Integrating Servlets and JSP: The Model View Controller (MVC) Architecture

Topics in This Chapter

- Understanding the benefits of MVC
- Using `RequestDispatcher` to implement MVC
- Forwarding requests from servlets to JSP pages
- Handling relative URLs
- Choosing among different display options
- Comparing data-sharing strategies
- Forwarding requests from JSP pages
- Including pages instead of forwarding to them

Chapter 15

Servlets are good at data *processing*: reading and checking data, communicating with databases, invoking business logic, and so on. JSP pages are good at *presentation*: building HTML to represent the results of requests. This chapter describes how to combine servlets and JSP pages to best make use of the strengths of each technology.

15.1 Understanding the Need for MVC

Servlets are great when your application requires a lot of real programming to accomplish its task. As illustrated earlier in this book, servlets can manipulate HTTP status codes and headers, use cookies, track sessions, save information between requests, compress pages, access databases, generate JPEG 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: as illustrated in Figure 15–1, JSP 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 Java code inside beans or even define your own JSP tags.

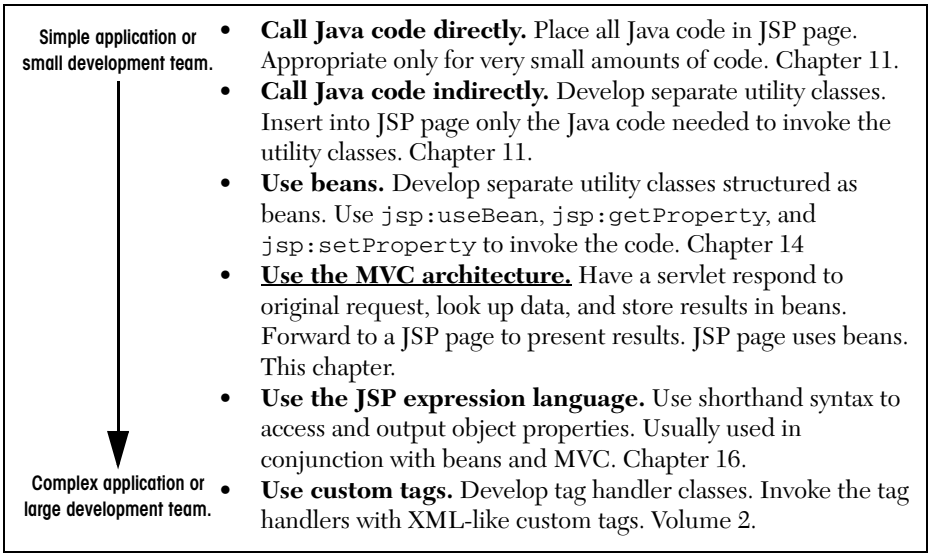

Figure 15–1 Strategies for invoking dynamic code from JSP.

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? Scripting expressions, 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. Similarly, what if you need complex reasoning just to determine the type of data that applies to the current situation? JSP is poor at this type of business logic.

The solution is to use *both* servlets and JavaServer Pages. In this approach, known as the Model View Controller (MVC) or Model 2 architecture, you let each technology concentrate on what it excels at. The original request is handled by a servlet. The servlet invokes the business-logic and data-access code and creates beans to represent the results (that's the *model*). Then, the servlet decides which JSP page is appropriate to present those particular results and forwards the request there (the JSP page is the *view*). The servlet decides what business logic code applies and which JSP page should present the results (the servlet is the *controller*).

MVC Frameworks

The key motivation behind the MVC approach is the desire to separate the code that creates and manipulates the data from the code that presents the data. The basic tools needed to implement this presentation-layer separation are standard in the

servlet API and are the topic of this chapter. However, in very complex applications, a more elaborate MVC framework is sometimes beneficial. The most popular of these frameworks is Apache Struts; it is discussed at length in Volume 2 of this book. Although Struts is useful and widely used, you should not feel that you must use Struts in order to apply the MVC approach. For simple and moderately complex applications, implementing MVC from scratch with `RequestDispatcher` is straightforward and flexible. Do not be intimidated: go ahead and start with the basic approach. In many situations, you will stick with the basic approach for the entire life of your application. Even if you decide to use Struts or another MVC framework later, you will recoup much of your investment because most of your work will also apply to the elaborate frameworks.

Architecture or Approach?

The term "architecture" often connotes "overall system design." Although many systems are indeed designed with MVC at their core, it is not necessary to redesign your overall system just to make use of the MVC approach. Not at all. It is quite common for applications to handle some requests with servlets, other requests with JSP pages, and still others with servlets and JSP acting in conjunction as described in this chapter. Do not feel that you have to rework your entire system architecture just to use the MVC approach: go ahead and start applying it in the parts of your application where it fits best.

15.2 Implementing MVC with RequestDispatcher

The most important point about MVC is the idea of separating the business logic and data access layers from the presentation layer. The syntax is quite simple, and in fact you should be familiar with much of it already. Here is a quick summary of the required steps; the following subsections supply details.

1. **Define beans to represent the data.** As you know from Section 14.2, beans are just Java objects that follow a few simple conventions. Your first step is define beans to represent the results that will be presented to the user.
2. **Use a servlet to handle requests.** In most cases, the servlet reads request parameters as described in Chapter 4.
3. **Populate the beans.** The servlet invokes business logic (application-specific code) or data-access code (see Chapter 17) to obtain the results. The results are placed in the beans that were defined in step 1.

4. **Store the bean in the request, session, or servlet context.** The servlet calls `setAttribute` on the request, session, or servlet context objects to store a reference to the beans that represent the results of the request.
5. **Forward the request to a JSP page.** The servlet determines which JSP page is appropriate to the situation and uses the `forward` method of `RequestDispatcher` to transfer control to that page.
6. **Extract the data from the beans.** The JSP page accesses beans with `jsp:useBean` and a `scope` matching the location of step 4. The page then uses `jsp:getProperty` to output the bean properties. The JSP page does not create or modify the bean; it merely extracts and displays data that the servlet created.

Defining Beans to Represent the Data

Beans are Java objects that follow a few simple conventions. In this case, since a servlet or other Java routine (never a JSP page) will be creating the beans, the requirement for an empty (zero-argument) constructor is waived. So, your objects merely need to follow the normal recommended practices of keeping the instance variables private and using accessor methods that follow the get/set naming convention.

Since the JSP page will only access the beans, not create or modify them, a common practice is to define *value objects*: objects that represent results but have little or no additional functionality.

Writing Servlets to Handle Requests

Once the bean classes are defined, the next task is to write a servlet to read the request information. Since, with MVC, a servlet responds to the initial request, the normal approaches of Chapters 4 and 5 are used to read request parameters and request headers, respectively. The shorthand `populateBean` method of Chapter 4 can be used, but you should note that this technique populates a *form* bean (a Java object representing the form parameters), not a *result* bean (a Java object representing the results of the request).

Although the servlets use the normal techniques to read the request information and generate the data, they do not use the normal techniques to output the results. In fact, with the MVC approach the servlets do not create *any* output; the output is completely handled by the JSP pages. So, the servlets do not call `response.setContentType`, `response.getWriter`, or `out.println`.

Populating the Beans

After you read the form parameters, you use them to determine the results of the request. These results are determined in a completely application-specific manner. You might call some business logic code, invoke an Enterprise JavaBeans component, or query a database. No matter how you come up with the data, you need to use that data to fill in the value object beans that you defined in the first step.

Storing the Results

You have read the form information. You have created data specific to the request. You have placed that data in beans. Now you need to store those beans in a location that the JSP pages will be able to access.

A servlet can store data for JSP pages in three main places: in the `HttpServletRequest`, in the `HttpSession`, and in the `ServletContext`. These storage locations correspond to the three nondefault values of the `scope` attribute of `jsp:useBean`: that is, `request`, `session`, and `application`.

- **Storing data that the JSP page will use only in this request.** First, the servlet would create and store data as follows:

  ```
  ValueObject value = new ValueObject(...);
  request.setAttribute("key", value);
  ```

  Next, the servlet would forward the request to a JSP page that uses the following to retrieve the data.

  ```
  <jsp:useBean id="key" type="somePackage.ValueObject"
               scope="request" />
  ```

  Note that request *attributes* have nothing to do with request *parameters* or request *headers*. The request attributes are independent of the information coming from the client; they are just application-specific entries in a hash table that is attached to the request object. This table simply stores data in a place that can be accessed by both the current servlet and JSP page, but not by any other resource or request.
- **Storing data that the JSP page will use in this request and in later requests from the same client.** First, the servlet would create and store data as follows:

  ```
  ValueObject value = new ValueObject(...);
  HttpSession session = request.getSession();
  session.setAttribute("key", value);
  ```

Next, the servlet would forward to a JSP page that uses the following to retrieve the data:

```
<jsp:useBean id="key" type="somePackage.ValueObject"
             scope="session" />
```

- **Storing data that the JSP page will use in this request and in later requests from any client.** First, the servlet would create and store data as follows:

```
ValueObject value = new ValueObject(...);
getServletContext().setAttribute("key", value);
```

Next, the servlet would forward to a JSP page that uses the following to retrieve the data:

```
<jsp:useBean id="key" type="somePackage.ValueObject"
             scope="application" />
```

As described in Section 15.3, the servlet code is normally synchronized to prevent the data changing between the servlet and the JSP page.

Forwarding Requests to JSP Pages

You forward requests with the `forward` method of `RequestDispatcher`. You obtain a `RequestDispatcher` by calling the `getRequestDispatcher` method of `ServletRequest`, supplying a relative address. You are permitted to specify addresses in the WEB-INF directory; clients are not allowed to directly access files in WEB-INF, but the server is allowed to transfer control there. Using locations in WEB-INF prevents clients from inadvertently accessing JSP pages directly, without first going through the servlets that create the JSP data.

Core Approach

If your JSP pages only make sense in the context of servlet-generated data, place the pages under the WEB-INF directory. That way, servlets can forward requests to the pages, but clients cannot access them directly.

Once you have a `RequestDispatcher`, you use `forward` to transfer control to the associated address. You supply the `HttpServletRequest` and `HttpServletResponse` as arguments. Note that the `forward` method of `RequestDispatcher` is

quite different from the `sendRedirect` method of `HttpServletRequest` (Section 7.1). With `forward`, there is no extra response/request pair as with `sendRedirect`. Thus, the URL displayed to the client does not change when you use `forward`.

Core Note

When you use the `forward` method of `RequestDispatcher`, the client sees the URL of the original servlet, not the URL of the final JSP page.

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` request parameter.

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";
  }
  String address;
  if (operation.equals("order")) {
    address = "/WEB-INF/Order.jsp";
  } else if (operation.equals("cancel")) {
    address = "/WEB-INF/Cancel.jsp";
  } else {
    address = "/WEB-INF/UnknownOperation.jsp";
  }
  RequestDispatcher dispatcher =
    request.getRequestDispatcher(address);
  dispatcher.forward(request, response);
}
```

Forwarding to Static Resources

In most cases, you forward requests to JSP pages or other servlets. In some cases, however, you might want to send requests to static HTML pages. 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 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.

Redirecting Instead of Forwarding

The standard MVC approach is to use the forward method of RequestDispatcher to transfer control from the servlet to the JSP page. However, when you are using session-based data sharing, it is sometimes preferable to use response.sendRedirect.

Here is a summary of the behavior of forward.

- Control is transferred entirely on the server. No network traffic is involved.
- The user does not see the address of the destination JSP page and pages can be placed in WEB-INF to prevent the user from accessing them without going through the servlet that sets up the data. This is beneficial if the JSP page makes sense only in the context of servlet-generated data.

Here is a summary of sendRedirect.

- Control is transferred by sending the client a 302 status code and a Location response header. Transfer requires an additional network round trip.
- The user sees the address of the destination page and can bookmark it and access it independently. This is beneficial if the JSP is designed to use default values when data is missing. For example, this approach would be used when redisplaying an incomplete HTML form or summarizing the contents of a shopping cart. In both cases, previously created data would be extracted from the user's session, so the JSP page makes sense even for requests that do not involve the servlet.

Extracting Data from Beans

Once the request arrives at the JSP page, the JSP page uses `jsp:useBean` and `jsp:getProperty` to extract the data. For the most part, this approach is exactly as described in Chapter 14. There are two differences however:

- **The JSP page never creates the objects.** The servlet, not the JSP page, should create all the data objects. So, to guarantee that the JSP page will not create objects, you should use

  ```
  <jsp:useBean ... type="package.Class" />
  ```

  instead of

  ```
  <jsp:useBean ... class="package.Class" />.
  ```

- **The JSP page should not modify the objects.** So, you should use `jsp:getProperty` but not `jsp:setProperty`.

The scope you specify should match the storage location used by the servlet. For example, the following three forms would be used for request-, session-, and application-based sharing, respectively.

```
<jsp:useBean id="key" type="somePackage.SomeBeanClass"
             scope="request" />

<jsp:useBean id="key" type="somePackage.SomeBeanClass"
             scope="session" />

<jsp:useBean id="key" type="somePackage.SomeBeanClass"
             scope="application" />
```

15.3 Summarizing MVC Code

This section summarizes the code that would be used for request-based, session-based, and application-based MVC approaches.

Request-Based Data Sharing

With request-based sharing, the servlet stores the beans in the `HttpServletRequest`, where they are accessible only to the destination JSP page.

Servlet

```
ValueObject value = new ValueObject(...);
request.setAttribute("key", value);
RequestDispatcher dispatcher =
  request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);
```

JSP Page

```
<jsp:useBean id="key" type="somePackage.ValueObject"
             scope="request" />
<jsp:getProperty name="key" property="someProperty" />
```

Session-Based Data Sharing

With session-based sharing, the servlet stores the beans in the `HttpSession`, where they are accessible to the same client in the destination JSP page or in other pages.

Servlet

```
ValueObject value = new ValueObject(...);
HttpSession session = request.getSession();
session.setAttribute("key", value);
RequestDispatcher dispatcher =
  request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);
```

JSP Page

```
<jsp:useBean id="key" type="somePackage.ValueObject"
             scope="session" />
<jsp:getProperty name="key" property="someProperty" />
```

Application-Based Data Sharing

With application-based sharing, the servlet stores the beans in the `ServletContext`, where they are accessible to any servlet or JSP page in the Web application. To guarantee that the JSP page extracts the same data that the servlet inserted, you should synchronize your code as below.

Servlet

```
synchronized(this) {
  ValueObject value = new ValueObject(...);
  getServletContext().setAttribute("key", value);
```

```
    RequestDispatcher dispatcher =
      request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
    dispatcher.forward(request, response);
  }
```

JSP Page

```
<jsp:useBean id="key" type="somePackage.ValueObject"
             scope="application" />
<jsp:getProperty name="key" property="someProperty" />
```

15.4 Interpreting Relative URLs in the Destination Page

Although a servlet can forward the request to an arbitrary location on the same server, the process is quite different from that of using the `sendRedirect` method of `HttpServletResponse`. 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 servlet URL or 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 simplest solution to this problem 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=...>`.

15.5 Applying MVC: Bank Account Balances

In this section, we apply the MVC approach to an application that displays bank account balances. The controller servlet (Listing 15.2) reads a customer ID and passes that to some data-access code that returns a `BankCustomer` value bean (Listing 15.3). The servlet then stores the bean in the `HttpServletRequest` object where it will be accessible from destination JSP pages but nowhere else. If the account balance of the resulting customer is negative, the servlet forwards to a page designed for delinquent customers (Listing 15.4, Figure 15–2). If the customer has a positive balance of less than $10,000, the servlet transfers to the standard balance-display page (Listing 15.5, Figure 15–3). Next, if the customer has a balance of $10,000 or more, the servlet forwards the request to a page reserved for elite customers (Listing 15.6, Figure 15–4). Finally, if the customer ID is unrecognized, an error page is displayed (Listing 15.7, Figure 15–5).

Listing 15.2 ShowBalance.java

```
package coreservlets;

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

/** Servlet that reads a customer ID and displays
 *  information on the account balance of the customer
 *  who has that ID.
 */

public class ShowBalance extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    BankCustomer customer =
      BankCustomer.getCustomer(request.getParameter("id"));
    String address;
    if (customer == null) {
      address = "/WEB-INF/bank-account/UnknownCustomer.jsp";
    } else if (customer.getBalance() < 0) {
      address = "/WEB-INF/bank-account/NegativeBalance.jsp";
      request.setAttribute("badCustomer", customer);
```

Listing 15.2 ShowBalance.java *(continued)*

```
    } else if (customer.getBalance() < 10000) {
      address = "/WEB-INF/bank-account/NormalBalance.jsp";
      request.setAttribute("regularCustomer", customer);
    } else {
      address = "/WEB-INF/bank-account/HighBalance.jsp";
      request.setAttribute("eliteCustomer", customer);
    }
    RequestDispatcher dispatcher =
      request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
  }
}
```

Listing 15.3 BankCustomer.java

```
package coreservlets;

import java.util.*;

/** Bean to represent a bank customer. */

public class BankCustomer {
  private String id, firstName, lastName;
  private double balance;

  public BankCustomer(String id,
                      String firstName,
                      String lastName,
                      double balance) {
    this.id = id;
    this.firstName = firstName;
    this.lastName = lastName;
    this.balance = balance;
  }

  public String getId() {
    return(id);
  }

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

Listing 15.3 BankCustomer.java *(continued)*

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

  public double getBalance() {
    return(balance);
  }

  public double getBalanceNoSign() {
    return(Math.abs(balance));
  }

  public void setBalance(double balance) {
    this.balance = balance;
  }

  // Makes a small table of banking customers.

  private static HashMap customers;

  static {
    customers = new HashMap();
    customers.put("id001",
                  new BankCustomer("id001",
                                   "John",
                                   "Hacker",
                                   -3456.78));
    customers.put("id002",
                  new BankCustomer("id002",
                                   "Jane",
                                   "Hacker",
                                   1234.56));
    customers.put("id003",
                  new BankCustomer("id003",
                                   "Juan",
                                   "Hacker",
                                   987654.32));
  }

  /** Finds the customer with the given ID.
   *  Returns null if there is no match.
   */

  public static BankCustomer getCustomer(String id) {
    return((BankCustomer)customers.get(id));
  }
}
```

Listing 15.4 NegativeBalance.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>You Owe Us Money!</TITLE>
<LINK REL=STYLESHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      We Know Where You Live!</TABLE>
<P>
<IMG SRC="/bank-support/Club.gif" ALIGN="LEFT">
<jsp:useBean id="badCustomer"
             type="coreservlets.BankCustomer"
             scope="request" />
Watch out,
<jsp:getProperty name="badCustomer" property="firstName" />,
we know where you live.
<P>
Pay us the
$<jsp:getProperty name="badCustomer" property="balanceNoSign" />
you owe us before it is too late!
</BODY></HTML>
```

Listing 15.5 NormalBalance.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Your Balance</TITLE>
<LINK REL=STYLESHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Your Balance</TABLE>
<P>
```

Listing 15.5 NormalBalance.jsp *(continued)*

```
<IMG SRC="/bank-support/Money.gif" ALIGN="RIGHT">
<jsp:useBean id="regularCustomer"
             type="coreservlets.BankCustomer"
             scope="request" />
<UL>
  <LI>First name: <jsp:getProperty name="regularCustomer"
                                   property="firstName" />
  <LI>Last name: <jsp:getProperty name="regularCustomer"
                                  property="lastName" />
  <LI>ID: <jsp:getProperty name="regularCustomer"
                           property="id" />
  <LI>Balance: $<jsp:getProperty name="regularCustomer"
                                 property="balance" />
</UL>
</BODY></HTML>
```

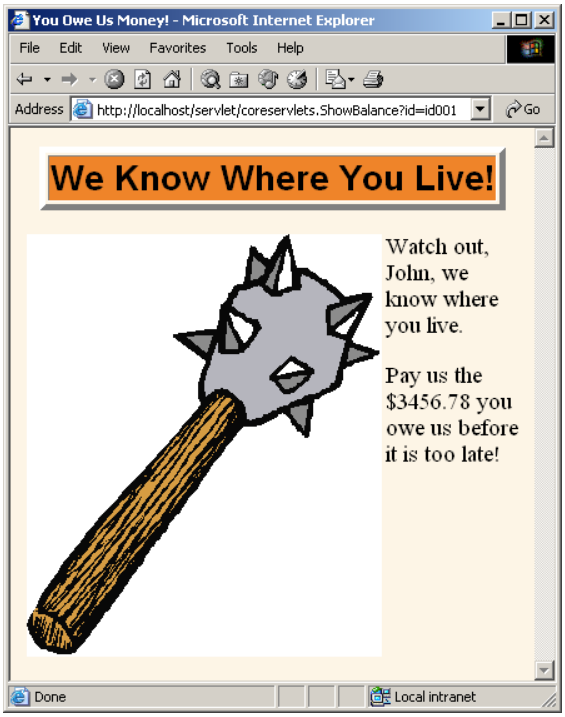

Figure 15–2 The `ShowCustomer` servlet with an ID corresponding to a customer with a negative balance.

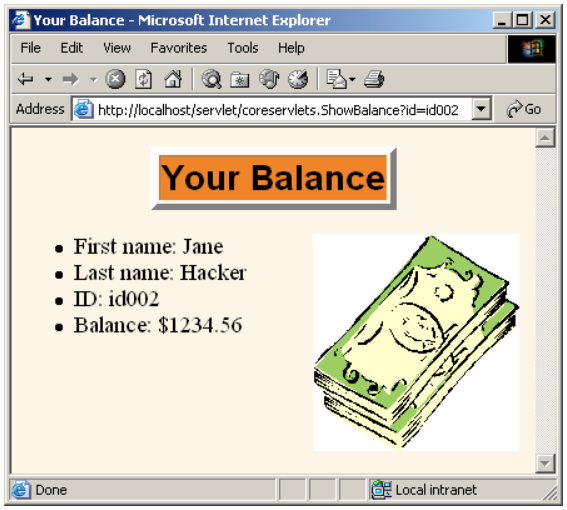

Figure 15–3 The `ShowCustomer` servlet with an ID corresponding to a customer with a normal balance.

Listing 15.6 HighBalance.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Your Balance</TITLE>
<LINK REL=STYLESHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Your Balance</TABLE>
<P>
<CENTER><IMG SRC="/bank-support/Sailing.gif"></CENTER>
<BR CLEAR="ALL">
<jsp:useBean id="eliteCustomer"
             type="coreservlets.BankCustomer"
             scope="request" />
It is an honor to serve you,
<jsp:getProperty name="eliteCustomer" property="firstName" />
<jsp:getProperty name="eliteCustomer" property="lastName" />!
```

Listing 15.6 HighBalance.jsp *(continued)*

```
<P>
Since you are one of our most valued customers, we would like
to offer you the opportunity to spend a mere fraction of your
$<jsp:getProperty name="eliteCustomer"  property="balance" />
on a boat worthy of your status. Please visit our boat store for
more information.
</BODY></HTML>
```

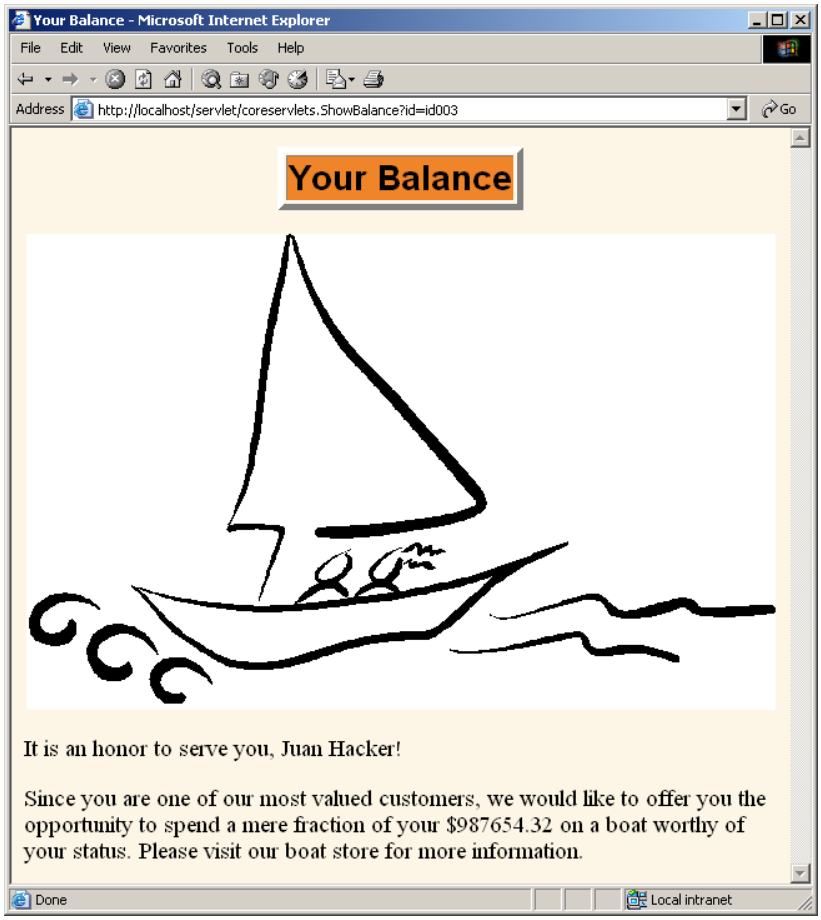

Figure 15–4 The `ShowCustomer` servlet with an ID corresponding to a customer with a high balance.

Listing 15.7 UnknownCustomer.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Unknown Customer</TITLE>
<LINK REL=STYLESHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Unknown Customer</TABLE>
<P>
Unrecognized customer ID.
</BODY></HTML>
```

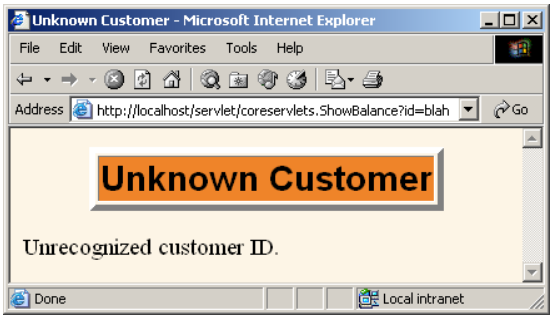

Figure 15–5 The `ShowCustomer` servlet with an unknown customer ID.

15.6 Comparing the Three Data-Sharing Approaches

In the MVC approach, a servlet responds to the initial request. The servlet invokes code that fetches or creates the business data, places that data in beans, stores the beans, and forwards the request to a JSP page to present the results. But, *where* does the servlet store the beans?

The most common answer is, in the request object. That is the only location to which the JSP page has sole access. However, you sometimes want to keep the results

around for the same client (session-based sharing) or store Web-application-wide data (application-based sharing).

This section gives a brief example of each of these approaches.

Request-Based Sharing

In this example, our goal is to display a random number to the user. Each request should result in a new number, so request-based sharing is appropriate.

To implement this behavior, we need a bean to store numbers (Listing 15.8), a servlet to populate the bean with a random value (Listing 15.9), and a JSP page to display the results (Listing 15.10, Figure 15–6).

Listing 15.8 NumberBean.java

```
package coreservlets;

public class NumberBean {
  private double num = 0;

  public NumberBean(double number) {
    setNumber(number);
  }

  public double getNumber() {
    return(num);
  }

  public void setNumber(double number) {
    num = number;
  }
}
```

Listing 15.9 RandomNumberServlet.java

```
package coreservlets;

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

/** Servlet that generates a random number, stores it in a bean,
 *  and forwards to JSP page to display it.
 */
```

Listing 15.9 RandomNumberServlet.java *(continued)*

```
public class RandomNumberServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    NumberBean bean = new NumberBean(Math.random());
    request.setAttribute("randomNum", bean);
    String address = "/WEB-INF/mvc-sharing/RandomNum.jsp";
    RequestDispatcher dispatcher =
      request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
  }
}
```

Listing 15.10 RandomNum.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Random Number</TITLE>
<LINK REL=STYLESHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<jsp:useBean id="randomNum" type="coreservlets.NumberBean"
             scope="request" />
<H2>Random Number:
<jsp:getProperty name="randomNum" property="number" />
</H2>
</BODY></HTML>
```

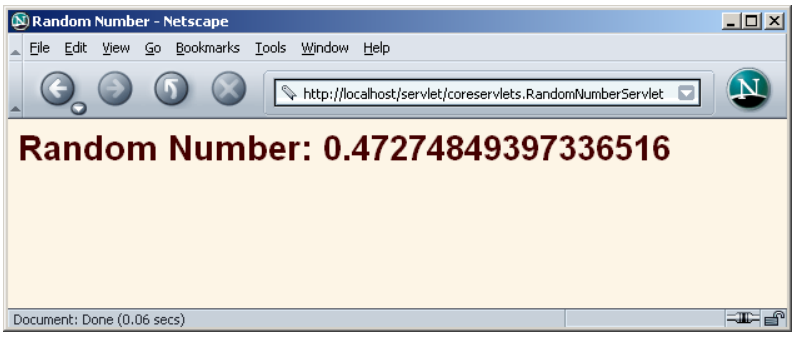

Figure 15–6 Result of `RandomNumberServlet`.

Session-Based Sharing

In this example, our goal is to display users' first and last names. If the users fail to tell us their name, we want to use whatever name they gave us previously. If the users do not explicitly specify a name and no previous name is found, a warning should be displayed. Data is stored for each client, so session-based sharing is appropriate.

To implement this behavior, we need a bean to store names (Listing 15.11), a servlet to retrieve the bean from the session and populate it with first and last names (Listing 15.12), and a JSP page to display the results (Listing 15.13, Figures 15–7 and 15–8).

Listing 15.11 NameBean.java

```
package coreservlets;

public class NameBean {
  private String firstName = "Missing first name";
  private String lastName = "Missing last name";

  public NameBean() {}

  public NameBean(String firstName, String lastName) {
    setFirstName(firstName);
    setLastName(lastName);
  }

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

  public void setFirstName(String newFirstName) {
    firstName = newFirstName;
  }

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

  public void setLastName(String newLastName) {
    lastName = newLastName;
  }
}
```

Listing 15.12 RegistrationServlet.java

```
package coreservlets;

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

/** Reads firstName and lastName request parameters and forwards
 *  to JSP page to display them. Uses session-based bean sharing
 *  to remember previous values.
 */

public class RegistrationServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession();
    NameBean nameBean =
      (NameBean)session.getAttribute("nameBean");
    if (nameBean == null) {
      nameBean = new NameBean();
      session.setAttribute("nameBean", nameBean);
    }
    String firstName = request.getParameter("firstName");
    if ((firstName != null) && (!firstName.trim().equals(""))) {
      nameBean.setFirstName(firstName);
    }
    String lastName = request.getParameter("lastName");
    if ((lastName != null) && (!lastName.trim().equals(""))) {
      nameBean.setLastName(lastName);
    }
    String address = "/WEB-INF/mvc-sharing/ShowName.jsp";
    RequestDispatcher dispatcher =
      request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
  }
}
```

Listing 15.13 ShowName.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Thanks for Registering</TITLE>
<LINK REL=STYLESHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>Thanks for Registering</H1>
<jsp:useBean id="nameBean" type="coreservlets.NameBean"
             scope="session" />
<H2>First Name:
<jsp:getProperty name="nameBean" property="firstName" /></H2>
<H2>Last Name:
<jsp:getProperty name="nameBean" property="lastName" /></H2>
</BODY></HTML>
```

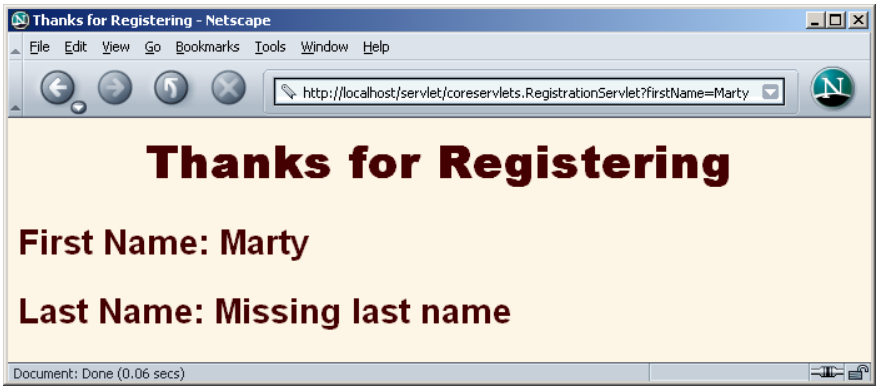

Figure 15–7 Result of `RegistrationServlet` when one parameter is missing and no session data is found.

Figure 15–8 Result of `RegistrationServlet` when one parameter is missing and session data is found.

Application-Based Sharing

In this example, our goal is to display a prime number of a specified length. If the user fails to tell us the desired length, we want to use whatever prime number we most recently computed for *any* user. Data is shared among multiple clients, so application-based sharing is appropriate.

To implement this behavior, we need a bean to store prime numbers (Listing 15.14, which uses the `Primes` class presented earlier in Section 7.4), a servlet to populate the bean and store it in the `ServletContext` (Listing 15.15), and a JSP page to display the results (Listing 15.16, Figures 15–9 and 15–10).

Listing 15.14 PrimeBean.java

```
package coreservlets;

import java.math.BigInteger;

public class PrimeBean {
  private BigInteger prime;

  public PrimeBean(String lengthString) {
    int length = 150;
    try {
      length = Integer.parseInt(lengthString);
    } catch(NumberFormatException nfe) {}
    setPrime(Primes.nextPrime(Primes.random(length)));
  }
```

Listing 15.14 PrimeBean.java *(continued)*

```
  public BigInteger getPrime() {
    return(prime);
  }

  public void setPrime(BigInteger newPrime) {
    prime = newPrime;
  }
}
```

Listing 15.15 PrimeServlet.java

```
package coreservlets;

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

public class PrimeServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String length = request.getParameter("primeLength");
    ServletContext context = getServletContext();
    synchronized(this) {
      if ((context.getAttribute("primeBean") == null) ||
          (length != null)) {
        PrimeBean primeBean = new PrimeBean(length);
        context.setAttribute("primeBean", primeBean);
      }
      String address = "/WEB-INF/mvc-sharing/ShowPrime.jsp";
      RequestDispatcher dispatcher =
        request.getRequestDispatcher(address);
      dispatcher.forward(request, response);
    }
  }
}
```

Listing 15.16 ShowPrime.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>A Prime Number</TITLE>
<LINK REL=STYLESHEET
      HREF="/bank-support/JSP-Styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<H1>A Prime Number</H1>
<jsp:useBean id="primeBean" type="coreservlets.PrimeBean"
             scope="application" />
<jsp:getProperty name="primeBean" property="prime" />
</BODY></HTML>
```

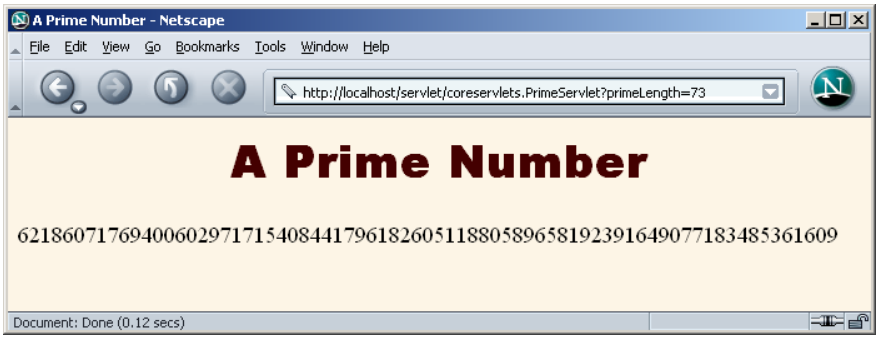

Figure 15–9 Result of `PrimeServlet` when an explicit prime size is given: a new prime of that size is computed.

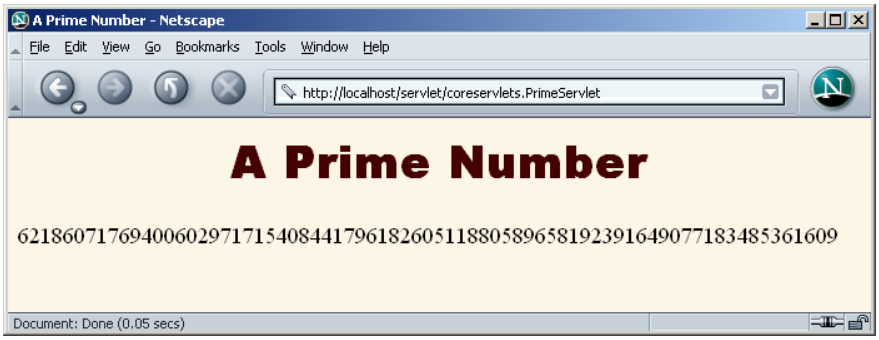

Figure 15–10 Result of `PrimeServlet` when no explicit prime size is given: the previous number is shown and no new prime is computed.

15.7 Forwarding Requests from JSP Pages

The most common request-forwarding scenario is one in which the request first goes 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 elsewhere. 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 `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 code 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 %>" />
```

The `jsp:forward` action, like `jsp:include`, can make use of `jsp:param` elements to supply extra request parameters to the destination page. For details, see the discussion of `jsp:include` in Section 13.2.

15.8 Including Pages

The `forward` method of `RequestDispatcher` relies on the destination JSP page to generate the *complete* output. The servlet is not permitted to generate any output of its own.

An alternative to `forward` is `include`. With `include`, the servlet can combine its output with that of one or more JSP pages. More commonly, the servlet still relies on JSP pages to produce the output, but the servlet invokes different JSP pages to create different *sections* of the page. Does this sound familiar? It should: the `include` method of `RequestDispatcher` is the code that the `jsp:include` action (Section 13.1) invokes behind the scenes.

This approach is most common when your servlets create portal sites that let users specify where on the page they want various pieces of content to be displayed. Here is a representative example.

```
String firstTable, secondTable, thirdTable;
if (someCondition) {
  firstTable = "/WEB-INF/Sports-Scores.jsp";
  secondTable = "/WEB-INF/Stock-Prices.jsp";
  thirdTable = "/WEB-INF/Weather.jsp";
} else if (...) { ... }
RequestDispatcher dispatcher =
  request.getRequestDispatcher("/WEB-INF/Header.jsp");
dispatcher.include(request, response);
dispatcher =
  request.getRequestDispatcher(firstTable);
dispatcher.include(request, response);
dispatcher =
  request.getRequestDispatcher(secondTable);
dispatcher.include(request, response);
dispatcher =
  request.getRequestDispatcher(thirdTable);
dispatcher.include(request, response);
dispatcher =
  request.getRequestDispatcher("/WEB-INF/Footer.jsp");
dispatcher.include(request, response);
```