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" />
```