continued)

```
  private static HotelInfo[] janeHotelData =
   { new HotelInfo("Marriot", "MAR-666B"),
     new HotelInfo("Holiday Inn", "HI-228-555") };
  private static FrequentFlyerInfo[] joeFrequentFlyerData =
    { new FrequentFlyerInfo("Java Airways", "321-9299-J"),
      new FrequentFlyerInfo("United", "442-2212-U"),
      new FrequentFlyerInfo("Southwest", "1A345") };
  private static RentalCarInfo[] joeRentalCarData =
    { new RentalCarInfo("National", "NAT00067822") };
  private static HotelInfo[] joeHotelData =
   { new HotelInfo("Red Roof Inn", "RRI-PREF-236B"),
     new HotelInfo("Ritz Carlton", "AA0012") };
  private static TravelCustomer[] travelData =
    { new TravelCustomer("jane@somehost.com",
                         "tarzan52",
                         "Jane",
                         "Programmer",
                         "Visa",
                         "1111-2222-3333-6755",
                         "(123) 555-1212",
                         "6 Cherry Tree Lane\n" +
                           "Sometown, CA 22118",
                         janeFrequentFlyerData,
                         janeRentalCarData,
                         janeHotelData),
      new TravelCustomer("joe@somehost.com",
                         "qWeRtY",
                         "Joe",
                         "Hacker",
                         "JavaSmartCard",
                         "000-1111-2222-3120",
                         "(999) 555-1212",
                         "55 25th St., Apt 2J\n" +
                           "New York, NY 12345",
                         joeFrequentFlyerData,
                         joeRentalCarData,
                         joeHotelData)
    };

  public static TravelCustomer[] getTravelData() {
    return(travelData);
  }
}
```

Listing 15.7 FrequentFlyerInfo.java

```
package coreservlets;

/** Simple class describing an airline and associated
 *  frequent flyer number, used from the TravelData class
 *  (where an array of FrequentFlyerInfo is associated with
 *  each customer).
 */

public class FrequentFlyerInfo {
  private String airlineName, frequentFlyerNumber;

  public FrequentFlyerInfo(String airlineName,
                           String frequentFlyerNumber) {
    this.airlineName = airlineName;
    this.frequentFlyerNumber = frequentFlyerNumber;
  }

  public String getAirlineName() {
    return(airlineName);
  }

  public String getFrequentFlyerNumber() {
    return(frequentFlyerNumber);
  }
}
```

Listing 15.8 RentalCarInfo.java

```
package coreservlets;

/** Simple class describing a car company and associated
 *  frequent renter number, used from the TravelData class
 *  (where an array of RentalCarInfo is associated with
 *  each customer).
 */

public class RentalCarInfo {
  private String rentalCarCompany, rentalCarNumber;

  public RentalCarInfo(String rentalCarCompany,
                       String rentalCarNumber) {
    this.rentalCarCompany = rentalCarCompany;
    this.rentalCarNumber = rentalCarNumber;
  }

  public String getRentalCarCompany() {
    return(rentalCarCompany);
  }

  public String getRentalCarNumber() {
    return(rentalCarNumber);
  }
}
```

Listing 15.9 HotelInfo.java

```
package coreservlets;

/** Simple class describing a hotel name and associated
 *  frequent guest number, used from the TravelData class
 *  (where an array of HotelInfo is associated with
 *  each customer).
 */

public class HotelInfo {
  private String hotelName, frequentGuestNumber;

  public HotelInfo(String hotelName,
                   String frequentGuestNumber) {
    this.hotelName = hotelName;
    this.frequentGuestNumber = frequentGuestNumber;
  }

  public String getHotelName() {
    return(hotelName);
  }

  public String getfrequentGuestNumber() {
    return(frequentGuestNumber);
  }
}
```

15.3 Including Static or Dynamic Content

If a servlet uses the `forward` method of `RequestDispatcher`, it cannot actually send any output to the client—it must leave that entirely to the destination page. If the servlet wants to generate some of the content itself but use a JSP page or static HTML document for other parts of the result, the servlet can use the `include` method of `RequestDispatcher` instead. The process is very similar to that for forwarding requests: call the `getRequestDispatcher` method of `ServletContext` with an address relative to the server root, then call `include` with the `HttpServletRequest` and `HttpServletResponse`. The two differences when `include` is used are that you can send content to the browser before making the call and that control is returned to the servlet after the `include` call finishes. Although the included pages (servlets, JSP

pages, or even static HTML) can send output to the client, they should not try to set HTTP response headers. Here is an example:

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("...");
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher("/path/resource");
dispatcher.include(request, response);
out.println("...");
```

The `include` method has many of the same features as the `forward` method. If the original method uses `POST`, so does the forwarded request. Whatever request data was associated with the original request is also associated with the auxiliary request, and you can add new parameters (in version 2.2 only) by appending them to the URL supplied to `getRequestDispatcher`. Also supported in version 2.2 is the ability to get a `RequestDispatcher` by name (`getNamedDispatcher`) or by using a relative URL (use the `getRequestDispatcher` method of the `HttpServletRequest`); see Section 15.1 (Forwarding Requests) for details. However, `include` does one thing that `forward` does not: it automatically sets up attributes in the `HttpServletRequest` object that describe the original request path in case the included servlet or JSP page needs that information. These attributes, available to the included resource by calling `getAttribute` on the `HttpServletRequest`, are listed below:

- `javax.servlet.include.request_uri`
- `javax.servlet.include.context_path`
- `javax.servlet.include.servlet_path`
- `javax.servlet.include.path_info`
- `javax.servlet.include.query_string`

Note that this type of file inclusion is not the same as the nonstandard *servlet chaining* supported as an extension by several early servlet engines. With servlet chaining, each servlet in a series of requests can see (and modify) the output of the servlet before it. With the `include` method of `RequestDispatcher`, the included resource cannot see the output generated by the original servlet. In fact, there is no standard construct in the servlet specification that reproduces the behavior of servlet chaining.

Also note that this type of file inclusion differs from that supported by the JSP `include` directive discussed in Section 12.1 (Including Files at Page Translation Time). There, the actual *source code* of JSP files was included in the page by use of the `include` directive, whereas the `include` method of `RequestDispatcher` just includes the *result* of the specified resource. On the other hand, the `jsp:include` action discussed in Section 12.2 (Including Files at Request Time) has behavior similar to that of the `include` method, except that `jsp:include` is available only from JSP pages, not from servlets.

15.4 Example: Showing Raw Servlet and JSP Output

When you are debugging servlets or JSP pages, it is often useful to see the raw HTML they generate. To do this, you can choose "View Source" from the browser menu after seeing the result. Alternatively, to set HTTP request headers and see the HTTP response headers in addition to HTML source, use the `WebClient` program shown in Section 2.10 (WebClient: Talking to Web Servers Interactively). For quick debugging, another option is available: create a servlet that takes a URL as input and creates an output page showing the HTML source code. Accomplishing this task relies on the fact that the HTML `TEXTAREA` element ignores all HTML markup other than the `</TEXTAREA>` tag. So, the original servlet generates the top of a Web page, up to a `<TEXTAREA>` tag. Then, it includes the output of whatever URL was specified in the query data. Next, it continues with the Web page, starting with a `</TEXTAREA>` tag. Of course, the servlet will fail if it tries to display a resource that contains the `</TEXTAREA>` tag, but the point here is the process of including files.

Listing 15.10 shows the servlet that accomplishes this task, and Listing 15.11 shows an HTML form that gathers input and sends it to the servlet. Figures 15–3 and 15–4 show the results of the HTML form and servlet, respectively.

Listing 15.10 ShowPage.java

```
package coreservlets;

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

/** Example of the include method of RequestDispatcher.
 *  Given a URI on the same system as this servlet, the
 *  servlet shows you its raw HTML output.
 */

public class ShowPage extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String url = request.getParameter("url");
    out.println(ServletUtilities.headWithTitle(url) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + url + "</H1>\n" +
                "<FORM><CENTER>\n" +
                "<TEXTAREA ROWS=30 COLS=70>");
    if ((url == null) || (url.length() == 0)) {
      out.println("No URL specified.");
    } else {
      // Attaching data works only in version 2.2.
      String data = request.getParameter("data");
      if ((data != null) && (data.length() > 0)) {
        url = url + "?" + data;
      }
      RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher(url);
      dispatcher.include(request, response);
    }
    out.println("</TEXTAREA>\n" +
                "</CENTER></FORM>\n" +
                "</BODY></HTML>");
  }

  /** Handle GET and POST identically. */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

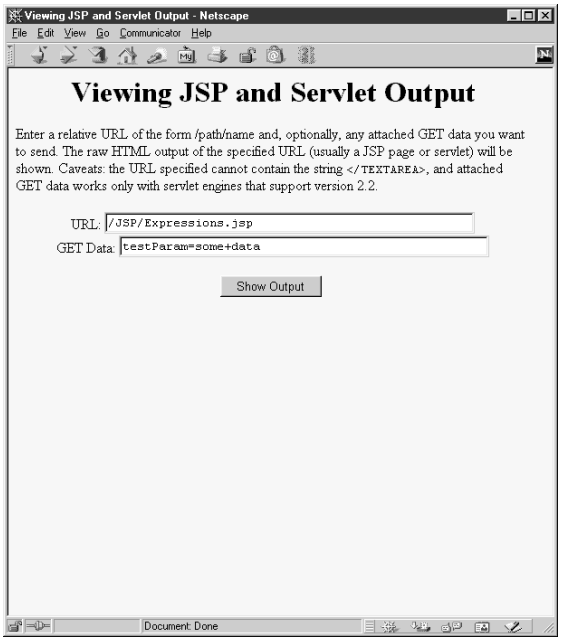


Figure 15–3 Front end to `ShowPage` servlet. See Listing 15.11 for the HTML source.

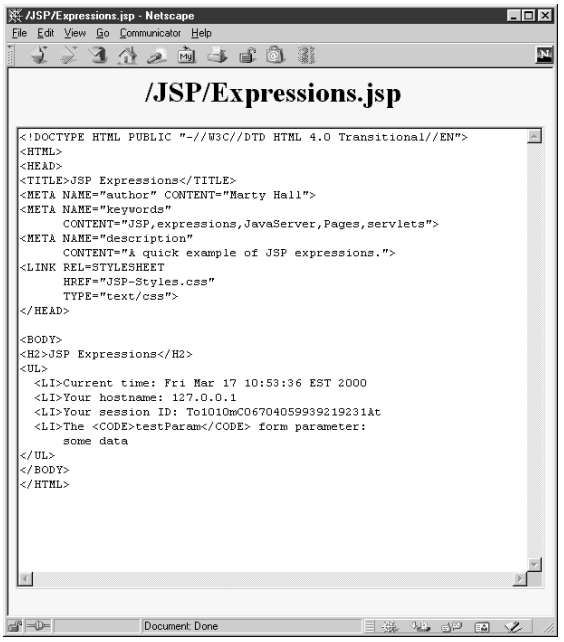


Figure 15–4 Result of `ShowPage` servlet when given a URL referring to `Expressions.jsp` (see Listing 10.1 in Section 10.2).

Listing 15.11 ShowPage.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Viewing JSP and Servlet Output</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Viewing JSP and Servlet Output</H1>
Enter a relative URL of the form /path/name and, optionally,
any attached GET data you want to send. The raw HTML output
of the specified URL (usually a JSP page or servlet) will be
shown. Caveats: the URL specified cannot contain the string
<CODE>&lt;/TEXTAREA&gt;</CODE>, and attached GET data works
only with servlet engines that support version 2.2.

<FORM ACTION="/servlet/coreservlets.ShowPage">
  <CENTER>
    URL:
    <INPUT TYPE="TEXT" NAME="url" SIZE=50 VALUE="/"><BR>
    GET Data:
    <INPUT TYPE="TEXT" NAME="data" SIZE=50><BR><BR>
    <Input TYPE="SUBMIT" VALUE="Show Output">
  </CENTER>
</FORM>

</BODY>
</HTML>
```

15.5 Forwarding Requests From JSP Pages

The most common request forwarding scenario is that the request first comes to a servlet and the servlet forwards the request to a JSP page. The reason a servlet usually handles the original request is that checking request parameters and setting up beans requires a lot of programming, and it is more convenient to do this programming in a servlet than in a JSP document. The reason that the destination page is usually a JSP document is that JSP simplifies the process of creating the HTML content.

However, just because this is the *usual* approach doesn't mean that it is the *only* way of doing things. It is certainly possible for the destination page to be a servlet. Similarly, it is quite possible for a JSP page to forward requests else-

where. For example, a request might go to a JSP page that normally presents results of a certain type and that forwards the request elsewhere only when it receives unexpected values.

Sending requests to servlets instead of JSP pages requires no changes whatsoever in the use of the `RequestDispatcher`. However, there is special syntactic support for forwarding requests from JSP pages. In JSP, the `jsp:forward` action is simpler and easier to use than wrapping up `RequestDispatcher` code in a scriptlet. This action takes the following form:

```
<jsp:forward page="Relative URL" />
```

The `page` attribute is allowed to contain JSP expressions so that the destination can be computed at request time. For example, the following sends about half the visitors to `http://host/examples/page1.jsp` and the others to `http://host/examples/page2.jsp`.

```
<% String destination;
   if (Math.random() > 0.5) {
     destination = "/examples/page1.jsp";
   } else {
     destination = "/examples/page2.jsp";
   }
%>
<jsp:forward page="<%= destination %>" />
```