CHAPTER 3

The Web Tier

by Greg Murray and Mark Johnson

A J2EE application's Web tier makes the application available on the World Wide Web. The Web tier handles all of a J2EE application's communication with Web clients. While the J2EE platform is designed for general-purpose enterprise information management, the Web tier merits special attention because of the high level of Web technology support J2EE provides.

This chapter describes effective use of J2EE Web tier technologies in J2EE applications. This chapter is specifically *not* about Web page design. The chapter begins with a discussion of the responsibilities of the Web tier in a J2EE application, and the roles that Web tier technologies may play in an MVC design. The next section is a compact overview of J2EE Web tier components and containers, including details on specific features of each technology. Following the technology section is a series of design issues and guidelines that explain the competing design forces that arise in typical Web tier designs, and describe specific techniques for how to resolve those forces. The final section is a set of implementation recommendations, with code samples from the Java Pet Store, that demonstrate how to use J2EE Web tier technology effectively.

3.1 The Purpose of the Web Tier

Drastically simplified, the Web tier's job is to receive input from HTTP clients, apply the received inputs to internal business logic, and send results back to the client using HTTP. The content produced by the Web tier is usually HTML or XML, though the Web tier can generate and serve content of any type. Business logic may be delegated to other tiers, or may be implemented entirely within the

Web tier. The J2EE component technologies, JSP pages and servlets, provide a powerful toolbox for creating Web applications.

The Web tier typically plays the following roles in a J2EE application:

- *Delivering application functionality to the Web.* Application designs usually separate business logic from data presentation. The Web tier receives HTTP inputs and produces HTTP outputs for business logic in a back-end application model, which may be shared with other types of clients.
- *Dynamic content generation.* The Web tier components are designed for producing dynamic Web content. Servlets and JSP pages can format data from multiple sources, based on user inputs and context, into entirely arbitrary data formats.
- *Data presentation and input collection.* The Web tier communicates with Web clients, so Web document formatting code naturally belongs in the Web tier. Also, user input arrives at the Web tier as HTTP, PUT, or GET actions. The Web tier receives these actions and translates them into a form (often some form of command object) that the rest of the application understands.
- *Screen flow.* The logic that determines which "screen" (that is, which page) to display next usually resides in the Web tier, since screen flow tends to be specific to client capabilities.
- *State management.* The Web tier has a simple, flexible mechanism for accumulating transaction data and context data over the lifetime of a user session.
- *Support for multiple and future client types.* The Web uses the HTTP protocol to deliver data to clients. The data are described by MIME types, an extensible (if limited) content typing mechanism. Any client capable of HTTP can participate in a Web application, and that client can use any type of downloadable content: even content types that do not yet exist.
- *Protocol translation.* EJB servers, databases, EIS-tier resource managers, and other network resources all have their own protocols. The Web tier can be thought of as a layer of software that translates HTTP requests and responses to and from these other protocols.
- *Possibly, business logic implementation.* The business logic of an application can be implemented entirely within the Web tier. Placing business logic in the Web tier is best for Web-only, low- to medium-volume applications, with only simple needs for transactions, data persistence, and concurrency. For larger-

scale, more sophisticated applications, the Web tier usually acts as a Web front-end for business logic implemented in EJB tier enterprise beans.

So, in summary, the Web tier specializes in dealing with all issues related to interactions between Web clients, like browsers and Web service peers, and the application model, which resides either in the Web tier itself, or in some other tier.

A designer may choose to implement business logic in the EJB tier, and reap the benefits of scalability, portability, concurrency, automatic resource management, security, etc. that enterprise beans provide. Or, business logic may be implemented in the Web tier, if the costs of using enterprise beans outweigh their advantages. But business logic and presentation logic should be separated from one another regardless of where they are implemented. (For more on this topic, see the guideline entitled *Separate business logic from presentation* on p. XX.)

3.2 MVC design in the Web tier

Model-View-Controller is an abstract architectural pattern, implying no particular implementation. This section describes MVC applied to the J2EE Web tier. In a typical J2EE application, the Web tier usually handles most controller functionality, often produces the application views, and sometimes the model. An introduction to the MVC pattern appears in the first chapter of this book.

The Web tier interacts with Web clients, which are often browsers. Web-tier components receive user requests from the client, perform application functions, and select and deliver content to the clients for display. These tasks are similar to those of an MVC controller, which receives user input, manipulates the application model, and selects the next user view. Since the responsibilities of the controller and the Web tier are so much alike, the Web tier is the logical place for the controller of an MVC Web application design.

In addition to controller functionality, much of the logic for MVC views may reside in the Web tier. Additionally, in Web-only applications, the MVC model may also be implemented in the Web tier. A Web-tier MVC model is often supported by EIS-tier resources (such as databases) for persistence, or for access to functionality in external packages or legacy applications.

The J2EE Web tier uses HTTP as a transport protocol for communicating with clients, but Web-tier components are not limited to serving HTML-over-HTTP Web browsers. Other client types the Web tier may serve include MIDP clients using proprietary protocols, rich clients using custom XML application

protocols, or Web service peers requesting services with ebXML or SOAP messages. Each of these examples uses a different application-level protocol, while using HTTP for transport. A properly-designed Web tier unifies access to application functionality for any client type. Web-tier components also provide some client types with a virtual session layer.

This section describes the Web tier in the role of MVC controller, along with some discussion Web-tier implementation of the MVC model and views.

3.2.1 Web-tier MVC Controller

J2EE BluePrints recommends implementing the controller of an MVC design in the Web tier of a J2EE Web application. A Web tier controller has the following responsibilities:

- translates HTTP actions (POST, GET, PUT) into actions on the MVC model
- selects the next view to be displayed, based on model state and user activity
- delivers view content to the Web client

The *Front Controller* design pattern describes a clean way to implement a Web-tier MVC controller. A Web application controller translates incoming HTTP requests into method calls on the objects representing the application model. The controller also selects the next user view to be displayed, based on the success or failure of the model operation, results returned, and other internal application state.

The Front Controller centralizes control of dispatching operations in response to requests, and of selecting the appropriate next user view. The Front Controller usually manages all Web input and output for an application, providing a single point of control for all application request processing. This approach simplifies client implementation, since the client always sends requests to and receives responses from the same URL. The Front Controller decouples presentation components from the operations those components perform, since the mapping between the two is localized to the controller. Decoupling presentation components from business logic makes it easier to vary the two independently, easing maintenance and supporting extensibility. The central point of dispatch is a logical place for such global facilities as security and logging.

In J2EE, a Web controller is typically implemented as a servlet, which is an HTTP request handler, written in the Java language and operating inside a Web

container. The Java Pet Store Front Controller is a servlet that receives all application requests, performs calls on the application model, and selects the next view to display. The user views chosen by the Front Controller are mostly JSP pages, which are discussed in the next section. A separate servlet, the templating service, assembles these JSPs into composite views.

3.2.2 Web-tier MVC Views

MVC views display retrieved from or produced by the MVC model. View components (also known as "presentation components") in the Web tier are usually HTML pages, JSP pages, or servlets. HTML pages contain static content, while dynamic content is usually generated by Web-tier JSPs and servlets. JSP pages are best used for generating text-based content, often HTML or XML. Servlets are most appropriate for generating binary content, or content with variable structure. (For an in-depth explanation of appropriate technology usage, see "Web-tier technology selection," starting on page 101.) Since HTML browsers are very lightweight clients, most dynamic content generation and even some content styling occur in the Web tier. Creation, styling, and display of MVC user views is usually split between the Web and Client tiers. Heavier-weight clients, such as "rich" standalone clients, application clients, or special content formats such as MacroMedia Flash or Adobe's PDF, can implement relatively more view functionality in the Client tier, and less in the Web tier.

See the chapter *The Client Tier* for more on J2EE client technologies.

3.2.3 Web-tier MVC Model

MVC model classes implement business logic without reference to any specific presentation technology. Simple J2EE applications may implement an MVC model as a collection of conventional Java objects, often in the form of Web-tier JavaBeans components used directly by JSP pages. Larger applications frequently replace these Web-tier model beans with enterprise beans, which offer scalability, concurrency, load balancing, automatic resource management, and other benefits. JavaBeans components provide quick access to local data, while enterprise beans provide remote access to concurrent data and shared business logic. Accordingly, many designers will choose a combination of Web-tier beans and EJB tier enterprise beans for modeling business objects.

The remote communication required by pre-2.0 enterprise bean implementations often inflicted high performance costs, especially in the case of entity beans.

Two features of the EJB 2.0 specification, one required and one optional, address this concern by eliminating remote communication in some cases. The required "local interface" feature of the EJB 2.0 specification allows enterprise beans within a single JVM instance to access each other locally. Local beans access each other using a direct method call, rather than a remote one. The EJB 2.0 specification also recommends an optional feature allowing Web-tier components to access enterprise beans locally. In both cases, enterprise beans still invoke each others' methods through calls to the container, but co-locating the components within the same JVM instance improves performance by eliminating expensive remote method calls. Accessing entity beans locally from the Web tier will likely be a specification requirement in future releases of the EJB specification.

See the chapter *The EJB Tier* for more on using enterprise beans in J2EE applications.

3.2.4 Non-interactive Web-tier components

From an MVC point of view, a "Web service" is simply all or part of an application model, exposed on the Web in a way that other applications can access the model with standard Web protocols. For example, consider an online Web auction application. Human bidders might use a browser to access the Web application as a series of JSP views, making bids, receiving outbid notifications by email, and so on. The auction application may also offer Web services for listing items at auction, or for making bids or checking current winning bids. The Web service interface would allow other applications, rather than humans using browsers, to participate in the auction. So an "auction watcher" application might access several auction applications' models through their Web service interfaces, collecting and consolidating information about items of interest from multiple auction sites.

Sophisticated Web services might also include controller functionality, particularly if the interaction between the Web service and the application it is serving is session-based. A non-interactive Web-tier controller would track state as it accumulates over a session, managing the conversation and performing transactions based on that state.

3.3 Web tier Technologies

J2EE provides a rich set of technologies and APIs for creating scalable Web applications. This section describes Web technologies that are available in J2EE designs.

The first section describes traditional Web-tier technologies, including static HTML pages and CGI (Common Gateway Interface) programs, to explain why servlets and JSP pages are useful. The bulk of the section discusses the two J2EE Web-tier component types (servlets and JSP pages), the Web container that manages instances of these components, and common techniques and interfaces for using these technologies effectively. The section ends with a discussion of interoperation with non-J2EE technologies, and a quick look at Web services.

3.3.1 Traditional Web-tier technologies

The earliest versions of the World Wide Web was based on basic HTTP servers, serving static HTML pages to HTML browsers, and much if not most of the Web still uses this model. But it quickly became clear that dynamic content, generated on demand, would make the Web a platform for delivery of applications, as well as content.

Several mechanisms have been developed to allow a Web server to generate content on demand. All dynamic content generation technologies can be thought of as mechanisms for adding new functionality to a Web server. From this point of view, a Web application is simply a complex Web server extension that provides application functionality.

Since J2EE is based on existing Web standards, the technologies described in this chapter remain fully compatible with J2EE implementations. J2EE Web-tier technologies provide a superset of the functionality offered by the older technologies described below. Easy migration from or seamless integration with legacy Web applications is one of the strengths of the J2EE Web tier.

The *Common Gateway Interface*, or CGI, was the earliest standardized dynamic content generation mechanism. For static content, a Web server usually maps a request URL to the location of a file on a disk. With CGI, a URL is instead mapped to a program, which when run generates dynamic content that the server transmits to the browser. CGIs are standalone programs, so they can be written in any programming language, including scripting languages such as Perl, Python, PHP, and tcl; and compiled languages such as C and C++. CGIs have a very simple interface with the server: they read request parameters from standard input, and write generated content (including HTTP headers) to standard output.

Although CGI remains a popular option for Web applications, its limitations have motivated the creation of more tightly-integrated Web-tier technologies. Performance has been a primary limitation of CGI: each HTTP request to a CGI program usually results in the creation of a heavyweight process in the host oper-

ating system. Load balancing, scaling, and high availability are also difficult with CGI, since such features are entirely the responsibility of the CGI developer. CGI developers must also deal entirely with issues of security, multithreading, persistence, application and session state maintenance, resource management, and so on. CGI's simplicity is a double-edged sword: it's easy to understand, but it doesn't offer much support to the developer.

The performance and scalability issues of CGI have usually been solved by moving the server extension from a heavyweight process in the operating system to a program or script running inside the server itself. Each service request then results in a function call, or perhaps in execution of a new thread, which then produces the requested content. Examples of internal server extension technologies include server extension APIs, server-side scripting, and Java servlets.

One early way server vendors enabled server extensions was to offer a *server extension API*. Extension APIs allow developers to create libraries that generate dynamic content. Examples of such APIs include NSAPI (for Netscape servers), Apache extension modules (for Apache), and ISAPI (for Microsoft Internet Information Server). Extension libraries are either statically linked with the server code before deployment, or dynamically loaded by the server at runtime. The Web server maps URLs to the loaded server extensions, and passes service requests to the loaded extension code. The extension code produces dynamic content to satisfy the request. This approach provides tighter integration with the server and alleviates the runtime performance overhead of CGI process creation. Yet server extension APIs are nonportable between server vendors, locking applications into a particular vendor's API and product line. Worse, server extension libraries can compromise system stability, since an extension library crash can take down the entire server.

An improvement to server extension APIs is *server-side scripting*, in which request URLs are mapped to a script that runs inside the server. The scripting engine for a server is essentially a server extension library that executes scripts on demand. *Fast CGI* is a server-side scripting interface that replaces an operating system CGI program with a CGI script running inside a scripting engine in the server. Other scripting languages and engines provide various services to server-side scripts, and the APIs for accessing the scripts vary across servers and vendors. Server-side scripts that fail usually don't crash the server, since the script interpreter can detect script failure, shut the script down gracefully, and report the error in a structured way. Server-side scripts are safer than extension APIs, and the scripts can be more portable, since they are interpreted. But each server-side scripting solution offers a different, incompatible way to interact with the script-

ing engine. Fast CGI scripts are portable, but their interface to the server (beyond that of traditional CGI) is not.

3.3.2 J2EE Web-tier Technologies

J2EE Web-tier technologies provide server extension mechanisms that provide the benefits of server-side scripting in a standardized, secure, and vendor-neutral way. This section defines Web applications and the technologies that support them; specifically, Web archives and the Web-tier component model, including servlets, JSP pages, and the J2EE Web container.

A *Web application* is a collection of Web-tier components, content, and configuration information. A Web application can be deployed into a *Web container* as a single functional unit, which operates as an application. The packaging and deployment unit for a Web application is a *Web archive* (with a filename ending in ".war"), a file which contains all of the class files and resources for the Web application. The Web archive also contains a deployment descriptor file that configures the application.

Once deployed, Web components execute inside a *Web container.* A container is a collection of classes in the server that cooperate to manage the lifecycle of component instances and provide data and services to those instances. A container is said to "contain" the component instances it manages. The J2EE specification defines a contract between the Web container and each Web component, defining the component's lifecycle, the behavior the component must implement, and the services that the server must provide to the component.

J2EE specifies two types of Web components: *servlets* and *JavaServer Pages™ (JSP™).*

A *servlet* is a Java class that extends a J2EE server, producing dynamic content in response to requests from the server. The server passes service requests to the servlet through the standard J2EE extension interface `javax.servlet`, which every servlet must implement.

A *JSP page* is an HTML page with special markup that provides customizable behavior for generating dynamic content at runtime. Although a JSP page is translated into a servlet when it is deployed, JSP page technology provides a document-centric, rather than a programmatic, way to specify dynamic content generation.

3.3.3 The Web Container

The container dispatches service requests to the components it contains, provides the components with context data (such as session state and information about the current request), and mediates the component's generation of response content. The Web container runs inside the Web server, translating HTTP requests to URLs into calls on component instances, and producing HTTP responses based on content generated by the component. The container also manages the component lifecycle, notifying components, by way of callbacks, of state changes such as instance initialization and destruction.

For example, an HTTP request to a servlet's URL causes the Web container to call the service method of an instance of that servlet class. The servlet instance then generates dynamic content, based on data passed in from the request. When the service method returns, the container returns the content to the Web server. The Web server then serves the generated content to the client that initiated the HTTP request.

The next few sections describe the components that may be hosted by a Web container, explain specific useful features of these components, and explain the rationale underlying each of those features.

3.3.4 Servlets

A servlet, as described in the introductory chapter, is a Java platform server extension component that runs inside a Web container. (See "The Web Container," above.)

Each servlet class in a Web application is mapped to one or more URLs in the Web server. When the server receives an HTTP request (GET, POST, etc.) to a servlet's URL, the server calls the corresponding servlet's service method (`service()`), which generates dynamic content for the response.

For example, the URL `http://j2ee-server/servlets/control` might be mapped in the server to an instance of class `AppControllerServlet`, which extends class `HttpServlet`. That servlet instance would handle any HTTP requests directed to its URL, either by producing response data, or by delegating the request to some other component.

Servlets, therefore, are similar to CGIs and server-side scripts, since they are pieces of code that generate dynamic content in response to an HTTP request to a particular URL.

Servlets offer some important benefits over earlier dynamic content generation technologies. Servlets are written in Java, so they run in a JVM instance

embedded inside the Web server. Servlets are generally faster than CGI programs or scripts, since they are run as compiled bytecodes, instead of as a heavyweight process or an interpreted script. Servlets are safer than extension libraries, since the JVM instance can usually recover from a servlet that exits unexpectedly. And servlets are more portable, and provide a richer set of standard services, than server-side scripting technologies.

Compiled servlet classes are portable in binary form across hardware platforms because of the Java platform's inherent runtime portability. In addition, servlet developers can rely on the presence of all standard Java platform APIs. But servlets are also portable across Web servers, because the contract between the servlet and its container is tightly defined by the Java Servlet Specification, and verified (in J2EE-branded products) by the J2EE Compatibility Test Suite. Web server portability provides servlet developers with consistent availability of a rich, vendor-neutral set of standard services, and ensures that servlets will operate correctly across different vendors' product lines. Servlet developers can create products that run properly on all J2EE-branded servers, and J2EE server customers can change technology vendors with little or no change to their application code. Finally, the specification and standardization of packaging and deployment for servlets (and other Web-tier components and resources) ensures that a Web application can be deployed as a single unit into any J2EE-branded server without code modifications or rebuilds.

The following discussion is a general introduction to servlet technology.

Servlet classes and interfaces

All servlets are subclasses of abstract class `javax.servlet.GenericServlet`, which is not protocol-specific. The servlet's `service()` method generates dynamic content when called by the servlet's container in response to a service request. The service method has two arguments, one of type `ServletRequest`, which encapsulates the input to the servlet; and another of type `ServletResponse`, which the servlet uses to communicate its response to the container.

A servlet for a particular protocol subclasses `GenericServlet`, overriding the service method to dispatch calls to its own protocol-specific methods. For example, abstract class `javax.servlet.HttpServlet` has methods for handling HTTP-specific operations; `doGet()` for GET, `doPost()` for POST, and so on. Each such method has arguments of interface types `HttpServletRequest` and `HttpServletResponse`. Through these interfaces, the servlet container communicates HTTP-specific information to the servlet, and receives response data from the servlet, respectively.

Since this chapter focuses on the Web tier, all of the following discussion pertains to HTTP servlets, which fulfill HTTP requests.

A servlet typically creates dynamic content by writing to either a `PrintWriter` (for textual output) or an `OutputStream` (for binary output), obtained from the `HttpServletRequest`. The following example shows how a servlet can create dynamic content in response to an HTTP request.

Example

Code Example 3.1 shows a "Hello, world!" servlet that generates dynamic content in response to an HTTP GET request. The servlet uses a `PrintWriter`, obtained from the `HttpServletResponse`, to communicate its dynamic content. The servlet also indicates the content's type and encoding by calling the response interface's method `setContentType()`.

```
public void HelloServlet extends HttpServlet {
   protected void doGet(HttpServletRequest q,
                        HttpServletResponse p)
                  throws ServletException, java.io.IOException {
      PrintWriter out = p.getPrintWriter();
      p.setContentType("text/text; charset=ISO-8859-1");
      out.println("Hello, world!");
   }
   //... implement other methods ...
}
```

Code Example 3.1 A simple "Hello, world" servlet

This servlet can be compiled into a class file, placed in a "Web archive" (.war) file, and deployed into a Web container within a Web server. The archive must include a deployment descriptor, which is an XML file that configures the relationship between the component and its container. The deployment descriptor includes an entry that indicates where in the server's URL namespace the servlet should reside.

An HTTP GET to the servlet's URL results in a call to the servlet's `service()` method, which in turn calls `doGet()`. The servlet communicates its dynamic content (the string "Hello, world!") to the server through the `PrintWriter`, and the server returns that content to the client.

See the design and implementation guidelines in sections 3.4 and 3.5 of this chapter for recommendations on how to use servlets effectively. (Note that while Code Example 3.1 demonstrates the simplest possible servlet code, it violates the best practice of using JSP pages instead of servlets for generating textual content.)

The next few sections describe servlet features discussed in depth later in the chapter.

3.3.4.1 Request forwarding and inclusion

Often, instead of generating dynamic content, a servlet simply examines the incoming request and dispatches that request to some other component, such as an HTML page, a JSP page, or another servlet. The Front Controller, described on page page 76 in this chapter, is an example of just such a servlet. A servlet redirecting a request to some other component is called *request forwarding*. When a servlet forwards a request to some other component, the server behaves as if that original component, rather than the servlet, had received the request. This allows a servlet to act as a "switchboard", selecting information resources anywhere on the Web, based on information encoded in the request. Front components can also create or modify state before forwarding; for example, placing a timestamp in request scope to indicate when the request was received.

Example

Servlets forward requests with the `RequestDispatcher` interface, implemented by the Web container. An example of a servlet forwarding a request appears in Code Example 3.2.

```
public void Saluez extends HttpServlet {
    protected void doGet(HttpServletRequest req,
                         HttpServletResponse rsp)
                  throws ServletException, java.io.IOException {
        // Set message in request scope
        rsp.setAttribute("message", "Salut, la monde!");

        // Forward request to JSP page that displays message
        ServletContext sc = getServletConfig().getServletContext();
        RequestDispatcher rd = sc.getRequestDispatcher("/msg.jsp");
        rd.forward(req, rsp);
    }
```

```
        //... implement other methods ...
    }
```

Code Example 3.2 A servlet that forwards a request to a JSP page

The sample code shows a simple servlet that sets an attribute "message" in request scope by calling method `setAttribute()` on the `HttpServletRequest`. The servlet then asks the container (via interface `ServletContext`) for a `RequestDispatcher` for the URL `/hi.jsp`. Finally, it forwards the request with `RequestDispatcher.forward()`, passing the original request and response objects as arguments. The JSP page `hi.jsp` will receive a request just as if it had been requested directly by the client, and will presumably display the message in some way. A different servlet might set a different message and redirect to the same JSP page, producing different content.

Web-tier controller servlets, particularly Front Controllers, typically use servlet forwarding to activate the next user view. Servlets may also use forwarding to apply the Decorator design pattern to existing servlets.

In addition to request forwarding, the `RequestDispatcher` has an `include()` method that allows a servlet to include the results of a request to another URL in the current response stream. A servlet can use this mechanism to perform server-side includes; that is, it can perform several requests, assembling the results into a single response.

For more on `forward()` and `include()`, see "Use RequestDispatcher methods forward() and include() correctly," starting on page 137.

3.3.4.2 Servlet filters

A *filter* is a component that can modify request or response data or headers for a Web-tier resource. A filter is unlike other Web-tier components in that, instead of producing a response, it usually either modifies the data or headers of an incoming request, or of an outgoing response, or both. Filters are written by component or application developers, and then configured in the Web application's deployment descriptor to "wrap" other Web resources. Filters are also composable into *filter chains*, which successively apply filters to a request or a response (see *Filter chaining* below).

Example

Code Example 3.3 demonstrates a simple filter. This filter simply sets the request scope attribute "message" to a greeting in Italian. The small XML fragment in the code sample indicates that the filter is to be applied to the URL `/hi.jsp` each time it is accessed. Any client request to `/hi.jsp` will be directed first to the filter's `doFilter()` method. The filter sets the "message" attribute in the request, and then calls the `doFilter()` on the `FilterChain`, which causes either the execution of the next filter in the chain, or a request to `/hi.jsp`, depending on whether this filter is the last in the chain. The result is that the JSP page receives the request with the "message" attribute set, just as in Code Example 3.2, except that the message is in Italian rather than French.

```
[Code]
public class CiaoFilter implements javax.servlet.Filter {
   public void doFilter(ServletRequest q, ServletResponse p,
                        FilterChain c) {
      q.setAttribute("Ciao, mondo!");
      c.doFilter(q, p);
   }
}

[Deployment descriptor]
<!-- Define the filter -->
<filter>
  <filter-name>ciaofilter</filter-name>
  <filter-class>CiaoFilter</filter-class>
</filter>
<!-- Map filter to wrap "/hi.jsp" -->
<filter-mapping>
   <filter-name>ciaofilter</filter-name>
   <url-pattern>/hi.jsp</url-pattern>
</filter-mapping>
```

Code Example 3.3 A simple filter and its deployment configuration

Filters are useful for adapting existing Web resources to new contexts (the Adapter pattern), for adding new functionality to existing resources (the Decora-

tor pattern), and as Adapters that transform the results of a resource into a new format.

Filters are self-contained decorators that add functionality to a servlet. When a servlet filter decorates an already-filtered servlet, the result is a *filter chain*.

Filter chains

Filters can be applied in series to a request and/or response. A sequence of filters, called a *filter chain*, performs a series of successive transformations on Web-tier resource requests, responses, or both. Filter chains are composed of instances of filters. As show above, filters are configured in the deployment descriptor for the web application. A filter chain is an ordered series of filter mappings, in the order they are defined in the deployment descriptor.

Example

The following excerpt from a Web application deployment descriptor defines a filter chain. (Assume that the filters are defined elsewhere with a `<filter>` tag.)

```
<filter-mapping>
    <filter-name>Gif2Tiff</filter-name>
    <url-pattern>/images/*.gif</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>TiffGrayscale</filter-name>
    <url-pattern>/images/*.gif</url-pattern>
</filter-mapping>
```

Code Example 3.4 A deployment descriptor excerpt that defines a filter chain

These two mappings define a filter chain that applies two filters to all files matching `/images/*.gif` in the application: first, a filter called `Gif2Tiff`, which presumably converts a GIF file to TIFF format; and second, `TiffGrayscale`, which converts all color pixels in a TIFF image to gray values. The result would be that all GIF files in the application's `/image` directory would be served as grayscale TIFF files. Notice that the filters may change not only the format of the requested data (from GIF to TIFF, and from color to grayscale), but also the HTTP headers. The `Gif2Tiff` filter would probably set the response Content-Type header to `image/tiff`.

You may have heard the term “servlet chaining”, which refers to a now-obsolete feature of some Web servers. Filter chaining supersedes servlet chaining.

3.3.5 JavaServer Pages™ (JSP™)

While servlets are portable, powerful, and extremely flexible, they have some limitations in practice. Most of the dynamic content in Web applications is composed primarily of “parameterized” HTML pages that, when served, change only in data values, and not in basic structure. Servlets that produce such content can be problematic for the following reasons:

- *Servlets can be tedious to write.* Many servlets that produce HTML are little more than Web pages with `System.out.println()` wrapped around each line, backslashes before each double-quote, and maybe a bit of logic or an iterator here and there. Such servlets are tedious to write, and difficult to read and maintain.
- *Servlets are programs.* For all but the simplest cases, servlets require programmers for development and maintenance. Programmers are not necessarily skilled in graphic design, and graphic designers are often not programmers. (Unfortunately, technology can’t solve the human resource problem of programmers who mistakenly think they are also graphic designers, or graphic designers who wrongly believe they are also programmers. Few people are truly skilled in both areas.) Servlets that combine program logic with sophisticated visual design can be difficult for both programmers and graphics designers to modify without introducing flaws.
- *Servlet code is not easily reusable*. HTML-based Web applications usually include a great deal of repetition. Many use cases occur multiple places in a Web application. HTML pages often contain repeated style elements to provide a consistent look and feel. But the servlet API provides little direct support for reusing either business or presentation logic. The result is usually a cut-and-paste coding style that sacrifices consistency and maintainability.
- *Servlets mix functional code with presentation code.* While this limitation may seem a bit abstract, it’s actually the issue at the root of all of the other problems. The basic problem is that business logic and data presentation are fundamentally different concerns. Mixing the two in a single technology (the servlet) results in confusion in design, implementation, and maintenance.

To address these difficulties, J2EE offers JavaServer Pages™ technology, also known as "JSP pages".

A JSP page is a template document containing special markup for executing embedded logic. JSP pages can be used to specify dynamic content of any type, but they are primarily used for creating structured content (such as HTML, XML, XHTML, PDF). JSP pages address the above servlet difficulties in the following ways:

- *JSP pages are easier to write than servlets.* JSP pages look very much like documents of the type they produce; a JSP page that produces HTML *looks like* HTML, with minor additional markup. No repeated `println()` statements or backslash escape characters are necessary.
- *JSP pages don't (always) require programming skills.* Because the JSP page looks like the document type it produces, an expert in that document format can edit it directly, even if that person is not a programmer. Any actual programming is more or less separate from the presentation code, so programmers and document format specialists can work relatively independently.
- *JSP pages support code reuse.* JSP pages have explicit code reuse mechanisms. There are special tags for using JavaBeans components from a JSP page, and programmers can define custom tags for specialized purposes. Include directives in the JSP tag language provide both compile-time and runtime reuse of document content. These mechanisms can improve code quality by reducing repeated code.
- *JSP pages separate function from form.* The JSP page format is a template for a document, with business logic embedded either in special markup in the JSP page (as scripting elements), or preferably in beans and/or custom tags. Separating function (business logic) from form (presentation) allows programmers and document authors to focus on their core skills. This separation of developer roles is one of the major benefits of the J2EE platform.

A JSP page looks to its author like a document, but Web containers actually implement JSP pages as servlets. Before a JSP page can be run in a JVM instance, it must be converted (by a translator) into a servlet written in the Java language, which is then compiled into a loadable class. The JSP translator handles the tedious and error-prone process of wrapping code lines in `println()` statements, backslash-escaping special characters, and so on. It also provides the code that performs server-side includes, evaluates expressions, invokes custom tags, and so

forth. A JSP is usually deployed directly to a Web container, which handles the translating and compiling.

JSP pages differ from servlets in their programming model. A JSP page is primarily a document that specifies dynamic content, rather than a program that produces content. JSP page technology provides a "document-centric" alternative to "programmatic" servlets for creating dynamic, structured data. That a JSP page is in fact implemented by a servlet is an implementation detail.

3.3.5.1 JSP Page Syntax

This section contains only a brief, incomplete introduction to JSP syntax. See the Resources section at the end of this chapter for references to details on JSP syntax and usage.

Directives, Actions, and Scripting Elements

JSP pages consist of fixed template data with special embedded markup for producing dynamic data. The template data is specified in the content format that the page produces (for example, HTML). Each piece of this special markup is called a *JSP element*. A JSP element either produces dynamic data for substitution within the template, or controls the translation or dynamic content generation process.

The JSP page description language has three types of elements:

- *Directives* are instructions that control the operation of the JSP translator, so it has an effect only while the JSP is being translated into servlet code. For example, the `include` directive lexically includes another file during translation, much as a C preprocessor `#include` directive does during preprocessing. Directives are delimited by the tokens `<%@` and `%>`. The directive to include the JSP file "header.html" at JSP page translation time would look like this:

```
<%@ include file="header.html" %>
```

- *Actions* are special tags that perform document-related tasks such as including another document, forwarding a request, or defining and manipulating server-side JavaBeans components. Unlike a directive, an action translates to Java code in the implementing servlet, and so an action's effects occur at runtime. Standard JSP actions are implemented by the JSP runtime support classes. They are required to use the XML namespace `jsp:`, to ensure compatibility with XSDL-validating XML processors. The following action includes the file

`header.jsp` into the current JSP page being output:

```
<jsp:include page="header.jsp"/>
```

Note that this include action behaves differently from the include directive shown above. The include directive lexically includes static content into the JSP at translation time. The include action includes either static or dynamic content into the JSP page's output at runtime. See the guideline "Use JSP include directives and actions appropriately," starting on page 132, for a discussion of this topic.

In addition to the standard actions defined in the specification, developers can add new actions, called *custom tags*. (See the topic *Custom tags* below.)

- *Scripting elements* produce code fragments in the servlet generated from the JSP page. Since the scripting elements translate to code, they are executed at runtime. The three types of scripting elements are *declarations* for defining fields and methods; *expressions* for outputting data values; and *scriptlets* for executing arbitrary code. Descriptions and examples of these three scripting element types appear below.

Declarations, Expressions, and Scriptlets

The three types of scripting elements are each lexically inserted into specific places in the servlet generated from the JSP page. While they all execute at runtime, their scope and precisely when they execute differs.

A *declaration* defines variables and methods in class scope. It produces no output from the JSP page's servlet. Any initializers execute whenever the initializers run for the servlet class. Declarations are delimited by the tokens `<%!` and `%>`. For example, the following line defines and initializes to 0 a private field `chapter` in the servlet class:

```
<%! private int chapter = 0; %>
```

An *expression* is an expression, written in the Java language, that is evaluated in the scope of the service method of the servlet translated from the JSP page, and is immediately written to the servlet output. It executes as the page output is being assembled by the servlet. Expressions are delimited by the tokens `<%=` and `%>`. The following line in a JSP page uses the value of the variable `chapter` (defined above) to output a chapter number as a list element (the expression is in italics for emphasis):

```
<LI>Chapter <%= chapter + 1 %></LI>
```

A *scriptlet* is a chunk of Java code that is lexically inserted into the service method of the servlet that implements the JSP page. Unlike an expression, however, a scriptlet produces no output directly. Scriptlets are delimited by the tokens <% and %>. Code example 1.5 below demonstrates how to use scriptlets, expressions, and declarations together to print the table of contents of a book. (For emphasis, scriptlets are in boldface, and the declaration and expressions are in italics.)

```
<%! private int chapter = 1; %>
<H1>Table of Contents</H1>
<UL>
<%  String[] chapters = getChapters(); // method defined elsewhere
    for (chapter = 0; chapter < chapters.length; chapter++)
    {
%>
<LI>Chapter <%= chapter + 1 %>: <%= chapters[chapter] %></LI>
<% } %>
</UL>
```

Code Example 3.5 A JSP page fragment with scriptlets, expressions, and a declaration

Scriptlets should be avoided, or at least used sparingly, in large applications. See the guideline “Avoid scriptlets and expressions,” starting on page 142, for a discussion of the liabilities of scriptlets.

Localization

JSP pages rely on Java’s built-in support for localizable content. Java represents characters internally as Unicode (ISO-10646). A JSP’s character encoding defaults to Latin-1 (ISO-8859-1). A page’s encoding can be set statically using the `page` directive’s `pageEncoding` attribute, or dynamically using method `ServletResponse.setContentType()`.

JSP Page Syntax and XML

A JSP page in JSP syntax cannot be well-formed XML, because the JSP syntax violates the lexical rules of XML 1.0. The JSP 1.2 specification does, however, offer an alternate, well-formed XML syntax, complete with an XML DTD (Document Type Definition).

The benefits of using XML syntax for JSP pages include document instance validation and superior tool integration. JSP pages using the XML syntax are directly deployable, without preprocessing into JSP syntax, to JSP 1.2-compliant containers. See the JavaServer Pages™ 1.2 Specification, Chapter 5, *JSP Documents*, for details.

3.3.5.2 Manipulating State With JavaBeans™ Components in JSP Pages

JSP pages can use JavaBeans components to maintain state just as servlets can. The JSP specification defines three actions for manipulating Web-tier beans from JSP pages. These actions are provided as a convenience: everything they do could be done with a combination of scriptlets and expressions.

Using these actions, JSP page can create and initialize a JavaBean instance and store it in a particular scope. Thereafter, the bean is accessible elsewhere on the page, as well as to other pages in the same scope. The `jsp:useBean` action references or creates a named instance of a bean in a particular scope. The actions `jsp:setProperty` and `jsp:getProperty` assign or retrieve a property value from a bean. The syntax for these actions appears below.

```
<jsp:useBean name="name" scope="scope" class="className"/>
```

The `jsp:useBean` action references a bean in some scope of the servlet generated from the JSP page. The name of the bean is an identifier used to reference the bean in other actions. The value of the `scope` attribute must be either `page` (meaning the bean applies only to the current page), or one of `request`, `session`, or `application`, meaning the same thing those terms mean for servlets. The bean's type is indicated here by the `class` attribute. The `jsp:useBean` action is powerful and somewhat complex; see section 4.1 of the JSP 1.2 specification for details.

```
<jsp:setProperty name="name" property="prname" value="value"/>
<jsp:setProperty name="name" property="prname" param="paname"/>
<jsp:setProperty name="name" property="*"/>
```

The `jsp:setProperty` action sets the value of a property or properties of a bean in the Web tier. The `name` attribute indicates the bean's name, as defined

elsewhere by a `jsp:useBean` action, and which is retrieved by method `PageContext.findAttribute()`. The `property` attribute indicates the name of the property to set. If a `value` attribute is specified, the property is set to that value, as shown in the first example above. (The value of the `value` attribute can be either a literal string value or a scriptlet expression. If an expression, it is called a *request-time attribute value*; see JSP 1.2 section 2.13.1 for details.) If a `param` attribute is specified, the property is set to the value of the current request parameter with that name (example 2 above). To simplify implementation, the special property attribute value "`*`" indicates that all bean properties should be set to the values of the current request parameters with matching names (example 3). Section 4.2 of the JSP specification covers the syntax and semantics of `jsp:setProperty`.

```
<jsp:getProperty name="name" property="prname"/>
```

The `jsp:getProperty` action outputs the value of the indicated property into the JSP's response stream. It is equivalent to an expression that retrieves and outputs the corresponding property value, defined and named elsewhere as a server-side bean. For example, consider the following alternatives:

```
<%= pageContext.findAttribute("user").getName(); %>
<jsp:getProperty name="user" property="name"/>
```

These two lines produce identical output, but the latter is well-formed XML, is more readable, and is language-neutral.

Example

The following code example shows a simple page that creates a User object to session scope used by later actions and scripting elements.

```
<jsp:useBean name="user" scope="session" class="myapp.User">
   <jsp:setProperty name="login" param="login"/>
   <jsp:setProperty name="password" param="passwd"/>
</jsp:useBean>

<h2><jsp:getProperty name="user" property="login"/> logged in</h2>
Welcome, <jsp:getProperty name="user" property="fullName"/>!
```

Code Example 3.6 Part of a JSP page that creates and uses a server-side bean

In Code Example 3.6, the `jsp:useBean` action creates a new User bean in session scope, and the enclosed `jsp:setProperty` actions initialize it. The first two such actions set property values based on named parameters in the request. The `jsp:getProperty` action within the heading fetches and outputs the `login` property, which was set when the `User` object was created. Notice that the `fullName` property, while not initialized by `jsp:setProperty`, is nevertheless available for the second `jsp:getProperty` action. This demonstrates that beans can act as more than just passive data containers; presumably, the `User` class is capable of somehow looking up a user's full name, given a login and password. In fact, class `User` may well be implemented by an enterprise bean.

While the JSP page JavaBean tags shown above are widely used, J2EE BluePrints recommends instead creating your own tags for your application, which store their own state where appropriate. The next section describes how to create custom tags.

3.3.5.3 Custom Tags

The standard JavaBean actions described above are reusable, well-formed, language-neutral markup tags that replace scripting elements in a JSP page. Pages using these standard actions are easier to read, because they express a page in terms of standard markup tags, instead of as a jumble of tags, special delimiters, and source code fragments.

In fact, these benefits are so compelling that JSP technology allows developers to define their own actions, called *custom tags*. Custom tags are defined and implemented by component and/or application developers, and used by page designers in JSP pages. Programmers define the syntax for a tag and implement the tag's behavior as a class. Page designers can then import and use sets of tags just as they use standard actions.

For example, a programmer might define a `today` tag that evaluates to the local time. The page designer could then use the tag in a JSP page, instead of using a scripting element. A scripting element to output the current date might look like this:

```
Today's date is:
<%
DateFormat df = DateFormat.getDateInstance();
Date today = new Date(System.currentTimeMillis());
out.print(df.format(today));
%>
```

The custom tag solution looks like this:

```
Today's date is: <dt:today/>
```

Custom tags provide several benefits. First, they are reusable, as scripting elements generally are not. Custom tags ease maintenance, since changing a tag's handler class changes the tag's behavior everywhere the tag is used. JSP pages in XML syntax that use custom tags can be validated against a DTD or XSDL schema, improving application quality and facilitating integration with development tools. Custom tags allow graphic designers, familiar with markup tags but possibly not with programming, to work exclusively with tags, instead of with a mixture of tags and cryptic scripting elements. Perhaps most importantly, custom tags can decouple data presentation (in JSP pages) from the program code implementing the presentation and business logic (in *tag libraries*, see below). This separation eases maintenance, clarifies the intent of each component, and allows programmers and page designers to work relatively independently of one another.

Defining a tag library

There are several discrete steps to defining a tag library. A developer defines a custom tag's syntax (its name, attributes, and other constraints) in a *Tag Library Descriptor* (TLD) file. (JSP 1.2 specification section 7.4 describes the TLD format; its DTD appears in Appendix C.) The developer implements the tag by writing a *tag handler class* that implements interface `javax.servlet.jsp.tagext.Tag`. A *tag library* is the deployment unit for the handler classes of the tags defined in a TLD file.

Once a tag is defined and implemented, a page designer uses a *taglib directive* in the JSP page to import the tags and associate a namespace prefix with them. Thereafter, the page designer can use the custom tag in the same way a standard action would be used.

Example

This example will show the creation and usage of the `today` tag example shown above.

1. Defining the tag within a TLD file is straightforward. The following XML fragment is a valid taglib descriptor:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

```
<taglib>
    <tlib-version>1.0</tlib-version>
    <short-name>DATETIME</short-name>
    <tag>
        <name>today</name>
        <tag-class>myTags.TodayTag</tag-class>
    </tag>
</taglib>
```

2. The next step is to implement the tag as a class. The TagSupport class provides default implementations that subclasses selectively override.

```
public class TodayTag extends TagSupport {
    public TodayTag() { super(); }
    public int doStartTag() {
        DateFormat df = DateFormat.getDateInstance();
        Date today = new Date(System.currentTimeMillis());
        try {
            pageContext.getOut().print(df.format(today));
        } catch (IOException e) {
            // Handle exception...
        }
        return EVAL_BODY_INCLUDE;
    }
}
```

3. The above class can be packaged into a tag library in the form of a jar file, which is deployed with the Web application.

4. The page designer imports the tag library into the page with a taglib directive, indicating the path to the library and a namespace prefix for the tags. The namespace prefix is entirely arbitrary, and is specific to the document where it is declared:

```
<%@ taglib uri="/datetime" prefix="dt"/>
```

5. The tags defined in the tag library can be used anywhere in the file that any other tag may occur:

```
<h1>Sales report for <dt:today/></h1>
```

3.3.6 Standard Tag Libraries

Standard tag libraries are sets of custom tags that provide a basic set of domain-neutral functionality for JSP pages. Standard tags typically perform such functions as Web resource inclusion, request forwarding, conditional logic, collection iteration, XSLT transformations, internationalization, state access, and HTML forms. Some companies have produced standard tag libraries that are intimately integrated with their tools and J2EE product lines. Other organizations have produced standard tag libraries for general use in J2EE applications. Apache Struts (see page 123), for example, includes a large library of standard tags.

Standard tag libraries often provide much of the basic functionality your JSP pages need. Mature libraries have been tested and optimized by a community of developers. Adopting a high-quality standard tag library can save application development time.

For implementation guidelines, see "Avoid heavy use of logic tags" on page 132.

3.3.7 Web-tier state

A servlet, servlet filter, or JSP page usually creates its response with data contained in the service request, but it often needs data from other sources to do its job. For example, a JSP page that lists items for sale needs access to inventory data, or a servlet may maintain the contents of an online shopping cart. A servlet filter might place a timestamp on an incoming request and forward that request to another Web component (see *Request forwarding* below).

Data that a Web-tier component uses to create a response is called *state*. Each item of Web-tier state has a *scope*, which determines the accessibility and lifetime of the item. Web-tier state is accessible to servlets, servlet filters, and JSP pages. Briefly, there are four types of state scope in the Web tier:

Application scope is "global memory" for a Web application. Application-scope state is stored in the Web container's `ServletContext` object. Objects in application scope are shared between all servlets in an application. Thread safety is the responsibility of the servlet developer. An inventory object in application scope, for example, is accessible to all servlets, servlet filters, and JSP pages in the application. State in application scope exists for the lifetime of the application, unless it is explicitly removed.

Session scope is data specific to a particular user session. HTTP is a "stateless" protocol, meaning that it has no way of distinguishing users from one another, or for maintaining data on users' behalf. The servlet API offers mecha-

nisms for associating data with a user session and accessing or modifying that data in subsequent requests. Session-scope state for an `HttpServlet` is stored in the Web container's `HttpSession` object (available from the `HttpServletRequest` argument to the service method). State in session scope is accessible to all Web components in the application and across multiple servlet invocations, but is accessible only within an individual user session. An online shopping cart is an example of data in session scope, since the contents of the cart are specific to a single client session, and available across multiple server requests. Unless removed explicitly, state in session scope lasts until the session ends. A session ends when it is explicitly closed, when it times out after a period of inactivity, or when its container is shut down or crashes.

Request scope is data specific to an individual server request, and is discarded when the service method returns. A Web component can read or modify data in request scope and then "forward" the request to another component. The component to which the request is forwarded then has access to the state. State in request scope is stored in a `ServletRequest` object, so it is accessible to any component receiving the request. Note that the values of get parameters and form variables are also in request scope. A servlet that places a timestamp in the `ServletRequest` object and then forwards the request to another servlet is an example of request-scope state. Request scope state is discarded when the server transmits the response to the request.

Page scope, applicable only to JSP pages, indicates that the data are only valid in the context of the current page. Page scope state is stored in a JSP page's `PageContext` object. When one JSP forwards to or includes another, each page defines its own scope. Page scope state is discarded when the program exits the page.

There are a variety of ways to store state in each scope. In-depth design discussions and guidelines for Web-tier state maintenance are treated in detail in section 3.4.1.10, "Web-tier state maintenance," starting on page 126.

3.3.7.1 Using JavaBeans™ Components with servlets

Web applications commonly layer access to the MVC model in the form of Web-tier JavaBeans components. Web-tier JavaBeans provide servlets and other Web-tier components with access to business logic and data. Servlets can store Web-tier JavaBeans in application, session, or request scope.

For example, a servlet can create a Web-tier Inventory component in application scope, where later servlet invocations will find it, ready for use. A `Shopping-`

`Cart` object placed in session state will be accessible to later servlet invocations within the same user session; each session will have an associated `ShoppingCart`.

JSP pages have special tags for accessing Web-tier JavaBeans; see "Manipulating State With JavaBeans™ Components in JSP Pages," starting on page 94; and "Maintain Web-tier session state with HttpSession," starting on page 137.

3.4 Web-tier Design Issues and Guidelines

The first part of this chapter described and demonstrated J2EE Web-tier technologies. The remainder of the chapter covers issues and guidelines for enterprise application design and implementation.

This section is about design: choosing system structure, software layering, tier selection, object distribution, and so on. Design concepts are about how a system works, not about implementation details. For coding advice, see "Implementation Issues and Guidelines," starting on page 131.

3.4.1 Web-tier Design Issues

Design issues are discussions of larger-scale problems encountered in a design, whose solutions depend on the application requirements and context. Issues are explorations of a particular concern and the context on which design choices depend.

3.4.1.1 Web-tier technology selection

Servlets and JSP pages are both Web-tier components, but typically play different roles in a design. XML can be used as a "hub" format for data interoperability, and can be styled with CSS or XSL on either the client or server. This section explains the design situations appropriate for both JSP pages and for servlets.

When to use JSP pages

JSP pages are most often used either as presentation components, playing the View role in an MVC application, or for creating structured, non-visual content such as XML messages.

When to use servlets

Servlets are primarily used as MVC controller components, as framework support components (like template processors, security adapters, loggers, etc.), or as presentation components for binary content.

Generating XML

XML can be generated either by JSP pages or by servlets. Which technology to choose depends on what sort of data the component is generating.

XML in interactive Web applications

While there has been a great deal of attention paid to using XML to create non-interactive Web services, XML can also be useful in interactive applications. A Web-tier controller could translate HTTP requests into invocation of Web service methods, and could use CSS, XSL, or custom code to transform the results of the invocation into the requesting client's presentation format. This approach is advisable only if there is no faster, more reliable, or lighter-weight way to access the application model, such as accessing underlying enterprise beans directly.

An interactive application can produce views as XML, and style that XML with CSS or XSL on the server or on the client. We advise this approach only if the XML's benefits outweigh the performance cost of generating, parsing, and transmitting the XML.

XML is also very useful in the internals of Web applications, particularly in configuring relationships between architectural and application components. (The use of XML as the format for J2EE deployment descriptors is a case in point.) The Java Pet Store configures view templating, request processing, and view selection with XML files deployed with the application.

3.4.1.2 "Model 1" vs. "Model 2" Architecture

The literature on J2EE Web-tier technology frequently uses the terms *Model 1* and *Model 2* without explanation. This terminology stems from early drafts of the JSP specification, which described two basic usage patterns for JSP pages. While the terms have disappeared from the specification document, they remain in common use. Model 1 and Model 2 simply refer to the absence or presence (respectively) of a controller servlet that dispatches requests from the client tier and selects views.

A Model 1 architecture consists of a Web browser directly accessing JSP pages in the Web tier. The JSP pages access Web-tier JavaBeans that perform

business operations on the data model, and the next view (JSP page, servlet, HTML page, etc.) to display is determined either by hyperlinks selected in the source document, or by parameters passed to GET or POST operations. The controller in a Model 1 architecture is decentralized, since each JSP page or servlet processes its own inputs (parameters from GET or POST), and the next page to display is determined by the current page being accessed. Some Model 1 architectures choose the next page for display in scriptlet code, but this usage is considered poor form. (See the design guideline "Avoid scriptlets and expressions," starting on page 142.)

A Model 2 architecture introduces a controller servlet between the browser and the JSP pages or servlet content being delivered. The controller centralizes the logic for dispatching requests to the next view, based on the request URL, input parameters, and application state. The Model 2 architecture decouples JSP pages and servlets from one another, and places responsibility for page flow within the controller. The *Front Controller* J2EE design pattern describes the benefits of centralizing all application request dispatching in a single servlet. The controller is also an application of the *Mediator* pattern, since it decouples view components from one another.

J2EE BluePrints recommends the Model 2 architecture for most applications. Model 2 applications are easier to maintain and extend, since views do not reference each other directly. The Model 2 controller servlet provides a single point of control for security and logging, and often encapsulates incoming data into a form usable by the back-end MVC model.

While the Model 2 architecture adds some complexity to an application, an MVC application framework can greatly simplify implementing a Model 2 application. MVC frameworks such as Apache Struts and JavaServer Faces include a configurable front controller servlet, and provide abstract classes that can be subclassed to handle request dispatches. Some frameworks include macro languages or other tools that greatly ease implementation of Model 2 applications.

The Model 1 architecture can provide a cleaner design for small, static applications. Model 1 architecture is suitable for applications which have very simple page flow, have little need for centralized security control and/or logging, and change little over time. Model 1 applications can often be refactored to Model 2 if application requirements change.

When to switch from Model 1 to Model 2

When you find that the JSP pages in your Model 1 application is using scripting elements, custom tags, or JavaScript to perform `RequestDispatcher.forward()` operations, it's time to start considering refactoring the application to Model 2.

Model 1 architectures are best when the page navigation is simple and fixed, and when the structure of the pages in the application can be managed by a simple directory structure. Such applications usually embed the page flow information in the links between the pages. The presence of `forward()` in a JSP page implies that there is logic embedded in the page making a decision about the next page to display.

Over time, as the application grows and changes, page flow logic accumulates. The application becomes difficult to maintain because the page flow logic is distributed across multiple pages. The best time to switch from Model 1 to Model 2 is before this maintenance problem arises. This is why J2EE BluePrints recommends choosing Model 2 from the outset, based on an existing Web controller framework that meets your needs. Model 1 remains a viable options for simple, static applications.

3.4.1.3 Web-tier MVC frameworks

As the Model 2 architecture has become more popular, quite a number of Web-tier MVC frameworks have arisen. MVC frameworks are a layer of software between the application and Web tier interfaces that simplify creating Web applications.

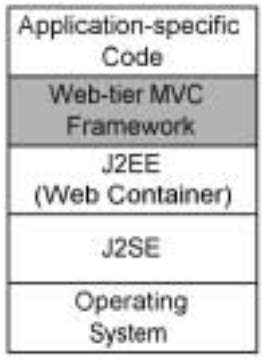

Figure 3.1 A stack of services support application code

Figure 3.1 shows a "stack" of services that support an application. The J2SE platform provides the Java platform on top of the host operating system, and the

Web container of the J2EE platform provides services related to serving Web requests. A Web-tier MVC framework, composed of structural classes, handles the details of such operations as dispatching requests, activating model operations, and selecting and assembling views. A Web-tier MVC framework makes Web-tier technologies easier to use, helping application developers to concentrate on business logic. Several Web-tier MVC framework packages have appeared since J2EE's debut.

J2EE BluePrints recommends that you select an existing Web-tier MVC framework for your application, instead of designing and building a Web-tier layer yourself. Depending on the package you choose, a Web-tier MVC framework can provide the following benefits to your application:

- *Decouples presentation and logic into separate components.* Frameworks encourage separating presentation and logic because the separation is designed into the extension interfaces.
- *Separates developer roles.* Application frameworks generally provide different interfaces for different developers. Presentation component developers tend to focus on creating JSP pages using custom tags, while logic developers tend to write action classes, tag handlers, and model code. This separation allows both types of developers to work more independently.
- *Provides a central point of control.*The controller servlets for most frameworks have a rich, customizable set of application-wide features such as templating, internationalization, access control, and logging.
- *Facilitates unit testing.* Since framework component interfaces are consistent, automated testing harnesses are easy to build and execute.
- *Can be purchased instead of built.* Time not spent developing structural code is available for developing business logic.
- *Provides a rich set of features.* Adopting a framework can leverage the expertise of a group of Web-tier MVC design experts. The framework may include useful features that you do not have the experience to formulate, or the time to develop.
- *Encourages the usage and development of reusable components.* Over time, developers and organizations can accumulate and share a toolbox of preferred components. Most frameworks incorporate a set of custom tags for view construction.

- *Provides stability.* A framework created and actively maintained by a large organization, and used and tested in a large installed base, is likely to be more stable than something you create yourself.
- *May simplify internationalization.* Most frameworks support a flexible internationalization strategy.
- *May support input validation.* Many frameworks have consistent ways to specify input validation. Validation is commonly available on the client side, on the server side, or both.
- *May be compatible with tools.* Good tools can improve productivity and reliablity. Some frameworks are integrated with rapid application development tool sets.

All of these benefits come at a cost, of course. You always have less control over a design you've acquired rather than formulated yourself. Some frameworks have to be purchased, though these are usually bundled with a tool set. Other people's code in your application means other people's *bugs* in your application. Yet most development projects find that quality is improved by the inclusion of a Web-tier framework.

The following sections describe a specific design approach for implement a Web-tier MVC architecture, as implemented in the Web Application Framework of the Java Pet Store sample application. Many of the key classes described (the controller, the templating service, the abstract action class, and so on) could be used for any application, not just for a pet store. Such classes are structural: they're like the load-bearing members of a building.

3.4.1.4 Web tier architectural issues

Most enterprise-level applications will benefit from the flexibility and extensibility offered by a Model 2 architecture. This section discusses some architectural concerns that arise in the Web-tier portion of a Model 2 MVC application.

Figure 3.2 The Web-tier controller in an MVC application

Figure 3.2 shows a block diagram of a Web-tier controller mediating communication between a Web client and an MVC application's model. In the diagram, the controller receives [1] an HTTP request from the client, and maps that request to [2] a method invocation (or series of invocations) on the application model. The controller uses the [3] result returned from the model, plus possibly state maintained in the controller, to [4] select the next view to display to the client.

This general diagram implies some important design questions, each of which are discussed in the sections immediately following:

- An application design must have a strategy for serving current and future client types.
- An application design can be structured to manage the complexity of the Web-tier controller's tasks. These tasks include mapping requests to application model operations, selecting and assembling views, and managing screen flow.
- Application model API design and technology selection have important implications for an application's complexity, scalability, and software quality.
- Choosing an appropriate technology for generating dynamic content improves development and maintenance efficiency.

3.4.1.5 Serving Multiple Client Types

A Web client uses HTTP to communicate with the Web tier. Application requirements and resources drive Web client selection. Web applications may support only one type of client with a single protocol, or multiple clients with different protocols, security policies, presentation logic, and workflows. Web clients may include several versions of a few different browsers, MIDP clients, so-called "rich" clients with standalone APIs, and Web service interfaces. Long-lived applications may need to be able to handle types of Web clients that do not exist when the application is developed.

Following are some options for how to service requests from clients that use different application-level protocols. (Web tier clients all use HTTP for transport.) Each of the following alternatives expands upon Figure 3.2 by adding complexity and flexibility.

.

Figure 3.3 Using a Front Controller to handle browser interaction

Applications with a single client type can implement a single front controller. For example, a browser-only application, as shown in Figure 3.3, can use a single Front Controller servlet to receive HTTP requests from the browser, translate the contents of these requests into operations on the application model, and serve result data views as HTML (or XML). Future client types can be accommodated by creating a new controller for each client type, as shown in Figure 3.4.

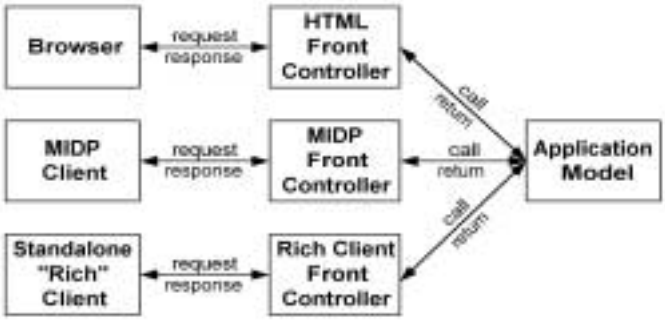

Figure 3.4 Supporting multiple client types with multiple controllers

The multiple-controller approach in Figure 3.4 provides extensibility for any future Web client types, including those that do not yet exist. In fact, since servlets aren't limited to HTTP, even non-Web clients can be supported by this architecture. A further benefit of this approach is that each controller can implement the workflow, presentation logic, and security constraints unique its client type. Notice also that the code implementing the application model is shared by all of the controllers. Sharing the application model code between controllers ensures identical application behavior across client types and eases maintenance and testing.

Some application functionality, particularly security, can be easier to manage from a single point. Introducing a protocol router as shown in Figure 3.5 can

provide a single point of control for all Web clients, each of which still retain their own controllers.

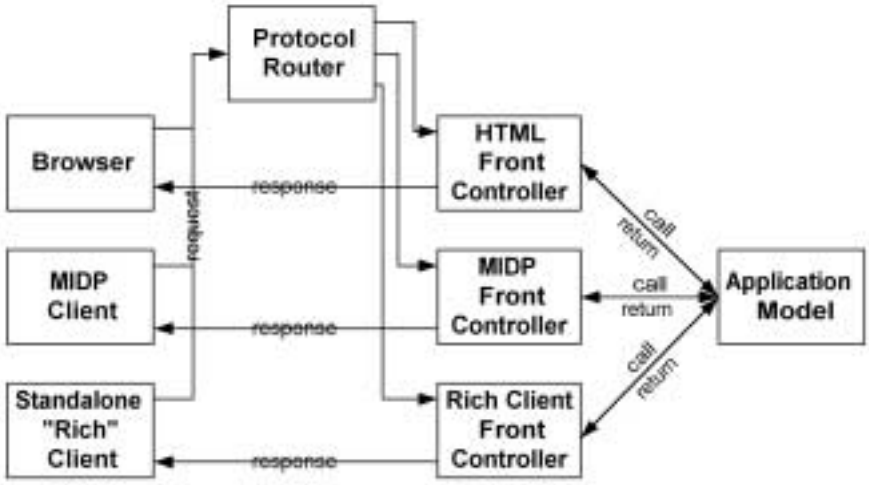

Figure 3.5 Using a protocol router for centralized control

The protocol router in Figure 3.5 is either a servlet or servlet filter that determines the client type based on HTTP request headers (usually the header `User-agent` indicates what sort of client is requesting service) and dispatches the request to the appropriate controller. Application-wide functionality such as security enforcement or logging can be implemented in the router and controlled from a single point. Behavior specific to a client type can be implemented in the controller for the client's particular protocol.

If the other Front Controllers in Figure 3.5 are servlets, the requests are dispatched using `RequestDispatcher.forward()`. But if the router is a servlet itself, it has already handled the HTTP request processing, so the Front Controllers can be simple Java objects to which the router delegates request processing.

Note that the client support alternatives shown above can be implemented incrementally. Each of the approaches can be built on the preceding one. Choose and adapt the alternative that most closely matches your current needs, and add new functionality if and when it is required. In general, it's best to choose the simplest architecture that works for your current needs.

These strategies for serving clients all use Web-tier controllers, the topic of the next section.

3.4.1.6 Web-tier Controller Design

A Web-tier MVC controller maps requests from the client tier to operations on the application model, and selects client views based on model and session state. Web-tier controllers have a lot of duties, so their internal complexity must be managed with careful design. Since most enterprise applications grow over time, extensibility is an important requirement. This section describes some strategies for the internal structure of a controller in the Web tier.

Identifying the operation to perform

When a controller receives an HTTP request, it needs to be able to distinguish what application operation is being requested. How can the client, for example, request that the server create a new user? There are several ways to indicate to the server which operation to perform. The more common methods include:

- Indicate the operation in a hidden form field, which is delivered to the controller via an HTTP POST operation; for example:

```
<FORM METHOD="POST" ACTION="http://myServer/myApp/myServlet">
    <INPUT TYPE="HIDDEN" NAME="OP" VALUE="createUser"/>
    <!-- other form contents... -->
</FORM>
```

- Indicate the operation in a HTTP GET query string parameter; for example:

```
http://myHost/myApp/servlets/myServlet?op=createUser
```

- Use a *servlet mapping* to map all URLs with a particular suffix or base URL to a specific servlet. A servlet mapping is a deployment descriptor definition that matches request URLs and dispatches those requests to servlets. A servlet mapping defines a pattern that is compared with each request URL beginning with a servlet's context path (the "root" path of the servlet). All matching requests are directed to the corresponding servlet. For example, imagine that a Web application's deployment descriptor defines the following servlet mapping:

```
<servlet-mapping>
    <servlet-name>myServlet</servlet-name>
```

```
    <url-pattern>*.op</url-pattern>
</servlet-mapping>
```

Imagine also that the servlet's context path is `http://myServer/myApp/servlets`. Then the request URL `http://myServer/myApp/createUser.op` will match the URL pattern `*.op`, and the request will be directed to `myServlet` for processing. Servlet `myServlet` can extract the name of the operation to be performed from the request URL. Servlet mappings are defined in Chapter 11 of the Java Servlet 2.3 specification.

Of the three options above, J2EE BluePrints recommends using servlet mappings when they are available. Servlet mappings provide the most flexible way to control where to route URLs based on patterns in the URLs. Most Web application frameworks (see "Web-tier MVC frameworks," starting on page 104) use servlet mappings to direct requests to the appropriate front controller for an application.

The Java Pet Store sample application uses a servlet mapping to handle request URLs. All requests with URLs ending in `.do` are routed to the main Web-tier controller servlet, `MainServlet.java`. HTML forms in JSP pages in the sample application POST to URLs ending in `.do`, so the Web container passes all such requests to the main servlet.

Invoking model operations

Once the controller has determined which operation to perform, it needs somehow to invoke the corresponding method on the application model, including identifying and passing along data as method parameters. A naive controller design might use a large if-then-else statement, as shown in Code Example 3.7 below.

```
if (op.equals("createUser")) {
    model.createUser(request.getAttribute("user"),
                     request.getAttribute("pass"));
} else if (op.equals("changeUserInfo") {
    // ... and so on...
}
```

Code Example 3.7 A poorly-designed controller

The "if-then-else" approach leads to a very large service method, difficult to read, and more difficult to maintain. A better approach is to define an abstract class `Action`, and delegate requests to concrete `Action` subclasses. Sample code for the abstract class `Action`, and a concrete class `CreateUserAction`, appear in X.

```
// Action.java:
public abstract class Action {
    protected Model model;
    public Action(Model model) { this.model = model; }
    public abstract String getName();
    public abstract Object perform(HttpServletRequest req);
};

// CreateUserAction.java:
public class CreateUserAction extends Action {
    public CreateUserAction(Model model) {
        super(model); // you gotta work...
    }
    public String getName() { return "createUser"; }
    public Object perform(HttpServletRequest req) {
        return model.createUser(req.getAttribute("user"),
                                req.getAttribute("pass"));
    }
}
```

Code Example 3.8 An abstract Action class and a concrete subclass

Code Example 3.8 defines an abstract class `Action` that defines a correspondence between the name of an operation, received by the controller from the client, to the execution of the corresponding operation on the model, by way of the `Action`'s `perform()` method. Its concrete subclass `CreateUserAction` has a local reference to the model, on which it invokes method `createUser()`, using parameters extracted from the HTTP request.

Subclasses of Action encapsulate the process of identifying parameters and executing model methods for each model operation.

```
public class ControllerServlet extends HttpServlet {
    private HashMap actions;
```

```
    public void init() throws ServletException {
        actions = new HashMap();
        CreateUserAction cua = new CreateUserAction(model);
        actions.put(cua.getName(), cua);
        //... create and add more actions
    }
    public void doPost(HttpServletRequest req,
                       HttpServletResponse resp)
        throws IOException, ServletException {
        //... First identify operation "op" from URL...
        // then find and execute corresponding Action
        Action action = (Action)actions.get(op);
        Object result = null;
        try {
           result = action.perform(req);
        } catch (NullPointerException npx) {
            //... handle error condition: no such action
        }
        // ... Use result to determine next view (see next section)
    }
//... other methods...
}
```

Code Example 3.9 Using a map to identify and execute Actions

Code Example 3.9 shows how a controller servlet maintaining a hash map of Action objects, each indexed by its name. Method `init()`, called when the servlet loads, fills the hash map with objects that can execute operations on the model, indexed by the name used to invoke them. When a request is received, the servlet's service method identifies the name of the operation to perform, looks up the operation, and executes it. The Action returns a "result" object that the servlet can then use, along with other data, to decide which view to display next.

The code samples shown above are greatly simplified for clarity and compactness. Some important issues, such as giving actions access to the `ServletContext`, were deliberately omitted. The Java Pet Store provides a full, working example of this sort of controller in the file `MainServlet.java`. The purpose of the code shown here is to demonstrate how to provide an extensible framework for dispatching operation requests from the client.

One way to improve the extensibility of the servlet code in Code Example 3.9 would be to initialize the `actions` hash map from an external configuration file. An external XML file, for example, could contain pairs of operation names and corresponding `Action` class names. The servlet could initialize the action map with the names and action classes referenced in the XML file. The XML file would be deployed as a resource in the Web application archive. Adding a new `Action` would then be as simple as writing the `Action` subclass, adding the new Action subclass to the application archive, and adding an operation name/action class pair to the configuration file. The controller servlet would then be able to dispatch calls to actions that did not even exist when the controller was written. The Java Pet Store request processor works in just this way.

So far, you've seen how a Web-tier controller can identify and dispatch model operations. But that's only half of the controller's job. It's also responsible for determining the next view to display.

Controlling screen flow

From the user's point of view, a Web application looks like a sequence of views whose navigation is affected by the functions being executed. Determining the screen flow (i.e., the succession of views) in an application is the second major duty of a controller.

In this section, the term "view" corresponds to a Web resource with a URL from which Web content is available. A view might be a JSP page, a servlet, or static content, or some combination of the three assembled into a page that is served to a Web client. Each view has a symbolic *name*, corresponding to a set of templating parameters that configure the page; see "Templating" on page 119.

Typically, the "next" view to display depends on one or more of:

- the current view, indicated by the request URL
- the contents of the request, in `HttpServletRequest`
- the results of any operation on the application model, returned by model method invocations
- possibly other server-side state, kept in `PageContext`, `ServletRequest`, `HttpSession`, and `ServletContext`.

For example, next view to display after a login view very likely depends on:

- the current view (the "login" view),
- the user id and password contained in the request,
- the success or failure of the login operation, and
- possibly other server-side state. Examples of such state might include a maximum number of allowed users (application scope), or the URL the user was trying to access when the need for a login operation was detected (session scope). See "Web-tier state maintenance," starting on page 126 for a description of state and its scope.

The controller uses these data to determine which view to display next. A Web controller "displays a view" by writing the view to the client in a format the client understands; for example, returning HTML to a browser. The controller could directly produce the content in its service method, but such a method would quickly grow to an unmanageable size, and would be difficult to extend.

Instead of a controller creating the next view directly, a better approach is for the controller to delegate content creation to other Web-tier components; namely, servlets and JSP pages. A Web controller can use request forwarding to activate a servlet or JSP page that produces the next view. (See "Request forwarding and inclusion," starting on page 85.) This strategy greatly simplifies the controller, since all it needs to do is determine the URL for the next view and forward the request to that URL. The component at that URL will be responsible for generating the (possibly dynamic) content representing the next view.

Since determining which view to display next is a complex process, involving data from many sources, it's best to encapsulate the process of choosing the view in a separate class that manages screen flow. The controller servlet then delegates responsibility for determining the "next" view, based on the criteria listed above, to the screen flow manager.

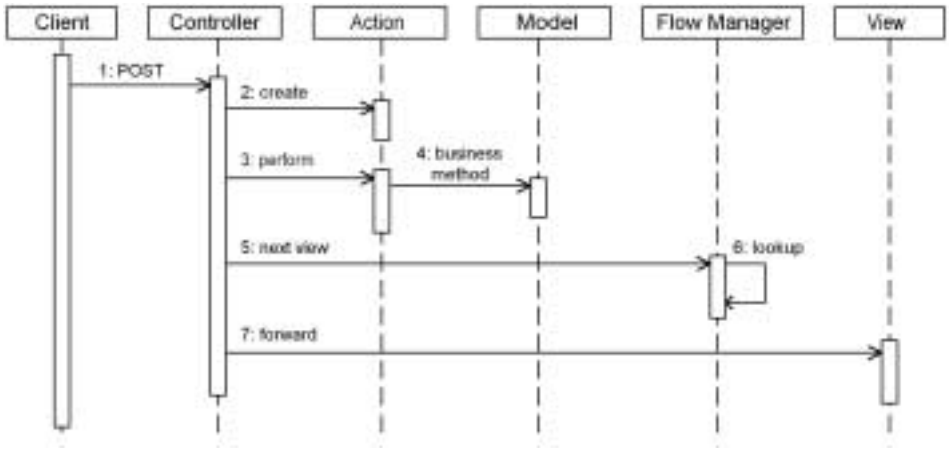

Figure 3.6 Web-tier controller OID

Figure 3.6 is an object interaction diagram that shows the Web-tier controller interacting with other Web-tier classes. The controller [1] receives a POST from the client, and [2] creates and [3] performs a corresponding `Action`, which results in [4] business method calls on the model, as described in the previous section. The controller then [5] calls the flow manager, which [6] selects the next view to display (usually a JSP page or a servlet), and [7] forwards the request to the selected view. The forwarding renders and transmits the selected view back to the client.

The "lookup" (step 6) in Figure 3.6 is where the screen flow manager does most of its work. Usually there is a one-to-one correspondence between the requested URL and a particular view to display. For example, the URL `/login` might correspond directly to the login view; the URL `/catalog` maps to the catalog view, and so on.

Since most URLs map to a single view, screen flow can be controlled declaratively by defining an externally-defined dictionary of URLs to view names. This dictionary, called a *screen flow map*, maps a particular request URL (like `/login`) to a specific view name (like "LOGIN"). The screen flow manager then uses the resulting view name to look up the corresponding view. Referencing views by name in this way allows more than one URL to map to a particular view.

In some cases, a particular request URL might depend on more than just the request URL: it might also depend on the results of an operation, or on server-side

state, as described above. For example, the next view after a “login” view very likely depends on whether the login succeeded.

In situations where the next view to display depends on server-side state as well as on the request URL, the screen flow map can define a *flow handler* class that programmatically determines the next view to display after a particular request URL. An application developer can write a screen flow handler object that examines the request (and other server-side state) to programmatically select the next view.

Example

The Java Pet Store uses a screen flow map to match URLs to their corresponding subsequent screens. Code Example 3.10 shows two URL mappings, and Figure 3.7 shows two HTTP requests to the URLs for those mappings. (The step numbers in the following text refer to Figure 3.7.)

```
<url-mapping url="/help" screen="HELP"/>

<url-mapping url="/verifysignin" screen="SIGN_IN_SUCCESS">
   <request-handler class="com.sun.j2ee.BluePrints.petstore.con-
trol.web.handlers.SigninHandler"/>
   <flow-handler class="com.sun.j2ee.BluePrints.petstore.con-
trol.web.handlers.SigninFlowHandler">
      <handler-result result="2" screen="SIGN_IN_ERROR"/>
      <handler-result result="1" screen="SIGN_IN_SUCCESS"/>
   </flow-handler>
</url-mapping>
```

Code Example 3.10 Excerpt from the Java Pet Store screen flow map

The first example in Code Example 3.10 simply maps the URL `/help` to the view named “HELP”. When the controller servlet [1] receives a request for the URL `/help`, it [2] sends the request to the screen flow manager, which [3] looks up the URL mapping and finds that the next view is called “HELP”. Since this URL mapping doesn’t define a request handler, no action is performed. The screen flow manager simply returns the name “HELP” to the main servlet, which

[4] forwards the request to the view called HELP. The user then receives the HELP view from the server.

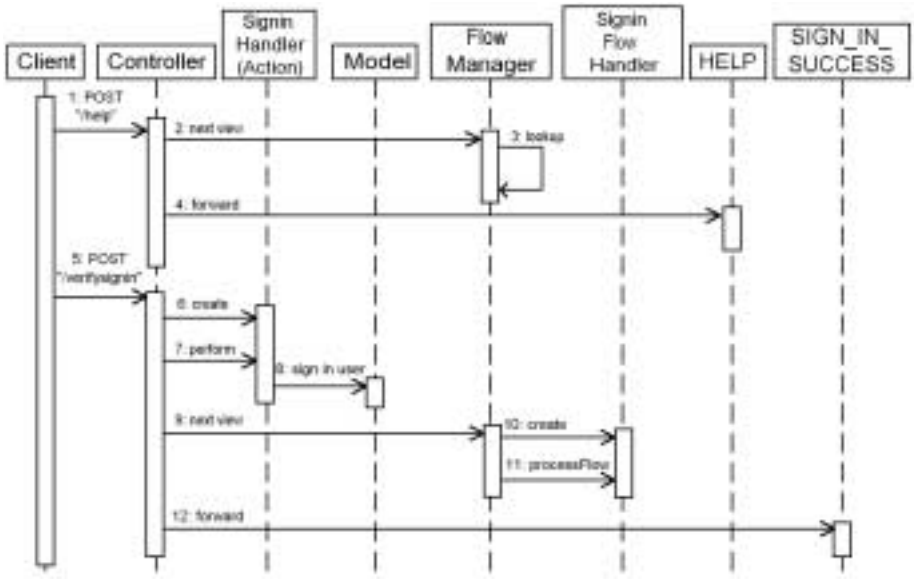

Figure 3.7 OID for two HTTP POSTs to the URLs mapped in Code Example 3.10

The second example in Code Example 3.10 is more complex. The URL mapping for `/verifysignin` has a request handler class implementing the action to perform, and a flow handler class that chooses the next view to display. When the controller [5] receives the URL `/verifysignin`, it [6] creates an instance of action class `SigninHandler` and passes the request to that object. The `SigninHandler` [7] performs the application operation: [8] it tries to sign the user into the application, using the user name and password encoded in the request parameters. The controller then [9] asks the screen flow manager for the next view. The screen flow manager [10] creates a `SigninFlowHandler` and [11] passes the request to it. The `SigninFlowHandler` looks at the state in the request and returns "1" if the user is signed in, and "2" if the user is not signed in. The `<handler-result>` tags act like a "switch" statement, indicating which view name to return depending on the return value of the flow handler. In Figure 3.7, the sign-in apparently succeeded, because [12] the controller forwards the request to the view called SIGN_IN_SUCCESS. The user receives a view indicating that login succeeded.

The last piece of the puzzle not yet explained is how to map the view names in the above design to an actual Web component (JSP page, servlet, etc.) that produces content for the user. The next section describes templates, which create named views with consistent layout.

3.4.1.7 Templating

One typical application requirement is that application views have a common layout. *Templating* is a mechanism for assembling multiple views into a single page. Each subview, such as a banner, a navigation bar, or some current "body" content, is specified as a separate component. A *template* is a presentation component that composes these subviews into a page with a specific layout. The parts of an application that share a particular template have the same layout, since the layout is controlled by the template. [+++ TBW: Reference Crupi composite view pattern. Also stress why templating is desirable.]

For example, Figure 3.8 shows the layout of a single page created by a template. Across the top of the page is a banner, on the left is a navigation menu, a footer appears at the bottom, and the body content occupies the remaining space.

Figure 3.8 A template composes other views into a consistent layout

Using templates in an application design centralizes control of the overall layout of pages in the application, easing maintenance. If you change the layout in the template file, the page layout changes for the entire application. More importantly, the individual subviews (like the "Navigation Menu" in Figure 3.8) are used referentially, instead of by copy-and-paste; therefore, changing a subview means changing a single source file, instead of changing all the files in which that subview occurs.

Implementation of templates is most easily explained by example. In the Java Pet Store, a JSP page called a *template file* controls the layout of the page. The template file is a standard JSP page that uses custom tags to include content from other JSP pages into each segment of the overall page. Which JSP page to include is indicated by the requested view name, as described in the previous section.

Code Example 3.11 is an example from the Java Pet Store that produces the layout shown in Figure 3.8. This file, called `template.jsp`, is a JSP page that produces HTML. The file specifies the layout of the page as standard HTML tags, and includes the content of other JSP pages using the custom tag `<j2ee:insert>`, shown underlined in the figure.

```
<%@ taglib uri="/WEB-INF/tlds/taglib.tld" prefix="j2ee" %>
<html>
 <head>
  <title>
   <j2ee:insert parameter="HtmlTitle" />
  </title>
 </head>

 <body bgcolor="white">
  <j2ee:insert parameter="HtmlBanner" />
  <j2ee:insert parameter="HtmlTopIndex" />
  <table height="85%" width="100%" cellspacing="0" border="0">
   <tr>
   <td valign="top">
    <j2ee:insert parameter="HtmlBody" />
    </td>
   </tr>
   <tr>
   <td valign="bottom">
    <j2ee:insert parameter="HtmlPetFooter" />
   </td>
```

```
    </tr>
    <tr>
    <td valign="bottom">
     <j2ee:insert parameter="HtmlFooter" />
    </td>
    </tr>
   </table>
  </body>
</html>
```

Code Example 3.11 The template JSP page for the layout shown in Figure 3.8

The `parameter` attribute of the insert tag indicates which JSP page is to be included at the point where the tag occurs. A separate *screen definitions file* for the application provides values for these parameters, based on the name of the requested view. An excerpt from the Java Pet Store screen definitions file appears in Code Example 3.12. In this example, the screen definitions file defines that it uses template.jsp as its template file, and then defines a series of screens. Each screen has a name, and a list of values for the parameters in the template file. The template file template.jsp uses the screen definitions file to look up the values for the parameter attributes of any insert tags it contains, based on the requested view name. For example, when template.jsp is requested with the view name MAIN, any instance of `<j2ee:insert parameter="HtmlBanner">` will be replaced with the contents of "`/banner.jsp`", based on the definition of the screen called "MAIN".

```
<screen-definitions>
  <template>/template.jsp</template>
  <screen>
    <screen-name>MAIN</screen-name>
   <parameter key="HtmlTitle" value="Welcome to Java Pet Store Demo"
direct="true"/>
    <parameter key="HtmlBanner" value="/banner.jsp" direct="false"/>
     <parameter key="HtmlBody" value="/index.jsp" direct="false"/>
     <parameter key="HtmlPetFooter" value="/petfooter.jsp" di-
rect="false"/>
    <parameter key="HtmlFooter" value="/footer.jsp" direct="false"/>
```

```
    <parameter key="Annotation" value="/estore/annota-
tion/ann_overview_j2ee.jsp" direct="true"/>
  </screen>

<!-- ... more screen definitions... -->
```

Code Example 3.12 Screen definition of the MAIN view for the Java Pet Store

The Java Pet Store implements templating with a templating service, a single servlet that process all templates. Requests with URLs ending in `.screen` are dispatched to `TemplateServlet.java`, which assembles and serves templated JSP pages (see "Templating," starting on page 119). The part of the controller responsible for selecting views forwards requests to URLs ending in `.screen`, so the Web container invokes the templating servlet, which assembles the requested view from a template.

The templating mechanism described above is totally generic, as are the Web-tier controller and dispatching mechanisms described previously. These classes are not tied to the code for any particular application. As a result, the Java Pet Store architecture contains a layer of domain-neutral code called the Web Application Framework. This framework code provides the MVC controller, request dispatching, and templating features described above. The next section discusses why you should consider using such a framework in your J2EE Web-tier designs.

3.4.1.8 Example Frameworks

There are a great number of Web-tier frameworks available. Some are vendor-specific frameworks integrated with specific tool products; others are freely-available, open-source projects. Three frameworks of particular interest are:

J2EE BluePrints Web Application Framework. The Java Pet Store is written in terms of the Web Application Framework, whose architecture is described in the previous several sections. This framework offers a front controller servlet, an abstract action class for Web-tier actions, a templating service and associated custom tags, and internationalization support. The purpose of the Web Application Framework is to demonstrate both the mechanisms and effective use of a Web-tier framework layer in an application design. It is suitable for small, non-critical applications, and for learning the principles of Web-tier MVC framework design and usage.

Apache Struts. Struts is a free, open-source, Web-tier MVC framework under development at the Apache Software Foundation. Struts is highly configurable, and has a large (and growing) feature list, including a front controller, action classes and mappings, utility classes for XML, automatic population of server-side JavaBeans, web forms with validation, and some internationalization support. It also includes a set of custom tags for accessing server-side state, creating HTML, performing presentation logic, and templating. Some vendors have begun to adopt and evangelize Struts. Struts has a great deal of mindshare, and can be considered an industrial-strength framework suitable for large applications. But Struts not yet a "standard" for which J2EE product providers can interoperably and reliably create tools.

JSR 127: JavaServer Faces. A Java Community Process effort is working to create a standardized Web-tier MVC framework. Current standard Web-tier technologies offer only the means for creating general content for consumption by the client. There is currently no standard server-side GUI component or dispatching model. JavaServer Faces will be an architecture and a set of APIs for dispatching requests to Web-tier model JavaBeans; for maintaining stateful, server-side representations of reusable HTML GUI components; and for supporting internationalization, validation, multiple client types, and accessibility. Standardization of the architecture and API will allow tool interoperation and the development of reusable Web-tier GUI component libraries.

The preceding several sections have covered application controller and view design. The final section describes design alternatives for the application model.

3.4.1.9 Application model design and technology selection

Notice that the "application model" in Figure 3.2 is generic: no particular technology or tier is implied. The application model is simply the programmatic interface to the application's functionality. The design of the model API and the technology chosen for the application model are both important design considerations.

Model API design patterns

An application model, as discussed above, is the programmatic interface to an entire application's functionality. It is the application in the form of an architecture of classes and those classes' APIs. The sample code above (such as in Code Example 3.8) shows a single Model class, most application models are a system of cooperating classes.

As the number of Model classes increases, though, it can become difficult for a developer to understand how all of the parts fit together. It may not be clear to a developer from reading API documentation how the various application model objects interact, particularly if that interaction is procedural, tends to change with time, or both.

A common way to simplify a complicated application model is to provide a class, called a *Facade*, that specializes in coordinating operations between cooperating classes. The Facade design pattern is a widely-used way to provide "one-stop shopping" for application functionality. It encapsulates and hides the complexity of classes that must cooperate in specific, possibly complex ways, and isolates its callers from business object implementation changes.

As a real-world example for how a facade can be useful, consider why you might hire an accountant to do your taxes. You want to provide the accountant with a few income forms, maybe a stack of receipts, and some basic information about yourself, and the accountant does your taxes for you. You certainly don't want to have to bother with all of the forms and procedures that a complex, bureaucratic tax code inflicts on you. You don't want to have to know about procedural changes, unless those changes imply new information (or fiscal) requirements. You just want to give your paperwork to the accountant and say, "Do my taxes," sign the form, and write a check (or, better yet, receive one).

Your accountant is your "facade" to the tax code. The accountant handles all the back-end complexity of taxes, provides you with a simplified interface, and gives you back only the information you need: How much you owe?

A facade unifies all application functions into a single interface. The Java Pet Store Web tier has the `WebClientController`, which receives application model requests from clients (usually from Web Application Framework `HTMLAction` subclasses) and implements them with application model classes.

One potential problem with a facade is that for large applications, the facade's API can become unreasonably large. Furthermore, extending a facade with a new function requires editing the facade class source code, recompiling, and redeploying.

The *Command* pattern solves the problem of an exploding class API by encapsulating each application function in a separate class. An instance of such a class represents a single request for an application service, plus any data necessary to perform the service. The application model can have a single method that executes each command object and processes results or errors.

For example, in a health insurance application, an EnrollPatientCommand might contain a Patient object, a Plan object, and an effective date. Instead of calling an

`enrollPatient()` method on the model, the developer would create a `Command` object and send it to the model's command executing method for execution. The model would then either execute the command or throw an exception.

The Java Pet Store class `WebClientController.java` is a model facade that uses a command pattern to keep down the size of the API. Every Web-tier application model function for the Java Pet Store is executed by passing a Command object to the Web Client Controller, which interprets the object and returns the result.

Using the Facade and Command design patterns together gives an application model extensibility and ease of use, without creating classes with an unmanageable number of methods.

Model technology selection

One of the most important decisions in a J2EE enterprise application is the selection of the technology for implementing the application model. There are essentially two choices: Web-tier Java objects, or EJB tier enterprise beans.

Web-tier Java objects provide high performance, because calls to them are local to a JVM instance. All Java developers understand how to create Java objects, while enterprise beans require special training. And Java objects can be easier to program, because they don't require all of the complexity of enterprise beans. But Java objects can also be *more* difficult to program than enterprise beans, because the services provided by the enterprise bean container must be written and managed manually. By themselves, Java objects are not persistent, secure, or transactional; they don't provide load balancing or failover; threading must be carefully synchronized; and they have no automatic resource management other than garbage collection. Still, for some applications, a model composed of Java objects, possibly managing their own persistence via JDBC, is an appropriate solution.

Enterprise beans in the EJB tier provide remote access to enterprise data and application logic. They can be transparently persistent, are declaratively secure and transactional, commonly offer load balancing and failover for scalability and availability, handle multithreaded access automatically, and provide automatic resource management. But enterprise beans technology is detailed to learn and complex to implement (though many of the complexities can be overcome with tools). Remote enterprise beans introduce method call latency and the need to handle remote exceptions. Creation of enterprise bean instances is straightforward, but is not as simple as with Java objects. Enterprise beans also require deployment descriptors, which require some knowledge to create. Enterprise

beans are a powerful, scalable, industrial-strength technology that require a non-trivial investment in technology and training.

The decision of whether to use enterprise beans in your application depends on your application requirements, your long-term goals, and your staffing and budget constraints.

3.4.1.10 Web-tier state maintenance

State maintenance decisions have an enormous impact on application performance, availability, and scalability. Such decisions include choosing the tier to manage state, selecting the appropriate scope for each item of state, and effectively tracking conversational state in a distributed environment.

State scope

The lifetime and accessibility of an item of server-side state is determined by its *scope*. An item of Web-tier state may be in application, session, request, or page scope, explained in "Web-tier state," starting on page 99. This section discusses the implications of scope in state maintenance.

The appropriate scope for an item of state depends largely on the purpose of the item in the application. It would make little sense, for example, to place a shopping cart class in application scope. Shopping cart contents are session state, so they are usually kept in session scope. But shopping cart contents maintained in Client-tier cookies would be in request scope, since they would be transmitted to the Web tier with each request. (While more scalable than server-side options, using cookies is discouraged; see "Avoid cookies," starting on page 135.)

Each state scope has implications for scalability, performance and reliability. State in *page* or *request* scope usually causes little trouble, because such data are usually not large or long-lived enough to cause resource problems. State in *application* scope is usually manageable if it is read-only. Writable application-scope state is best managed by enterprise beans, which are designed for scalable concurrent and distributed access to data and logic.

State in *session* scope has the greatest impact on application scalability and performance. Separate session-scope state accumulates for each connected user, unlike application-scope state, which is shared by users and servlets. And session-scope state exists across requests, unlike request-scope state, which is discarded when a response is served.

J2EE BluePrints recommends maintaining session state with stateful session beans in the EJB tier if enterprise beans are being used. For Web-only applica-

tions, we recommend maintaining it in the Web tier with interface `javax.servlet.http.HttpSession`. `HttpSession` frees the developer from the details of session state management, and ensures portability and scalability of Web components. The remainder of this section discusses the implications of Web-tier session state management.

How the Web container manages session state

Application servers typically track user sessions with some combination of cookies and/or URL rewriting to store a session ID on the client. The session id identifies the session, and the server is responsible for identifying the appropriate `HttpSession` object and associating it with the `HttpServletRequest`, which is then passed to a component's service method. All of the details of cookies and URL rewriting are handled by the object that implements `HttpSession`. The "Session Tracking" section of the Java Servlet Tutorial explains how to manage sessions.

Distributed servlets and conversational state

If a servlet is marked "distributable" in its deployment descriptor, the container may create multiple instances of the servlet, in multiple JVM isntances, possibly on multiple machines. Distributing a servlet improves scalability, because it allows Web request load to be spread across multiple servers. It can also improve availability by providing transparent failover between servlet instances.

But distributing multiple instances of a servlet across multiple JVM instances raises the issue of how to support conversational state. If each request a user makes might be routed to a different server, how can the system keep track of that user's session state?

J2EE product providers solve this problem in different ways. One approach, called *sticky server affinity*, associates a particular client with a particular servlet instance for the duration of the session. This solves the session state problem, since each session is "owned" by the servlet serving it. But this approach can sacrifice availability, because when a servlet, JVM instance, or server crashes, all of the associated sessions can be lost. Sticky server affinity can also make load balancing more difficult, since sessions are "stuck" on the servers where they started.

Another approach to solving the distributed conversational state problem is *state migration*. State migration serializes and moves or copies session state between servlet instances. This solution maintains the availability benefits of servlet distribution, and facilitates load balancing, because sessions can be moved from more to less-loaded servers. But state migration can increase network traffic

between clustered servers. Each time a client updates session state, all other copies of that state must be updated. If a database is used to store session state persistently (as is often the case), the database can become a performance bottleneck. The containers must also cooperate to resolve simultaneous update collisions, where two clients accessing the same session (one browser window opened from another, for example) update different copies of the same session state.

The J2EE specification leaves up to the J2EE product provider the opportunity to provide added value by solving the issue of distributed conversational state in the implementation, while maintaining the consistent J2EE interface. A good solution to this problem can be a selling point for a J2EE vendor. Designers considering a Web-tier-only architecture for high-performance applications should be sure to understand how prospective vendors address this issue.

Stateful session beans are designed specifically for handling distributed conversational state, but in the EJB tier, instead of in the Web tier; see "Manage Conversational State in Stateful Session Beans" on page 196 for rationale. Stateful session beans can manage conversational state for stateless servlets, providing scalability1. This is one of the reasons to consider using enterprise beans in an application.

For distributed Web-tier implementation guidelines, see also "Avoid using context attributes in distributable servlets," starting on page 138.

Advantages of Web-tier session state

Easy implementation. Since the application server handles the implementation of `HttpSession`, the developer is freed from bothering with the details of designing, implementing, and testing code for managing session state.

Optimization. An application server's `HttpSession` implementation is optimized and tested for that server, and therefore will probably be more efficient and reliable than a custom solution.

Potentially richer feature set. An application server's implementation of session state management may include such features as failover, cluster support, and so on, that go beyond the base-level requirements of the J2EE platform specifications. The system architect can select a server platform with the differentiating features that best suit the application requirements, while maintaining J2EE technology compatibility and portability.

Portability. The `HttpSession` interface is standardized, and so must pass the J2EE Compatibility Test Suite (CTS) across all J2EE-branded application servers. Read about the role of the CTS and J2EE branding to ensure portability at `http://java.sun.com/j2ee/compatibility.html`.

Scalability. `HttpSession` can most effectively manage storage of session state in caches and/or server clusters.

Evolvability. Application server vendors are constantly improving their offerings. Servers will maintain existing interfaces for backward compatibility, even as they add features that improve performance and reliability. An `HttpSession` implementation that works properly today will work better tomorrow as improved server versions become available, with little or no change to the source code.

Disadvantages of Web-tier session state

Limited to Web clients. The Web Tier is by definition limited to servicing Web clients (though HTML "screen scraping" or XML-over-HTTP may overcome this limitation somewhat). The `HttpSession` interface is limited to HTTP communications. Other client types will require reimplementation of session state management.

Session state not guaranteed to survive Web container crashes. Some application servers persist session state or provide failover, so sessions can span container crashes or restarts. But not all servers support that functionality, since the specification doesn't require it. As a result, restarting a container can invalidate all sessions in progress, losing all of their associated state. If this is a problem for your application, consider either selecting a server that provides session failover, or consider storing session state in the EIS tier.

3.4.2 Web-tier Design Guidelines

Design guidelines provide specific advice about how most effectively to apply J2EE technology to a design. Guidelines are advice that applies in most if not all cases.

3.4.2.1 Separate business logic from presentation

There are several reasons why presentation and business logic should be divided into separate software layers, even if they both reside together in the Web tier.

Maintainability. Most business logic occurs in more than one use case of a particular application. Copying the code for business logic from a scriptlet and pasting it into another JSP page, or into a client, expresses the same business rule in two places in the application. Any future change to that logic will require two edits instead of one. Business logic expressed in a separate component and

accessed referentially is easier to maintain. The component can be modified in one place in the source code, and the behavior changes everywhere the component is referenced. Similar arguments hold for reusing presentation logic with server-side includes, custom presentation tags, and stylesheets.

Client independence and code reuse. Intermingling data presentation and business logic ties the business logic to that particular type of client, making new client types more difficult to add. For example, business logic in JSP scriptlets cannot be used by a servlet or an application client; the code must be reimplemented for the other client types.

Isolating developer skill sets. Code that deals with data presentation, request processing, and business rules all at once is difficult to read, especially for a developer who may specialize in only one of these concerns.

For these reasons, it's good practice to create business and presentation layers in an application design. The business layer provides only application functionality, with no reference to presentation. The presentation layer presents the data and input prompts to the user (or to another system), delegating application functionality to the business layer. Business rules can then be changed in the one layer, with little or no modification to the presentation layer. The presentation or workflow of the application can change without affecting business logic code. Business logic, available referentially as simple method calls on business objects, can be used by multiple client types. And developers that specialize in either business rules or presentation can work relatively independently, concentrating on the concerns and technologies that they best understand.

3.4.2.2 Keep protocols out of model APIs

Part of separating business logic from presentation is keeping references to HTTP or other protocols out of the model APIs. A model class should be usable from outside of a Web container. Model dependencies on the Web container create unnecessary coupling, decreasing design clarity and making maintenance and unit testing more difficult. Specifically, nowhere in your application model should there be imports of packages `javax.servlet` or any of its subpackages. If a model class contains, for example, a reference to `HttpRequest`, consider why that method needs access to `HttpSession` and refactor to eliminate the dependency. Web protocols and interfaces are properly the role of the MVC controller and views.

3.5 Implementation Issues and Guidelines

This section is about implementation: coding style, preferring or avoiding certain mechanisms (and why), toolsets, libraries, and so on. Implementation concepts are about code and how it operates. For design advice and discussion, see "Web-tier Design Issues and Guidelines," starting on page 101.

3.5.1 Web-tier Implementation Issues

The following sections describe issues in Web-tier application implementation. Some clarify possible areas of confusion, such as the difference between JSP include directives and actions. Others have to do with using the technology effectively, such as appropriate use of JSP pages and servlets. Implementation issues are specific advice about technology, but their application depends on context. Deciding where and when the advice applies is up to the implementer.

3.5.1.1 JSP™ Pages Recommendations

This section addresses recommendations for effective use of JSP pages.

Use JSP pages as templates for structured textual content

JSP pages are most appropriately used for producing content that is textual and usually of consistent structure. Examples of content types commonly produced by JSP pages include HTML, XML, XHTML, DHTML, and delimiter-separated or fixed-field-width flat files. JSP pages can also be used for producing unstructured text files.

JSP pages use class `javax.servlet.jsp.JSPWriter` to write content to the response stream. This class automatically encodes any bytes written in the platform's default encoding. This is good for textual output, because it ensures text output is properly encoded. But this limitation means that JSP pages cannot be used to produce binary content directly.

JSP pages are best used for content that is partially fixed, with some elements that are filled dynamically at runtime. A JSP page contains fixed content called "template data" (not to be confused with the templating mechanism described in this chapter). Custom tags or scripting elements occurring at various points in the template data are replaced at runtime with dynamic data, producing customized content.

JSP pages may also be appropriately used for describing how to assemble textual data from other resources, such as is described in "Templating," starting on page 119.

JSP pages are usually not appropriate for creating binary content, for activating application domain logic, for creating content with highly variable structure, or for controlling request routing. Servlets are better for those situations.

Implementation guidelines for JSP pages and servlets start on page 135. For more on appropriate uses of servlets, see "Servlet Recommendations," starting on page 133.

Avoid heavy use of logic tags

Standard tag libraries usually provide some logic tags that loop, perform iterations, evaluate expressions, and make decisions. Beware using standard tag libraries to perform a great deal of logic in JSP pages. Pushing logic out of programs and into custom tags provides little benefit, and violates separation of logic and presentation. JSP pages that are programs written in XML syntax are at least as difficult to maintain as other types of programs. If you anticipate or have heavily "procedural" JSP pages, consider refactoring to push logic into new custom tags, or perform that logic in a servlet or helper class.

Use JSP include directives and actions appropriately

A JSP `<%@include@%>` directive, for example

```
<%@ include file="header.jsp" @%>
```

includes literal text "as is" in the JSP page and is not intended for use with content that changes at runtime. The include occurs only when the servlet implementing the JSP page is being built and compiled.

The `<jsp:include>` action allows you to include either static or dynamic content in the JSP page. Static pages are included just as if the `<%@include@%>` directive had been used. Dynamic included files, though, act on the given request and return results that are included in the JSP page. The include occurs each time the JSP page is served.

The JSP `<%@include@%>` directive is commonly used to modularize web pages, to reuse content, and to keep web page size manageable. Using a JSP `<%@include@%>` directive results in a larger servlet and, if abused, could lead to

code bloat. Page headers and footers are common examples of when an include directive might be correctly used.

The `<jsp:include>` action is commonly used to insert dynamically-generated content into a JSP page at the time it is served, or to select among alternatives in content for inclusion at runtime. A `<jsp:include>` action is more flexible than an `<%@include@%>` directive, since it can select the content to include at runtime. But a `<jsp:include>` action requires runtime processing and is therefore slower than an `<%@include@%>` directive.

Each time a particular JSP page is requested, all of its included pages are evaluated (and recompiled if necessary). The child page must redeclare the components it uses (for example, JavaBeans and objects provided by the container). Input data is shared between pages by way of the `HttpSession` object.

The Java Pet Store uses primarily `<jsp:include>` actions, because most of its JSP pages produce dynamic content.

One important, subtle difference between the `<%@include@%>` directive and the `<jsp:include>` action: the `<%@include@%>` directive is problematic for internationalization, because the `contentType:charset` of the included page cannot be set independently of the including page. If you need control of the page encoding, the `<jsp:include>` action is your only choice in JSP 1.1.

Implementation note: Some Web container implementations (including Tomcat) do not automatically track modifications to files included by an `<%@include@%>` directive. If you change a file that is included, you also need to force a recompile by "touching" (change the modification date of) the "parent" files that include the modified file.

3.5.1.2 Servlet Recommendations

This section describes appropriate use of servlets in an application design.

Use servlets to implement services

With the exception of generated binary content, servlets generally provide no visual functionality. Instead, think a servlet as an information service provided by an application. Non-binary display content can usually be generated by JSP pages. A servlet can perform whatever service it provides -- templating, security, personalization, MVC control -- and then select a presentation component (often a JSP page) and forward the request to that component for display. The Java Pet Store implements its MVC control and templating services as servlets. From this point of view,

servlet filters can be thought of as customizations or extensions of the services that servlets provide.

Use servlets as controllers

A Web-tier controller, which determines how to handle a request and chooses the next view to display, should be implemented as a servlet in a Model 2 Web-tier design. A controller activates application operations and makes decisions, which are essentially procedural tasks best suited for program code in servlets.

Controllers should not be implemented as JSP pages. Since JSP pages that are mostly logic are a mixture of markup tags and program code, they are difficult to read and maintain, especially for Web developers who are not also programmers.

```
public class CreditCardController extends HttpServlet {
   protected void doPost(HttpServletRequest req,
                         HttpServletResponse res) throws... {

      String creditCard = req.getParameter("creditCard");
      String jspPage = "/process" + creditCard + ".jsp";
      ServletContext sc = getServletContext();
      RequestDispatcher rd = getRequestDispatcher(jspPage);
      rd.forward(req, res);
   }
}
```

Code Example 3.13 A servlet properly used as a controller

An example of a servlet being used properly as a controller appears in Code Example 3.13 above; the same controller implemented improperly as a JSP page appears in Code Example 3.17 on page 140. Comparing the two, it should be clear that the controller shown above is a cleaner implementation.

See also “Avoid request forwarding from JSP pages” on page 140.

Use servlets to generate binary content

Binary content should be generated by servlets; in fact, JSP pages can’t create binary content (see page 131 for why). Servlets can output binary content by setting the `Content-Type` HTTP header to the MIME type of the content being generated,

and then writing the binary data to an OutputStream acquired from the ServletRequest, as shown in the code fragment below:

```
public class BinaryContentServlet extends HttpServlet {
    public void service(HttpServletRequest req,
                        HttpServletResponse rsp) throws...
        rsp.setHeader("Content-type", "image/jpg");
        OutputStream os = rsp.getOutputStream();
        // ... now write binary data to the OutputStream
```

Code Example 3.14 The beginning of a servlet that produces binary content

A servlet can request and write to either an `OutputStream` or a `PrintWriter`, but not both.

3.5.2 Web-tier Implementation Guidelines

J2EE BluePrints offers the following guidelines that we believe will hold true for most Web-tier implementations.

3.5.2.1 Avoid cookies

J2EE BluePrints recommends against using cookies for storing session state. One of the major goals of J2EE technology is to allow application developers to concentrate on implementing application functionality. An application server frees developers of messy system implementation details such as client session state mechanisms. In your J2EE application, it's usually better to allow the application server to handle the details of where and how session state is stored. Using either enterprise beans or a Web container's `HttpSession` implementation can provide reliable, scalable access to client-side session state through a portable interface, and saves the developer the trouble of implementing a custom solution.
Disadvantages to using cookies for session state include:

- Cookies are controlled by programming a low-level API, which is more difficult to implement than some other approaches.
- All data for a session are kept on the client. Corruption, expiration or purging of cookie files can all result in incomplete, inconsistent, or missing informa-

tion.

- Size limitations on cookies differ by browser type and version, but a least-common-denominator approach mandates a maximum cookie size of 4,096 bytes. This limitation can be eliminated by storing just references to data (session ids, user ids, etc.) in cookies, and retrieving the data as necessary from another tier (at the cost of increased server complexity and resource usage). In fact, this is equivalent to the recommended practice of storing session state in the Web tier.
- Cookies may not be available for many reasons: the user may have disabled them, the browser version may not support them, the browser may be behind a firewall that filters cookies, and so on. Servlets and JSP pages that rely exclusively on cookies for client-side session state will not operate properly for all clients. Using cookies, and then switching to an alternate client-side session state strategy in cases where cookies aren't available, complicates development and maintenance.
- Since Web clients transmit to a server only those cookies created by that server, servers with different domain names can't share cookie data. For example, JavaPetStore.com may want to allow the user to shop from its own shopping site, as well as from JavaPetFood.com. Since JavaPetFood.com can't access JavaPetStore.com's cookies, there's no easy way to unify the shopping sessions between the two servers.
- Historically, cookie implementations in both browsers and servers have tended to be buggy, and/or vary in their conformance to standards. While you may have control of your servers, buggy or nonconformant browser versions are still in use by many people.
- Browser instances share cookies, so users cannot have multiple simultaneous sessions.
- Cookie-based solutions work only for HTTP clients. This is because cookies are a feature of the HTTP protocol. Notice that while package `javax.servlet.http` supports session management (via class `HttpSession`), package `javax.servlet` has no such support.
- In a servlet, the `HttpServletResponse` and `HttpServletRequest` objects passed to method `HttpServlet.service()` can be used to create cookies on the client and use cookie information transmitted during client requests. JSPs can also use cookies, in scriptlet code or, preferably, from within custom tag code.

The next section describes how to maintain session state in the Web tier. See also the topic “Web-tier state maintenance,” starting on page 126.

3.5.2.2 Maintain Web-tier session state with HttpSession

A Web container provides session management its JSP pages and servlets by way of interface `HttpSession`. Typically, the container tries to use a cookie to save user session state on the client. If the client refuses to accept the cookie for some reason (see “Avoid cookies,” above), the container will usually switch to URL rewriting. URL rewriting works in cases where cookies will not, but suffer from other problems. Rewritten URLs tend to be long and ugly, are expensive to produce for pages with many links, and often don't "bookmark" well. Furthermore, rewritten URLs are difficult to use with legacy web pages, because the URLs in such pages are static.

It’s redundant to write your own code for managing cookies, URL rewriting, session ids, and so forth, since the container (via `HttpSession`) does all that for you. Don’t make work for yourself!

In general, it's preferable to save session state in the `HttpSession` object itself, using its methods `getAttribute()` and `setAttribute()`. Using these methods allows the Web container to maintain the state in a way most effective for your particular application and server.

See also “Web-tier state maintenance,” starting on page 126.

High-performance or highly-scalable applications may need to use enterprise beans to meet their requirements. Managing session state with stateful session beans in the EJB tier, instead of in the Web tier, provides scalability and other benefits.

3.5.2.3 Use RequestDispatcher methods forward() and include() correctly

When writing servlet chains, keep in mind that the `RequestDispatcher.forward()` requires that the body of the servlet response be empty. Writing something to the response and then calling `forward()` will cause either a runtime exception or will discard the data written. Use `RequestDispatcher.forward()` to delegate processing of an entire request to another component. Use `RequestDispatcher.include()` to build a response containing results from multiple Web resources.

3.5.2.4 Avoid using context attributes in distributable servlets

Context attributes, stored in `ServletContext`, are shared by all servlets in a Web application. But context attributes are specific to the JVM instance in which they were created. Servlets that communicate by sharing context attributes may not operate properly if distributed, because context attributes aren't replicated between Web containers in different JVM instances. To share data between distributed servlets, place the data in a session object, store it in the EIS tier in a database or distributed cache, or use an enterprise bean.

One possible exception to this guideline might be to use context attributes as a shared, data cache between the servlets in each Web container, since cache hits and misses affect only an application's performance, and not its behavior.

See section 3.3.1 of the Servlet 2.3 specification for details.

3.5.2.5 Avoid servlets that print mostly static text

Servlets composed of mostly `println()` statements would be better implemented as JSP pages. JSP pages were designed for creating textual content that combines template data with dynamic data values. Servlets that print a great deal of text, performing some logic between the print lines, are tedious to write and difficult to maintain. Every delimiter in the quoted strings written by the servlet must be properly escaped with a backslash, reducing readability. Updating the visual presentation of such a servlet requires modifying and recompiling a program, instead of updating a page of markup.

```
public class PopulateServlet extends HttpServlet {
  public void doGet(HttpServletRequest  req,
                    HttpServletResponse res) throws ... {
   ...
   if (dbConnectionClosed) {
     PrintWriter out = response.getWriter();
     out.println("<html>");
     out.println("<body bgcolor=white>");
     out.println("<font size=\"+5\" color=\"red\">Can't con-
nect</font>");
     out.println("<br>Confirm your database is running");
     out.println("</body></html>");
```

```
    }
}
```

Code Example 3.15 Bad practice: a servlet printing static content

The servlet in Code Example 3.15 on page 139 shows an example of a servlet inappropriately used to generate static content. It's difficult to read, requires careful delimiter escaping, and would probably need a programmer for nontrivial modifications to the presentation. A better option appears in Code Example 3.16, where the servlet detects an error and forwards the request to a JSP, which reports the error. This maintains proper separation of function from presentation, and allows Web developers and programmers to work with the technology each best understands.

PopulateServlet.java:

```
public class PopulateServlet extends HttpServlet {
  protected void doGet(HttpServletRequest  req,
                       HttpServletResponse res) throws ... {
    ...
   if (dbConnectionClosed) {
     ServletContext ctx = getServletContext();
     ctx.getRequestDispatcher("/db_failed.jsp").forward(req, res);
   }
}
```

db_failed.jsp:

```
<html>
 <body>
  <br><font color="red">Unable to Connect</font>
  <br>Confirm that your database is running
 </body>
<html>
```

Code Example 3.16 A servlet providing functionality, delegating display to a JSP page

3.5.2.6 Avoid request forwarding from JSP pages

When a JSP page calls `RequestDispatcher.forward()`, either directly or with a `<jsp:forward>` custom tag, it is usurping the controller role that properly belongs to a controller servlet.

```
<% String creditCard = request.getParameter("creditCard");
    if (creditCard.equals("Visa")) { %>
<jsp:forward page="/processVisa.jsp"/>
<% } else if (creditCard.equals("American Express")) { %>
<jsp:forward page="/processAmex.jsp"/>
<% } %>
```

Code Example 3.17 Bad practice: JSP page acting as a controller

Such a JSP page appears in Code Example 3.17. The scriptlets and tags in this example conditionally forward the request to another JSP page based on a request attribute. If each of the `<jsp:forward>` tags were replaced with a single line of Java code that performed the forward, the result would be a JSP page containing nothing but a scriptlet. This code would obviously be better implemented as a servlet; see Code Example 3.13 on page 134 for a servlet implementation of this functionality.

3.5.2.7 Use JavaBeans to share data between servlets and JSP pages

Remember that a JSP page is implemented as a servlet, specified as a page of markup instead of as a program. JSP pages can access servlet state using scripting expressions or custom tags, but the result of a JSP page is a servlet. So, sharing data between a servlet and a servlet *generated from a JSP page* is the same as sharing data between servlets.

Servlets can share information with other servlets by using an object that implements interface `javax.servlet.http.HttpSession`. (Note that this answer works only for HTTP sessions. `HttpSession` represents a session object only for package `javax.servlet.http`, which is the only package that defines a standardized session interface.)

An object that implements `HttpSession` can be retrieved from either the `HttpServletRequest` or `HttpServletResponse` arguments to method `HttpServlet.service()`. In servlet code, getting a session object is easy:

```
public void service(HttpServletRequest req,
                    HttpServletResponse res) ... {
   HttpSession session = req.getSession();
```

To share an object between servlets (or between invocations of the same servlet), use method `HttpSession.setAttribute()`, like this:

```
UserData userData = new userData;
userData.setName("Moliere");
session.setAttribute("userData", userData);
```

This code "binds" the `UserData` to the user session. Any servlet (or JSP) called within the same session has access to it, through method `HttpSession.getAttribute()`, for example:

```
HttpSession session = req.getSession();
UserData userData = (UserData)session.getAttribute("userData");
String userName = userData.getUsername();
```

In a JSP page, the object can be accessed with the custom tag <jsp:useBean>, lke this:

```
<jsp:useBean id="userData" type="UserData" scope="session"/>
```

The tag above either retrieves from the `HttpSession` object or creates an instance of class `UserData`, depending on whether such an object already exists in the session. It binds the instance to the session as the name `userData`. (The binding is performed behind the scenes by calling `setAttribute()` on the session.)

Elsewhere in the JSP page, you can get or set the properties of this JavaBean component:

```
<!-- get the userData property userData-->
<jsp:getProperty name="userData" property="username"/>

<!-- set all userData properties to values of
     corresponding request parameter names -->
<jsp:setProperty name="userData" property="*"/>

<!-- set userData property username to value of request
```

```
     parameter "username" -->
<jsp:setProperty name="userData" property="username"/>

<!-- set username to parameter "new_user_name" -->
<jsp:setProperty name="userData" property="username"
                 param="new_user_name"/>

<!-- set username to string "Anonymous Coward" -->
<jsp:setProperty name="userData" property="username"
                 value="Anonymous Coward"/>
```

All these examples describe how to share information between servlets in *session scope*, meaning that the information being accessed is associated with a particular user session. These techniques work similarly for application, page, and request scopes. For more on state scope, see the discussions on page 99 and page 126.

3.5.2.8 Avoid scriptlets and expressions

J2EE BluePrints recommends using custom tags instead of scriptlets and expressions in JSP pages for the following reasons:

- *Scriptlet code is not reusable.* Scriptlet code appears in exactly one place: the JSP page that defines it (except for page inclusion). If the same logic is needed elsewhere, it must be either included (decreasing readability) or copied and pasted into the new context.

  Custom tags can be reused by reference.

- *Scriptlets encourage copy/paste coding.* Since scriptlet code appears in just one place, it's often copied to a new context. When the scriptlet needs to be modified somewhere, it usually needs the same change in every place to which it was copied. Finding all copies of the scriptlet and updating them is an error-prone maintenance headache. With time, the copies tend to diverge, making it unclear exactly which scriptlets are copies of others, further frustrating maintenance.

  Custom tags centralize code in one place: change it in the tag, and the behavior changes everywhere.

- *Scriptlets mix logic with presentation.* Scriptlets are islands of program code in a sea of presentation code. Changing either requires some understanding of

what the other is doing, to avoid breaking the relationship between the two. Scriptlets can easily confuse the intent of a JSP page presentation component by expressing program logic within the presentation.

Custom tags represent program logic, freeing the JSP page to concentrate on presentation.

- *Scriptlets break developer role separation.* Since programming and Web page design are mixed when scriptlets are used, Web page designers need either to know how to program, or must know which parts of their pages to avoid modifying. Poorly-implemented scriptlets can have subtle dependencies on the surrounding template data. Consider, for example, the following line of code:

```
<% out.println("<a \"href=\"" + url + "\">" + text); %> </a>
```

  It would be very easy to change this line in a way that breaks the page, especially for someone who doesn't understand what the line is doing.

  Custom tags retain the separation of developer roles, because programmers create the tags, and page designers use them.

- *Scriptlets make JSP pages difficult to read, and therefore, to maintain.* JSP pages with scriptlets are a mixture of structured tags, JSP page delimiters, and Java code. The Java code in scriptlets often uses "implicit" objects, which are not declared anywhere (except in the JSP specification). Even consistent indentation doesn't help readability much for nontrivial pages.

  JSP pages with custom tags are composed of tags and character data, which is much easier to read. They can even be well-formed XML, and therefore validatable.

- *Scriptlet compile errors can be difficult to interpret.* Most JSP page compilers do a poor job of translating line numbers between the source page and the generated servlet. Even those that do so emit error messages that often depend on invisible context such as implicit objects or surrounding template data. With poor error reporting, a missed semicolon can cost hours of development time.

  Erroneous code in custom tags will not compile, either, but all of the context for determining the problem is present in the custom tag code, and the line numbers don't need translation.

- *Scriptlet code is difficult to test.* Unit testing of scriptlet code is virtually impossible. Since scriptlets are embedded in JSP pages, the only way to execute

them is to execute the page and test the results.

A custom tag can be unit tested, and errors can be isolated to either the tag or the page in which it is used.

Expressions are somewhat less problematic than scriplets, mostly because they tend to be small.

J2EE BluePrints recommends using custom tags to separate developer roles, contain and reuse logic and state, and improve readability, testability, and error reporting.

3.6 Summary

3.7 References and Resources