1

Building Web Applications in WebLogic

Web applications are an important part of the Java Enterprise Edition (Java EE) platform because the Web components are responsible for key client-facing presentation and business logic. A poorly designed web application will ruin the best business-tier components and services. In this chapter, we review key web application concepts and technologies and their use in WebLogic Server, and we provide a number of recommendations and best practices related to web application design and construction in WebLogic Server.

This chapter also provides the foundation for the discussion of recommended web application architectures in Chapter 2 and the construction and deployment of a complex, realistic web application in Chapters 3, 4, and 5.

Java Servlets and JSP Key Concepts

In this section we review some key concepts related to Java servlets and JavaServer Pages. If you are unfamiliar with these technologies, or if you need additional background material, you should read one of the many fine books available on the subject. Suggestions include *Head First Servlets and JSP: Passing the Sun Certified Web Component Developer Exam* by Bryan Basham et. al. (O'Reilly & Associates, 2008), *Java Servlet Programming Bible* by Suresh Rajagopalan et. al. (John Wiley & Sons, 2002), and *Java Servlet Programming* by Jason Hunter (O'Reilly & Associates, 2001).

Characteristics of Servlets

Java servlets are fundamental Java EE platform components that provide a request/response interface for both Web requests and other requests such as XML messages or file transfer functions. In this section, we review the characteristics of Java servlets as background for a comparison of servlets with JavaServer Pages (JSP) technology and the presentation of best practices later in the chapter.

Servlets Use the Request/Response Model

Java servlets are a request/response mechanism: a programming construct designed to respond to a particular request with a dynamic response generated by the servlet's specific Java implementation. Servlets may be used for many types of request/response scenarios, but they are most often employed in the creation of HyperText Transfer Protocol (HTTP) responses in a web application. In this role, servlets replace other HTTP request/response mechanisms such as Common Gateway Interface (CGI) scripts.

The simple request/response model becomes a little more complex once you add chaining and filtering capabilities to the servlet specification. Servlets may now participate in the overall request/response scenario in additional ways, either by preprocessing the request and passing it on to another servlet to create the response or by postprocessing the response before returning it to the client. Later in this chapter, we discuss servlet filtering as a mechanism for adding auditing, logging, and debugging logic to your web application.

Servlets Are Pure Java Classes

Simply stated, a Java servlet is a pure Java class that implements the `javax.servlet.Servlet` interface. The application server creates an instance of the servlet class and uses it to handle incoming requests. The `Servlet` interface defines the set of methods that should be implemented to allow the application server to manage the servlet life cycle (discussed later in this chapter) and pass requests to the servlet instance for processing. Servlets intended for use as HTTP request/response mechanisms normally extend the `javax.servlet.http.HttpServlet` class, although they may implement and use the `Servlet` interface methods if desired. The `HttpServlet` class implements the `Servlet` interface and implements the `init()`, `destroy()`, and `service()` methods in a default manner. For example, the `service()` method in `HttpServlet` interrogates the incoming `HttpServletRequest` object and forwards the request to a series of individual methods defined in the `HttpServlet` class based on the type of request. These methods include the following:

- `doGet()` for handling `GET`, conditional `GET`, and `HEAD` requests
- `doPost()` for `POST` requests
- `doPut()` for `PUT` requests
- `doDelete()` for `DELETE` requests
- `doOptions()` for `OPTIONS` requests
- `doTrace()` for `TRACE` requests

The `doGet()`, `doPost()`, `doPut()`, and `doDelete()` methods in `HttpServlet` return a `BAD_REQUEST` (400) error as their default response. Servlets that extend `HttpServlet` typically override and implement one or more of these methods to generate the desired response. The `doOptions()` and `doTrace()` methods are typically not overridden in the servlet. Their implementations in the `HttpServlet` class are designed to generate the proper response, and they are usually sufficient.

A minimal HTTP servlet capable of responding to a `GET` request requires nothing more than extending the `HttpServlet` class and implementing the `doGet()` method.

WebLogic Server provides a number of useful sample servlets showing the basic approach for creating HTTP servlets. These sample servlets are located in the `samples/server/examples/src/examples/webapp/servlets` subdirectory beneath the WebLogic Server home directory, a directory we refer to as `$WL_HOME` throughout the rest of the book.

Creating the HTML output within the servlet's `service()` or `doXXX()` method is very tedious. This deficiency was addressed in the Java EE specification by introducing a scripting technology, JavaServer Pages (JSP), discussed later in this chapter.

Servlets Must Be Registered in the Application

Servlets will only be invoked by the application server if they have been registered in the application and associated with a specific URL or URL pattern. The standard mechanism for registering a servlet involves `<servlet>` and `<servlet-mapping>` elements within the application's `web.xml` file as shown here:

```
<servlet>
  <servlet-name>SimpleServlet</servlet-name>
  <servlet-class>
    professional.weblogic.ch01.example1.SimpleServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>SimpleServlet</servlet-name>
  <url-pattern>/simple</url-pattern>
</servlet-mapping>
```

When a user accesses the specified URL, `/simple`, the application server will invoke the `doGet()`, `doPost()`, or other `doXXX()` method on the servlet class.

WebLogic Server provides an alternate annotation-based technique for registering servlets and specifying the mapped URL pattern: The `@WLServlet` annotation. The following annotation, placed at the top of the `SimpleServlet` source file, eliminates the need for `web.xml` entries for this servlet:

```
@WLServlet (
    name = "SimpleServlet",
    mapping = {"/simple"}
)
public class SimpleServlet extends HttpServlet
{
...
}
```

The `@WLServlet` annotation syntax includes all of attributes available in the `web.xml` technique, including `loadOnStartup`, `initParams`, and `runAs` values. This annotation technique represents a viable, if non-standard, approach for registering and configuring servlets in your application.

Servlets Have a Life Cycle

A servlet is an instance of the `Servlet` class and has a life cycle similar to that of any other Java object. When the servlet is first required to process a request, the application server loads the servlet

class, creates an instance of the class, initializes the instance, calls the servlet's `init()` method, and calls the `service()` method to process the request. In normal servlet operation, this same instance of the `Servlet` class will be used for all subsequent requests.

Servlets may be preloaded during WebLogic Server startup by including the `<load-on-startup>` element in the `web.xml` file for the web application or by including the `loadOnStartup` attribute in the `@WLServlet` annotation block in the servlet's class definition. You can also provide initialization parameters in the `web.xml` file using `<init-param>` elements or by including them in the `@WLServlet` annotation block. WebLogic Server will preload and call `init()` on the servlet during startup, passing the specified initialization parameters to the `init()` method in the `ServletConfig` object.

An existing servlet instance is destroyed when the application server shuts down or intends to reload the servlet class and create a new instance. The server calls the `destroy()` method on the servlet prior to removing the servlet instance and unloading the class. This allows the servlet to clean up any resources it may have opened during initialization or operation.

Servlets Allow Multiple Parallel Requests

Servlets are normally configured to allow multiple requests to be processed simultaneously by a single servlet instance. In other words, the servlet's methods must be thread-safe. You must take care to avoid using class- or instance-level variables unless access is made thread-safe through synchronization logic. Typically, all variables and objects required to process the request are created within the `service()` or `doXXX()` method itself, making them local to the specific thread and request being processed.

> **Best Practice**
>
> Servlets that allow multiple parallel requests must be thread-safe. Do not share class- or instance-level variables unless synchronization logic provides thread safety.

Servlets may be configured to disallow multiple parallel requests by defining the servlet class as implementing the `SingleThreadModel` interface:

```
...
public class TrivialSingleThreadServlet
    extends HttpServlet implements SingleThreadModel
{
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        System.out.println("Here!");
    }
...
```

This simple change informs the application server that it may not process multiple requests through the same servlet instance simultaneously. Although WebLogic Server continues to implement this mechanism for enforcing single-threaded servlets, the Servlet 2.4 specification has deprecated its

use. The specification encourages developers to protect critical sections of servlet code using synchronization logic. Of course, using synchronization logic around non-thread-safe code comes with a price — it invariably creates bottlenecks and latency in high volume systems as threads wait for their turn to execute the protected code. If the code within the critical section takes too long to execute, overall performance and scalability of your system will suffer. Avoid using the `SingleThreadModel` interface in your applications. Design your servlets be thread-safe and minimize their use of synchronization blocks.

Best Practice

Avoid using single-threaded servlets. Design your servlets be thread-safe and minimize their use of synchronization blocks to avoid potential performance issues.

Servlets May Access Request Data

The `HttpServletRequest` parameter passed in to the `service()` or `doXXX()` method contains a wealth of information available to the servlet during the processing of the request. Useful data in the `HttpServletRequest` is summarized in Table 1-1.

This is not an exhaustive list of the methods available on the `HttpServletRequest` class or its superclass, `ServletRequest`. Refer to the servlet documentation at Link 1-1 in the book's online Appendix at `http://www.wrox.com` or a good reference book on servlets for a complete list including parameter types, return types, and other details.

Servlets Use Session Tracking

A servlet is a request/response mechanism that treats each incoming request as an independent processing event with no relationship to past or future requests. In other words, the processing is stateless. The HTTP protocol is also a stateless protocol: Each request from the web browser is independent of previous or subsequent requests. Linking current requests to previous requests from the same client requires a mechanism for preserving context or state information from request to request. A number of HTML-based techniques exist for preserving context or state information:

- *Cookies* may be set in responses and passed back to the server on subsequent requests.
- *URL-rewriting* may be used to encode small amounts of context information on every hyperlink on the generated page.
- *Hidden form fields* containing context information may be included in forms.

These techniques all have limitations, and none provides the robust data types and flexibility needed to implement true state management. Fortunately, the session tracking capability defined in the Java EE servlet model provides an excellent solution.

Session tracking provides a flexible hashtable-like structure called an `HttpSession` that can be used to store any serializable Java object and make it available in subsequent requests. To identify the specific client making the request and look up its session information, session tracking uses a cookie or URL-encoded session ID passed to the server on subsequent requests. In WebLogic Server, this

session ID has the name `JSESSIONID` by default and consists of a long hash identifying the client plus creation-time and cluster information. The format of the session ID is

```
JSESSIONID=SESSION_ID!PRIMARY_JVMID_HASH!SECONDARY_JVM_HASH!CREATION_TIME
```

Table 1-1: Information Available in the HttpServletRequest

Type of Information	Access Methods
Parameters passed in the query string or through form input fields	`getParameterNames()`, `getParameter()`, `getParameterValues()`, `getQueryString()`
Server information	`getServerName()`, `getServerPort()`
Client characteristics	`getRemoteAddr()`, `getRemoteHost()`, `getAuthType()`, `getRemoteUser()`
Request information	`getContentType()`, `getContentLength()`, `getProtocol()`, `getScheme()`, `getRequestURI()`
HTTP headers	`getHeaderNames()`, `getHeader()`, `getIntHeader()`, `getDateHeader()`
Cookies sent by browser	`getCookies()`
Session information	`getSession()`, `getRequestedSessionId()`, `isRequestedSessionIdValid()`, ...

WebLogic Server uses exclamation marks to separate portions of the session ID. The first portion is used by the session tracking implementation in WebLogic Server to look up the client's `HttpSession` object in the web application context. Subsequent portions of the session ID are used to identify primary and secondary servers for this client in a WebLogic Server cluster and to track the creation time for this session. Chapter 12 discusses WebLogic Server clustering in detail as part of the discussion of administration best practices.

Using session tracking in a servlet is as simple as calling the `getSession()` method on the passed-in `HttpServletRequest` object to retrieve or create the `HttpSession` object for this client and then utilizing the `HttpSession` interface to get and set attributes in the session.

WebLogic Server supports several forms of session persistence, a mechanism for providing session failover. The two most commonly used forms are in-memory replication and JDBC persistence. When using these types of session persistence, be careful not to place very large objects in the `HttpSession`. WebLogic Server tracks changes to the session object through calls to the `setAttribute()` method. At the end of each request, the server will serialize each new or modified attribute, as determined by the arguments to any `setAttribute()` calls, and persist them accordingly.

Recognize that persisting a session attribute will result in WebLogic Server serializing the entire object graph, starting at the root object placed in the `HttpSession`. This can be a significant amount of data if the application stores large, coarse-grained objects in the session. Multiple fine-grained objects can provide superior performance, provided that your application code updates only a subset of the fine-grained objects (using `setAttribute`) in most cases. We talk more about in-memory session replication and clustering in Chapter 12.

Best Practice

Use session tracking to maintain state and contextual information between servlet requests. When using session persistence, avoid placing large objects in the session if your application tends to update only a small portion of these objects for any particular request. Instead, use multiple fine-grained objects to reduce the cost of session persistence.

To summarize, servlets are a reliable pure Java mechanism for processing HTTP requests. It can be tedious to generate the HTML response through the simple `println()` methods available on the response `Writer` object, however. As we discuss in Chapter 2, servlets are better suited for processing incoming requests and interacting with business objects and services than for the generation of HTML responses.

If servlets are a tedious way to create HTML, what is available in the Java EE specification for efficiently creating HTML responses? JavaServer Pages technology, the subject of the next section of this chapter, is specifically design to be a powerful tool for creating HTML.

Characteristics of JavaServer Pages

JavaServer Pages (JSP) technology was introduced in the Java EE platform to provide an alternative to servlets for the generation of server-side HTML content. Although a detailed discussion of JSP technology is beyond the scope of this book, some key concepts and characteristics are worth a brief review.

JSP Is a Scripting Technology

Recall that one of the important characteristics of servlets is their pure Java nature. Servlets are Java classes that are written, compiled, and debugged much like any Java class. JavaServer Pages, on the other hand, are a script-based technology similar to Microsoft's Active Server Pages (ASP) technology or Adobe's Cold Fusion scripting language. Like these scripting languages, special tags and script elements are added to a file containing HTML to produce a combination of static and dynamic content. In the case of JSP, these added elements are Java code or special JSP tags that interact with JavaBeans and other Java EE components in the application.

JSP Pages Are Converted to Servlets

The key to understanding JSP pages is to recognize that the JSP file itself is simply the input for a multistep process yielding a servlet. In the key processing step, the JSP page is parsed by the application server and converted to the equivalent pure Java servlet code. All text that is not part of JSP tags and scripting elements is assumed to be part of the HTTP response. This text is placed in output writing calls within the generated servlet method that processes requests. All Java scripting elements and tags become additional Java code in the servlet. The generated servlet is then compiled, loaded, and used to process the HTTP request in a manner identical to a normal servlet.

Figure 1-1 depicts this process for a trivial JSP page with a small amount of scripted Java code embedded on the page. The `sample.jsp` page is converted to the equivalent pure Java servlet code, compiled into a servlet class, and used to respond to the original and subsequent HTTP requests.

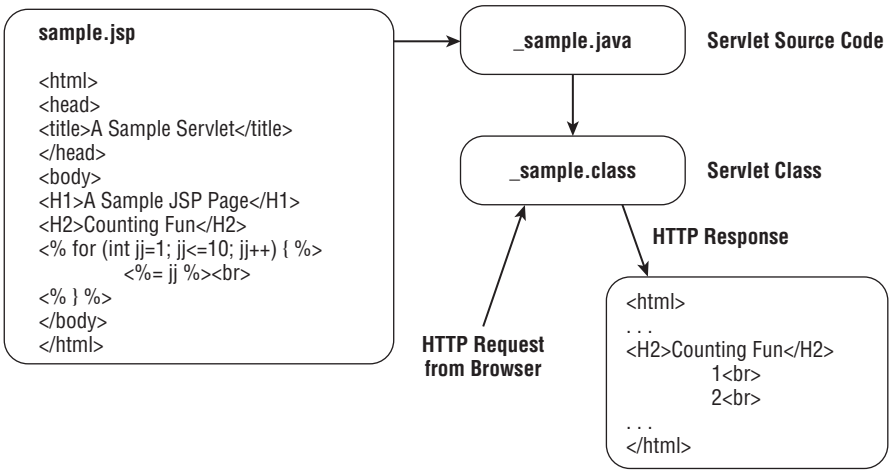

Figure 1-1: JSP page is converted to a servlet.

The parsing, conversion, compiling, and classloading steps required to accomplish this transformation are handled by the application server. You don't have to perform any of these steps ahead of time or register the resulting servlet — all of this is done automatically by the server. Note that the processing and compiling can be done prior to deployment using utilities provided by WebLogic Server, a technique known as precompiling the JSP pages.

In WebLogic Server, the resulting servlet is a subclass of `weblogic.servlet.jsp.JspBase` by default. `JspBase` is a WebLogic-provided class that extends `HttpServlet` and forwards `service()` calls to a method called `_jspService()`. You may also create a custom base class for JSP-generated servlets to replace the default `JspBase` class.

Many Tags and Scripting Elements Are Available

JSP technology provides a rich set of scripting elements and tags for creating dynamic content. Table 1-2 lists some of the important elements available.

Table 1-2: JSP Syntax Elements

Element	Syntax	Description
Scriptlet	<% *scriptlet code* %>	Java code placed directly in `_jspservice()` method at this location.
Declaration	<%! *declaration* %>	Java code placed within the generated servlet class above the `_jspservice()` method definition. This usually defines class-level methods and variables.
Expression	<%= *expression* %>	Java expression evaluated at run time and placed in the HTML output.
page	<%@ page *attribute*=''*value*'' ... %>	Controls many page-level directive attributes and behaviors. Important attributes include `import`, `buffer`, `errorPage`, and `extends`.
include	<%@ include file=''*filename*'' %>	Inserts the contents of the specific file in the JSP page and parses/compiles it.
taglib	<%@ taglib uri=''...'' prefix=''...'' %>	Defines a tag library and sets the prefix for subsequent tags.
jsp:include	<jsp:include page=''...''/>	Includes the response from a separate page in the output of this page.
jsp:forward	<jsp:forward page=''...''/>	Abandons the current response and passes the request to a new page for processing.
jsp:useBean	<jsp:useBean id=''...'' class=''...'' scope=''...''/>	Declares the existence of a bean with the given class, scope, and instance name.

Many more elements and tags are available. A detailed discussion of these elements is beyond the scope of this book. Consult one of the books listed at the beginning of this chapter for a complete list of JSP elements and tags, or browse Sun's JSP area at Link 1-2 for more information.

All Servlet Capabilities Are Available

Because JSP pages are converted to servlets, all of the capabilities and techniques available in servlets are also available in JSP pages. The `HttpServletRequest` and `HttpServletResponse` parameters are available, along with a number of predefined variables available in the JSP page, as listed in Table 1-3.

JSP scriptlet code may access these implicit objects directly because all scriptlet code is placed in the generated `_jspService()` method code below the definition of these objects. Nevertheless, the

direct use of implicit objects in JSP scriptlet code is considered poor form. The JavaServer Pages Standard Tag Library (JSTL), discussed later in this chapter, provides access to data stored within these implicit objects — and many others — in a much safer and more standard way.

Table 1-3: JSP Implicit Objects

Object	Type	Description
request	`javax.servlet.http.HttpServletRequest`	Provides access to request information and attributes set at the request scope.
response	`javax.servlet.http.HttpServletResponse`	Reference to the response object being prepared for return to the client.
pageContext	`javax.servlet.jsp.PageContext`	Provides access to attributes set at the page scope.
session	`javax.servlet.http.HttpSession`	Session object for this client; provides access to attributes set at the session scope.
application	`javax.servlet.ServletContext`	Application context; provides access to attributes set at the application scope.
out	`javax.servlet.jsp.JspWriter`	PrintWriter object used to place text output in the HTTP response.
config	`javax.servlet.ServletConfig`	Reference to the servlet configuration object set during initialization; provides access to initialization parameters.

Session tracking is available by default in JSP pages. If your application is not using session tracking, you should disable it to avoid unnecessary session persistence. Although there is no explicit way to disable session tracking for the entire web application, servlets will not create sessions unless the servlet code calls the `getSession()` method. JSP pages may disable sessions using the `page` directive:

```
<%@ page session="false" %>
```

Even if your JSP does nothing with the session information, WebLogic Server must persist the last access time for the session at the end of the request processing. It is best to disable session tracking explicitly in JSP pages that do not use it.

Best Practice

Disable session tracking in JSP pages that do not require this feature to avoid unnecessary session persistence.

Like servlets, JSP pages are normally multithreaded and may process multiple requests simultaneously. The same thread-safety restrictions that apply to servlets also apply to JSP pages unless the JSP is configured to be single threaded. In a JSP page a special `page` directive is used to configure this attribute:

```
<%@ page isThreadSafe="false" %>
```

If the `isThreadSafe` attribute is set to `false`, the resulting servlet will implement the `SingleThreadModel` interface. This technique, like the related servlet technique, is deprecated in the Servlet 2.4 specification and should be avoided.

Best Practice

Avoid declaring JSP pages to be single threaded. Code that is not thread-safe should be encapsulated in some other Java class and controlled using synchronization blocks.

JSP Response Is Buffered

As we said, servlets and JSP pages are request/response mechanisms: An HTTP request is made by the browser, and an HTML response is generated by the servlet or JSP page. In both cases, this response is normally *buffered*, or held in memory on the server temporarily, and sent back to the calling browser at the end of the processing.

By default, output created in the generated servlet code is buffered, along with HTTP headers, cookies, and status codes set by the page. Buffering provides you with these important benefits:

- Buffered content may be discarded completely and replaced with new content. The `jsp:forward` element relies on this capability to discard the current response and forward the HTTP request to a new page for processing. Note that the `errorPage` directive uses `jsp:forward` to send the processing to the error page if an error is caught in the JSP page, so buffering is also required for proper error page handling.
- Buffering allows the page to add or change HTTP headers, cookies, and status codes after the page has begun placing HTML content in the response. Without buffering, it would be impossible to add a cookie in the body of the JSP page or change the response to be a redirect (302) to a different page once output is written because the headers and cookies have already been sent.

When the buffer fills, the response is committed, and the first chunk of information is sent to the browser. Once this commit occurs, the server will no longer honor `jsp:forward`, HTTP header changes (such as redirects), or additional cookies. The server will generate an `IllegalStateException` if any of these operations is attempted after the buffer fills and the response is committed.

The default size of the JSP output buffer is 8KB in WebLogic Server, which you can control using the `page` directive in each JSP page:

```
<%@ page buffer="32kb" %>
```

Output buffering may also be turned off using this directive by specifying `none` for a size, but this practice is not recommended.

Output buffers should be set to at least 32KB in most applications to avoid filling the buffer and committing the response before the page is complete. The minor additional memory requirement (32KB times the number of threads) is a small price to pay for correct error page handling and the ability to add cookies and response headers at any point in large pages.

Best Practice

Always use output buffering in JSP pages. Increase the size of the buffer to at least 32KB to avoid redirect, cookie, `jsp:forward`, and error page problems.

JSP Pages Have Unique Capabilities

Unique capabilities are available in JSP pages that are not present in servlets. The most important of these is the ability to embed custom XML tags within the JSP page.

Custom tags provide a mechanism to interact with a custom developed Java class that encapsulates business logic, presentation logic, or both. Custom tag elements are placed in the JSP page by the developer and then parsed and preprocessed by the application server during the conversion from JSP to servlet. The tag elements are converted by the server to the Java code required to interact with the tag class and perform the desired function. Later in this chapter we discuss custom tags and commonly used tag libraries in more detail and present best practices for their use in WebLogic Server.

To summarize, JavaServer Pages technology is a scripting language used to create HTML responses. JSP pages are converted to pure Java servlets by the application server during processing, and they can perform nearly any task a pure Java servlet can perform. JSP pages also have unique directives, features, and customization capabilities unavailable to servlets.

Why not use JSP for everything and forget servlets completely? Although it is possible to do so, servlets often provide a better mechanism for implementing presentation-tier business logic. Chapter 2 addresses this issue in detail and provides guidance for the proper use of each technology.

Web Application Best Practices

Now that you have reviewed some of the key concepts related to web applications in WebLogic Server, it's time to dig in and discuss best practices. So many options are available to designers and developers of Java EE web applications that it would require an entire book to list and explain all of the web application best practices we could conceivably discuss. In this section, we've attempted to discuss the best practices we feel are applicable to the widest variety of development efforts or are most likely to improve the quality or performance of your WebLogic Server web applications.

The best practices contained in this chapter cover everything from recommended techniques for using custom tags to proper packaging of your web application to caching page content for performance. They are presented in no particular order of importance, because the importance of a given best practice depends greatly on the particular application you are building.

Ensure Proper Error Handling

Unhandled exceptions that occur during the execution of a servlet or JSP-generated servlet cause the processing of that page to stop. Assuming the response has not been committed, the JSP output buffer will be cleared and a new response generated and returned to the client. By default, this error response contains very little useful information apart from the numeric error code.

What you need is a friendly, informative error page containing as much information as possible to help during debugging. Fortunately, there is a built-in mechanism for specifying a custom error page for use in handling server errors during processing.

First, you construct an error page to present the error information to the user in a friendly fashion. At a minimum, it should display the exception information and a stack trace. To be more useful during debugging, it can display all request and HTTP header information present using the methods available on the `HttpServletRequest` object. Portions of an example error page are shown in Listing 1-1. The entire page is available on the companion web site located at `http://www.wrox.com/`.

Listing 1-1: ErrorPage.jsp.

```
<%@ page isErrorPage="true" %>
<html>
<head><title>Error During Processing</title></head>
<body>
<h2>An error has occurred during the processing of your request.</h2>
<hr>
<h3><%= exception %></h3>
<pre>
<%
    ByteArrayOutputStream ostr = new ByteArrayOutputStream();
    exception.printStackTrace(new PrintStream(ostr));
    out.print(ostr);
%>
</pre>
```

Continued

Listing 1-1: ErrorPage.jsp. *(continued)*

```
<hr>
<h3>Requested URL</h3>
<pre>
<%= HttpUtils.getRequestURL(request) %>
</pre>

<h3>Request Parameters</h3>
<pre>
<%
Enumeration params = request.getParameterNames();
while(params.hasMoreElements()){
    String key = (String)params.nextElement();
    String[] paramValues = request.getParameterValues(key);
    for(int i = 0; i < paramValues.length; i++) {
        out.println(key + " : "  + paramValues[i]);
    }
}
%>
</pre>

<h3>Request Attributes</h3>
<pre>
...
</pre>

<h3>Request Information</h3>
<pre>
...
</pre>

<h3>Request Headers</h3>
<pre>
...
</pre>
```

Second, place a `<%@ page errorPage=" ... " %>` directive on all JSP pages in the application specifying the location of this error JSP page. Listing 1-2 presents a simple example JSP page that declares the error page explicitly. Normally, you would do this through a common include file shared by all pages rather than including the directive on every page.

Listing 1-2: ErrorCreator.jsp.

```
<%@ page errorPage="ErrorPage.jsp" %>
<html>
<head></head>
<body>
<!-- Do something sure to cause problems -->
<% String s = null; %>
The string length is: <%= s.length() %><p>
</body>
</html>
```

Accessing the `ErrorCreator.jsp` page from a browser now causes a useful error message to be displayed to the user. The page could conform to the look and feel of the site itself and could easily include links to retry the failed operation, send an email to someone, or go back to the previous page.

As an alternative to specifying the `errorPage` on each individual JSP page, a default error-handling page may be specified for the entire web application using the `<error-page>` element in `web.xml`:

```
<error-page>
    <error-code>500</error-code>
    <location>/ErrorPage.jsp</location>
</error-page>
```

These two mechanisms for specifying the error page may look very similar but are, in fact, implemented quite differently by WebLogic Server. The `<%@ page errorPage="... " %>` directive modifies the generated servlet code by placing all JSP scriptlet code, output statements, and other servlet code in a large try/catch block. Specifying the error page in `web.xml` does not affect the generated servlet code in any way. Instead, uncaught exceptions that escape the `_jspService()` method in the original page are caught by the web container and forwarded to the specified error page automatically.

Which technique is best? Unless the target error page must differ based on the page encountering the error, we recommend the `<error-page>` element in `web.xml` for the following reasons:

- A declarative and global technique has implicit benefits over per-page techniques. Individual pages that require different error pages can easily override the value in `web.xml` by including the `page` directive.
- The information describing the original page request is more complete if the `<error-page>` element is used rather than the `page` directive. Specifically, calling `request.getRequestURL()` in the error page returns the URL of the original page rather than the URL of the error page, and additional attributes are placed on the request that are not present if the `page` directive is employed.

Best Practice

Create a friendly and useful error page, and make it the default error page for all server errors using the `<error-page>` element in `web.xml`. Override this default error page using the `page` directive in specific pages, if necessary.

Use JSTL Tags to Reduce Scriptlet Code

The JavaServer Pages Standard Tag Library (JSTL) is a custom tag library that encapsulates many core functions required within JSP pages and virtually eliminates the need for JSP scriptlet code. Common constructs, such as conditionals, loops, accessing request or session data, placing data in the response HTML output, formatting output, displaying language-sensitive strings, and many other functions, are implemented in the JSTL library in a standard way. JSTL represents a huge improvement over the old `jsp:useBean` and `jsp:getProperty` techniques.

Custom tags can be difficult to create, but no knowledge of their construction is required to use them successfully. All you need is a good reference on the tag library you are trying to use and a basic understanding of the syntax for calling custom tags within your JSP pages.

Calling Custom Tags in JSP Pages

Custom tags are invoked within JSP pages by embedding the appropriate XML tags in your page using the syntax

```
<prefix:tagname attribute1="value1" ...  attributeN="valueN" />
```

or

```
<prefix:tagname attribute1="value1" ...  attributeN="valueN" >
  ...
</prefix:tagname>
```

The prefix represents the short name you gave a particular library when it was declared in your page, the tagname is the specific tag or function identifier within the library, and the attribute/value pairs are the data or settings needed by the tag for proper operation.

For example, if you've declared the JSTL `core` library using a prefix `c` with a `taglib` directive like this:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

you would invoke the `out` tag within the JSTL core library to display the contents of the request parameter `employeeNum` using the following syntax:

```
<c:out value="${requestScope.employeeNum}"/>
```

The equivalent scriptlet code, for comparison purposes, might look like this:

```
<%= request.getParameter("employeeNum") %>
```

Although this JSTL example does not appear to be significantly shorter or simpler than using scriptlet code in this trivial example, the difference becomes much larger with more complex operations.

Using Expression Language in JSTL Calls

The JSTL libraries support the use of Expression Language (EL) within many of the attribute/value pairs supplied to the custom tags. To understand the difference between an invocation with and without EL support, consider the following two calls to the `out` tag in the `core` JSTL library:

```
<c:out value="requestScope.employeeNum" />

<c:out value="${requestScope.employeeNum}" />
```

The first call passes a simple string to the tag implementation via the `value` attribute. This string is simply placed in the output response being generated by the JSP, and the users would see the string `requestScope.employeeNum` on their web page.

The second call informs the custom tag implementation that it is using the EL syntax by including `${ ... }` characters within the `value` attribute passed to the tag. The tag sees this syntax and treats the

string within the braces as an expression that should be parsed and treated as a request for data from some source available to the page.

By specifying `requestScope` in the expression, we are indicating that the tag should look only in the `HttpServletRequest` object, and by specifying `employeeNum` we are telling the tag which parameter or attribute we want from the request object. The tag will find the object located in the request under this key, invoke `toString()` on it, and place the result in the output. The users will, hopefully, see a valid employee number on their web page.

As a second example, consider the following tag invocation:

```
<c:out value="${employee.myAddress.line1}" />
```

This example does not specify a source scope (`requestScope, sessionScope,` and so on), and has three parts. The tag implementation will perform the following steps as it parses and processes this expression:

- It will first search within all of the scopes available to it (starting with `pageScope` and ending with `applicationScope`) for some attribute stored using `employee` as the key. Let's assume it finds an object of the class `CompanyEmployee` located in the `HttpSession` stored with this key.
- The tag will then examine the retrieved `CompanyEmployee` object and attempt to invoke a `get` method based on the next identifier in the expression. In this case the method attempted would be `getMyAddress()`. Assuming such a method exists, the underlying `Address` object is extracted and the processing continues.
- The tag will next attempt to invoke `getLine1()` on the `Address` object and extract the resulting object, most likely a simple `String` object in our example.
- Finally, the tag will invoke `toString()` on the object returned by `getLine1()` to obtain the text that should be placed in the HTML response output.

This *chaining* of identifiers using the dot operator is very common in JSTL attribute values. It allows access to specific nested properties within objects stored on the request or session with a simple, compact syntax.

Entries in a `List` or `Map` object can be accessed with the same dot operator or through the use of an alternate bracket syntax as shown in these two equivalent tag invocations:

```
<c:out value="${sessionScope.stateNames.NY}" />

<c:out value="${sessionScope.stateNames["NY"]}" />
```

Hard-coding the map's key into the JSP page is clearly of limited value, and it too can be replaced by an expression that returns the key to be used in the `Map` lookup. The following example replaces the fixed value of `NY` with the value passed in through a request parameter called `stateCode`:

```
<c:out value="${sessionScope.stateNames[param.stateCode]}" />
```

In this example we look in the session for the `Map`, and look in the passed-in request parameters for the key. These are two examples of implicit objects that are available within the Expression Language. Table 1-4 presents a complete set of these implicit objects.

Table 1-4: JSP Implicit Objects

Identifier	Description
`pageScope`	`Map` containing page-scoped attributes
`requestScope`	`Map` containing request-scoped attributes
`sessionScope`	`Map` containing session-scoped attributes
`applicationScope`	`Map` containing application-scoped attributes
`param`	`Map` containing the primary values of the request parameters
`paramValues`	`Map` containing all values of the request parameters as `String` arrays
`header`	`Map` containing the primary values of the request headers
`headerValues`	`Map` containing all values of the request headers as `String` arrays
`cookie`	`Map` containing all cookies accompanying the request
`initParam`	`Map` containing the context initialization parameters of the web application
`pageContext`	The `PageContext` instance for the page, providing access to all JSP implicit objects

Although the Expression Language syntax is very powerful, it can be a bit confusing at times. Find a good reference on JSTL, including the EL, and refer to it often until you get the hang of it. You'll find that 95% or more of your custom tag invocations will use the EL syntax for one or more of the attributes in the tag, because it is the only recommended way to pass dynamic data into the tags.

Above all, resist the temptation to use the old-style scriptlet code in your JSP pages. With a little experimentation, you'll find there is very little you can do in scriptlet code that you cannot accomplish with JSTL and the Expression Language syntax.

Best Practice

Master the Expression Language (EL) syntax and use it extensively in the JSTL tags within your JSP pages. Use expressions in a clear and consistent manner to improve readability, choosing appropriate operators and constructs. Avoid JSP scriptlet code if at all possible.

JSTL Contains Five Tag Libraries

The JavaServer Pages Standard Tag Library (JSTL) contains the following tag libraries:

- The JSTL `core` library contains tags for common operations such as flow control (conditionals, looping), generating output, setting variables, and creating standard HTML links. Nearly every application makes use of the core library.
- The JSTL `fmt` library contains tags that perform formatting operations such as looking up and displaying localized messages, generating formatted output using templates, parsing text, and so on. Many applications use this library.
- The JSTL `sql` library provides a tag-based approach for executing SQL statements from within JSP pages. Because this is rarely a good idea, this library is rarely used.
- The JSTL `XML` library provides tags that query and display elements and attributes within XML documents using the XPath syntax. It can be a viable alternative to using complex XSLT transformations when trying to display XML data as HTML output.
- The JSTL `functions` library contains tags that reproduce the `String` functions available in Java code. The use of this library should be avoided in favor of performing text-related searches, substrings, or other functions in Java code prior to invoking the JSP page.

A detailed look at the tags, attributes, and correct usage of each of these JSTL libraries is beyond the scope of this book. The example web application built in subsequent chapters will make extensive use of the `core` and `fmt` libraries within its JSP pages, and there will be ample opportunity at that time to highlight the important tags and their correct usage.

Best Practice

Plan to use the JSTL `core` and JSTL `fmt` libraries in all of your web applications. These two libraries provide the most generally useful tags and will be required for all but the simplest JSP pages.

Use Custom Tags for Selected Behaviors

Custom tags are a powerful mechanism for extending the basic JSP tag syntax to include custom developed tags for interacting with Java components, modifying response content, and encapsulating page logic. The JSTL tag libraries discussed previously are good examples of custom tags that can reduce or eliminate the need for scriptlet code in the JSP page and improve maintainability.

The power of custom tags comes with a cost, of course: complexity. Custom tags add an entirely new layer to the architectural picture and require a strictly defined set of classes and descriptor files to operate. Although a detailed description of the steps required to create custom tags is beyond the scope of this text, it is instructive to review the key concepts to frame the recommendations we will be making.

Custom Tag Key Concepts

Custom tags require a minimum of three components:

- The *Tag Handler Class* is a Java class implementing either the `javax.servlet.jsp.tagext.Tag` or `BodyTag` interfaces. The tag handler class defines the behavior of the tag when invoked in the JSP page by providing set methods for attributes and implementations for key methods such as `doStartTag()` and `doEndTag()`.
- The *Tag Library Descriptor* (TLD) file contains XML elements that map the tag name to the tag handler class and provide additional information about the tag. This file defines whether the tag contains and manipulates JSP body content, whether it uses a tag extra information class, and the name of the library containing this tag.
- *JSP Pages* contain `<%@ taglib ... %>` declarations for the tag library and individual tag elements in the page itself to invoke the methods contained in the tag handler class.

Custom tags may also define a *Tag Extra Information* (TEI) class, extending `javax.servlet.jsp.tagext.TagExtraInfo`, that defines the tag interface in detail and provides the names and types of scriptlet variables introduced by the tag. During page generation, the JSP engine uses the TEI class to validate the tags embedded on the page and include the correct Java code in the generated servlet to introduce variables defined by the custom tag.

Custom Tags are Different from Tag Files

Don't confuse custom tags with the new *tag files* functionality added in the JSP 2.0 specification: They are quite different in both intent and implementation.

As stated above, custom tags provide a powerful mechanism for extending the basic JSP tag syntax to include custom developed tags for interacting with Java components. They efficiently and safely replace the use of scriptlet code required to call methods on these Java components.

Tag files, on the other hand, provide a new way to include shared pieces of JSP-generated output within a main page. They are essentially an alternative to the use of `<jsp:include>` actions or `<%@ include file="..." %>` directives, and provide no direct access to utility Java classes or other Java components. Shared JSP pages and page fragments previously located with other JSP content are renamed to end with a `.tag` suffix and moved to a special `tags` directory below `WEB-INF`. These shared pages and fragments are then included in the main page content using new custom tags.

Apart from simplifying the passing of parameters to a shared JSP page, and providing the ability for the called tag file to control when the body of the calling tag is evaluated and inserted in the output, it is difficult to find any advantage that would pay for the complexity tag files add to an application. If you find the older, simpler methods for sharing JSP pages and fragments unable to accommodate your needs, consider using the new tag files mechanism, but don't convert simple JSP pages and fragments to tag files without a good reason.

We will not be covering the creation or use of tag files in this book, nor will we be using them in the example programs.

Best Practice

The JSP 2.0 tag files mechanism provides a new approach for sharing and including JSP pages and page fragments that might make sense if the older `<jsp:include>` action and `<%@include file="..." %>` directive do not meet your needs. Be sure there is sufficient value in changing to the new approach since it represents additional complexity.

Custom Tag Use Is Easy — Development Is Complex

It is important to keep the appropriate goal firmly in mind when evaluating a new technology or feature for potential use on your project. In the case of technologies such as custom tags, the goal is to improve the readability and maintainability of the JSP pages. The assumption is that by reducing or eliminating scriptlet code the page will be easier to understand and maintain, which is true enough, but the JSP pages are only one part of the total system being developed. The beans and custom tags are part of the system as well, and any improvement in maintainability of the JSP pages must be weighed against the complexity and maintenance requirements of the beans and tags themselves.

Custom tag development, in particular, is complex. The complexity is not evident until the tasks being performed become more realistic, perhaps requiring TEI classes, body content manipulation, handling of nested tags, or other more advanced behaviors. Examine the source code for some tag libraries available in the open source community to get a sense of the requirements for a realistic, production-ready tag library. Is your development team ready to tackle this level of development? Are the people being earmarked for maintenance of the application capable of maintaining, extending, or debugging problems in the tag library? These are valid questions you should consider when making your decision to build a custom tag library.

Using custom tags, on the other hand, is relatively easy. As you saw in the discussion of the JSTL libraries, it requires a simple declaration at the top of the JSP page and a few straightforward XML elements in the page to invoke a custom tag and produce the desired behavior.

In the end, the decision comes down to the benefits of using custom tags versus the effort to develop and maintain the custom tags. Clearly a tag that is developed once and used on many pages may pay for itself through the incremental benefits accrued across multiple uses. Taken to the limit, the most benefit will come from a tag used in many pages that is acquired rather than internally developed, eliminating all development and maintenance effort on the tag itself. This should be your goal: Use custom tags, but don't develop them.

Best Practice

Custom tags are easy to use but difficult to develop and maintain, so make every effort to locate and use existing tag libraries from reputable sources rather than developing your own custom tags.

Table 1-5: Custom Tag Sources

Location	Description
`http://jakarta.apache.org/taglibs`	This source has a number of open source tag libraries, providing everything from string manipulation to regular expression handling to database access. It also hosts an implementation of the JSTL specification.
`http://jakarta.apache.org/struts`	Struts is a model-view-controller framework that includes a number of useful tag libraries.
`http://www.servletsuite.com/jsp.htm`	This commercial vendor, with more than 350 different tag libraries, offers free binary download and evaluation.

In additional to the standard JSTL tag libraries packaged in WebLogic Server, useful tag libraries are available from various vendors and open source communities. Table 1-5 provides a short list to get you started in your search.

We will be using selected custom tags from the Spring MVC framework in the example application in Chapters 3 and 4 to create HTML form elements with automatic handling of posted data during processing.

Use Servlet Filtering for Common Behaviors

Servlet filtering, a feature of servlets introduced in the Servlet 2.3 specification, provides a declarative technique for intercepting HTTP requests and performing any desired preprocessing or conditional logic before the request is passed on to the final target JSP page or servlet. Filters are very useful for implementing common behaviors such as caching page output, logging page requests, providing debugging information during testing, and checking security information and forwarding to login pages. Figure 1-2 illustrates the basic components of the filtering approach and shows the incoming HTTP request passing through one or more `Filter` classes in the `FilterChain` collection defined for this page request.

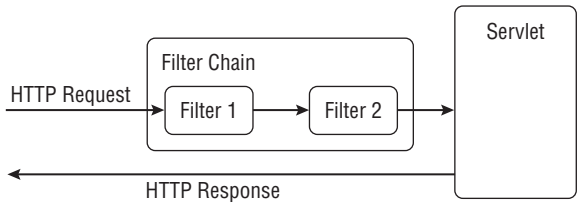

Figure 1-2: Servlet filtering.

Placing a filter in the path of a particular servlet or JSP request is a simple two-step process: Build a class that implements the `javax.servlet.Filter` interface, and register that class as a filter for the desired

pages and servlets using either entries in the `web.xml` descriptor file or annotations in the filter class. To illustrate this process, we will build and deploy a simple but useful filter that intercepts servlet and JSP requests and logs `HttpServletRequest` information before passing the request on to the intended JSP page or servlet.

Building a Simple SnoopFilter Filter Class

The first step is the construction of a filter class called `SnoopFilter` that implements the `javax.servlet.Filter` interface and performs the desired logging of request information. Simply put, the `doFilter()` method writes information from the `HttpServletRequest` object to `System.out` before forwarding to any additional filters in the filter chain or to the final destination page itself. The source for `SnoopFilter` is available from the companion web site (`http://www.wrox.com/`).

Registering SnoopFilter in the Application

Registering a filter normally requires a set of elements in the web application descriptor file, `web.xml`. These elements declare the filter class and define the pages or servlets to which the filter should be applied. In this simple example, you want all pages and servlets in the application to be filtered through `SnoopFilter`, and the `web.xml` file includes the following elements:

```
<filter>
  <filter-name>SnoopFilter</filter-name>
  <display-name>SnoopFilter</display-name>
  <description></description>
  <filter-class>
    professional.weblogic.ch01.example1.SnoopFilter
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>SnoopFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

The `<url-pattern>/*</url-pattern>` element declares that all pages and servlets in the application should be filtered using `SnoopFilter`, so every page request will go through the filter before normal processing begins. The server's `stdout` stream will therefore contain detailed request information for every page request, which is potentially very useful during development and debugging.

Clearly the same general logging capability could have been placed in a helper class, custom tag, or simple scriptlet included in each JSP page or servlet, but the ability to control the specific pages or groups of pages using the `SnoopFilter` in a declarative manner (via `<url-pattern>` elements) has significant advantages.

WebLogic Server also supports an annotation-based approach for registering a filter and specifying the URL pattern. The following `@WLFilter` syntax is equivalent to the `web.xml` entries shown above:

```
@WLFilter (
    name = "SnoopFilter",
    mapping = {"/*"}
```

```
)
public class SnoopFilter implements Filter
{
    ...
}
```

Although this is obviously a simple example, `SnoopFilter` illustrates the value of filters for preprocessing activities such as logging, auditing, or debugging in Java EE web applications. Filters are not limited to writing output to `stdout`; they can easily write information to separate log files, insert rows in database tables, call EJB components, add or modify request attributes, forward the page request to a different web application component, or perform any other desired behavior unconditionally or based on specific request information. They are a very powerful tool in the Java EE servlet specification.

Best Practice

Use filters to implement common behaviors such as logging, auditing, and security verification for servlets and JSP pages in your web applications.

Response Caching Using the CacheFilter

WebLogic Server includes a filter called `CacheFilter` that provides page-level response caching for web applications. This filter operates at the complete page level rather than surrounding and caching only a section of JSP content in a page. The `CacheFilter` may also be used with servlets and static content, unlike the related `wl:cache` custom tag, which works only in JSP pages.

The `CacheFilter` is registered like any other servlet filter. Define the filter in the `web.xml` file, and specify the `<url-pattern>` of the page or pages to cache. Use initialization parameters in the filter registration to define timeout criteria and other cache control values. For example, to cache the response from a specific JSP page for 60 seconds, register the `CacheFilter` using elements similar to the following:

```
<filter>
  <filter-name>CacheFilter1</filter-name>
  <filter-class>weblogic.cache.filter.CacheFilter</filter-class>
  <init-param>
    <param-name>timeout</param-name>
    <param-value>60</param-value>
  </init-param>
</filter>
...
<filter-mapping>
  <filter-name>CacheFilter1</filter-name>
  <url-pattern>CacheFilterTest1.jsp</url-pattern>
</filter-mapping>
```

The `CacheFilterTest1.jsp` page will execute the first time the URL is accessed by any client, and the content of the HTTP response will be cached by the filter and used for all subsequent access requests for 60 seconds.

Additional initialization parameters for the `CacheFilter` include the following:

Name The name of the cache. It defaults to the request URI.

Timeout Timeout period for the cached content. It defaults to seconds, but it may be specified in units of `ms` (milliseconds), `s` (seconds), `m` (minutes), `h` (hours), or `d` (days).

Scope The scope of the cached content. Valid values are request, session, application, and cluster. Note that `CacheFilter` does not support page scope. It defaults to application scope.

Key The names of request parameters, session attributes, and other variables used to differentiate cached content. The key is supplied using a `scope.name` syntax, with possible scope values of parameter, request, application, and session. Multiple keys can be supplied, separated by commas.

Vars The names of variables used or calculated by the page that should be cached alongside the HTTP output. When the cached version of the page is retrieved and used, these cached variables will be placed in their respective scopes as if the page had executed again. It uses the same syntax as the key parameter.

Size The maximum number of unique cache entries based on `key` values. It defaults to unlimited.

Max-cache-size The maximum size of a single cache entry. It defaults to 64k.

Very simple JSP pages or servlets may be cacheable using only a `timeout` setting as long as the output does not depend on any request or session variables. Most pages, however, will require the use of the `key` initialization parameter to create multiple cached versions of the page, one for each value of the key specified in this setting.

The `CacheTest2.jsp` example program in Listing 1-3 is an example of a page that depends on a single request parameter, `howmany`, and will require a different cached version of the output for each value of that parameter.

Listing 1-3: CacheTest2.jsp.

```
<HTML>
<BODY>
<%
int jj = Integer.parseInt(request.getParameter("howmany"));
System.out.println("Inside JSP page with howmany of " + jj);
%>
<H2>We're going to count from 1 to <%= jj %><H2>
<%
for (int ii = 1; ii <= jj; ii++) {
    out.print(ii + "<br>");
}
%>
</BODY>
</HTML>
```

The `CacheFilter` would be registered to cache this page content with a dependency on the `howmany` request parameter as follows:

```
<filter>
  <filter-name>CacheFilter2</filter-name>
```

```
    <filter-class>weblogic.cache.filter.CacheFilter</filter-class>
    <init-param>
      <param-name>timeout</param-name>
      <param-value>60</param-value>
    </init-param>
    <init-param>
      <param-name>key</param-name>
      <param-value>parameter.howmany</param-value>
    </init-param>
  </filter>
  ...
  <filter-mapping>
    <filter-name>CacheFilter2</filter-name>
    <url-pattern>CacheFilterTest2.jsp</url-pattern>
  </filter-mapping>
```

Accessing this page with a specific value of `howmany` in the query string causes the entire page to be executed one time. Subsequent page hits with the same `howmany` parameter value will return the same content without executing the page. Supplying a different value for `howmany` will cause the page to be executed for that value and the contents cached using that key value. In other words, if you hit the page five times with different `howmany` values, you've created five different cached versions of the HTTP response using the `howmany` value as the key. This technique is very slick and very powerful for improving site performance.

WebLogic Server also includes a `wl:cache` custom tag that provides a very similar caching capability for any JSP content placed in the body of the custom tag. It has the ability to cache based on key values, like `CacheFilter`, and can cache small portions of a JSP page rather than an entire page. However, the `CacheFilter` approach has an obvious advantage over the `wl:cache` technique: Caching is performed using a declarative technique rather than embedding custom tags in the page itself. This defers the definition of caching behavior to deployment time and allows easier control of the caching parameters and scope using the `web.xml` descriptor elements.

Best Practice

Use the `CacheFilter` instead of `wl:cache` tags for page-level response caching whenever possible to provide better flexibility during deployment.

Note that a JSP page included using the `<jsp:include>` action is considered a separate page for the purposes of caching. It can therefore be configured to cache independently from the parent page, which may be helpful. Recognize, however, that this included page will not be invoked if the parent page execution is skipped due to caching. Plan accordingly!

Creating Excel Files Using Servlets and JSP Pages

Creating spreadsheets using servlets and JSP pages is a useful way to provide users with results they can sort, manipulate, and print using Microsoft Excel or other spreadsheet applications. Servlets are the preferred mechanism, but JSP pages can also be used if you take steps to avoid unintended newline characters in the output stream.

To create a spreadsheet using a servlet, build the servlet in the normal manner but set the content type to `application/vnd.ms-excel` in the response header to indicate that the response should be interpreted as a spreadsheet. Data written to the response `Writer` object will be interpreted as spreadsheet data, with tabs indicating column divisions and newline characters indicating row divisions. For example, the `SimpleExcelServlet` servlet in Listing 1-4 creates a multiplication table using simple tabs and newlines to control the rows and columns in the result.

Listing 1-4: SimpleExcelServlet.java.

```
package professional.weblogic.ch01.example2;

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

public class SimpleExcelServlet extends HttpServlet
{
    public static final String CONTENT_TYPE_EXCEL =
        "application/vnd.ms-excel";

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException
    {
        PrintWriter out = response.getWriter();
        response.setContentType(CONTENT_TYPE_EXCEL);

        out.print("\t"); // empty cell in upper corner
        for (int jj = 1; jj <= 10; jj++) {
            out.print("" + jj + "\t");
        }
        out.print("\n");

        for (int ii = 1; ii <= 10; ii++) {
            out.print("" + ii + "\t");
            for (int jj = 1; jj <= 10; jj++) {
                out.print("" + (ii * jj) + "\t");
            }
            out.print("\n");
        }
    }
}
```

Normal registration of this servlet in `web.xml` is all that is required in most cases.

```
<servlet>
  <servlet-name>SimpleExcelServlet</servlet-name>
  <servlet-class>
    professional.weblogic.ch01.example2.SimpleExcelServlet
  </servlet-class>
</servlet>

<servlet-mapping>
```

```
  <servlet-name>SimpleExcelServlet</servlet-name>
  <url-pattern>/simpleexcel</url-pattern>
</servlet-mapping>
```

As noted earlier, WebLogic Server supports a `@WLServet` annotation for registering servlets and specifying URL patterns. In this case, the `SimpleExcelServlet` source file could include the following `@WLServlet` annotation to eliminate the need for `web.xml` entries:

```
@WLServlet (
    name = "SimpleExcelServlet",
    mapping = {"/simpleexcel"}
)
public class SimpleExcelServlet extends HttpServlet
{
    ...
}
```

In both registration approaches, users accessing the `/simpleexcel` location will be presented with a spreadsheet embedded in their browser window. The servlet may also be registered for a `<url-pattern>` that includes an `.xls` file extension to assist the users by providing a suitable default file name and type if they choose to use `Save As...` from within the browser:

```
<servlet-mapping>
  <servlet-name>SimpleExcelServlet</servlet-name>
  <url-pattern>/multitable.xls</url-pattern>
</servlet-mapping>
```

Simple tab- and newline-based formatting may be sufficient in many cases, but you can achieve additional control by building HTML tables and using HTML formatting options such as `<b>` and `<i>` in the generated output. Because the content type was specified as `ms-excel`, these HTML tags are interpreted by the browser and spreadsheet application as equivalent spreadsheet formatting options.

The `FancyExcelServlet` example servlet in Listing 1-5 builds the same multiplication table as `SimpleExcelServlet` but uses HTML to control formats and cell sizes.

Listing 1-5: FancyExcelServlet.java.

```
package professional.weblogic.ch01.example3;

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

public class FancyExcelServlet extends HttpServlet
{
    public static final String CONTENT_TYPE_EXCEL =
        "application/vnd.ms-excel";

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException
```

```
    {
        PrintWriter out = response.getWriter();
        response.setContentType(CONTENT_TYPE_EXCEL);

        out.print("<table border=1>");
        out.print("<tr>");
        out.print("<td> </td>"); // empty cell in upper corner
        for (int jj = 1; jj <= 10; jj++) {
            out.print("<td><b>" + jj + "</b></td>");
        }
        out.print("</tr>");

        for (int ii = 1; ii <= 10; ii++) {
            out.print("<tr>");
            out.print("<td><b>" + ii + "</b></td>");
            for (int jj = 1; jj <= 10; jj++) {
                out.print("<td>" + (ii * jj) + "</td>");
            }
            out.print("</tr>");
        }
        out.print("</table>");
    }
}
```

You can also use JSP pages to create spreadsheets with one complication: The output of a JSP page often contains many unintended newline characters caused by extra whitespace around directives and scriptlet tags, making it difficult to control the spreadsheet formatting when using simple tab and newline techniques. HTML formatting similar to the `FancyExcelServlet` works better in JSP pages used to create spreadsheets. Listing 1-6 presents the JSP equivalent to the `FancyExcelServlet`.

Listing 1-6: FancyExcelPage.jsp.

```
<% response.setContentType("application/vnd.ms-excel"); %>
<html>
<body>
<table border=1>
<tr>
  <td> </td>
  <% for (int jj = 1; jj <= 10; jj++) { %>
    <td><b><%= jj %></b></td>
  <% } %>
</tr>
<% for (int ii = 1; ii <= 10; ii++) { %>
  <tr>
    <td><b><%= ii %></b></td>
    <% for (int jj = 1; jj <= 10; jj++) { %>
      <td><%= (ii * jj) %></td>
    <% } %>
  </tr>
<% } %>
</table>
</body>
</html>
```

Viewing Generated Servlet Code

Viewing the servlet code generated for a particular JSP page can be instructive while learning JSP technology and useful during the testing and debugging process. Often the error report received during the execution of the JSP page indicates the line in the generated servlet code, but finding the JSP scriptlet code or tag that caused the error requires inspection of the Java code.

Generated Java servlet code will be kept alongside the generated servlet class files if the `keepgenerated` parameter is set to `true` in the `<jsp-descriptor>` section of the `weblogic.xml` descriptor file. The equivalent option for keeping the generated Java code for JSP pages compiled using the `weblogic.appc` utility or `wlappc` Ant task is `keepgenerated` placed on the command line or within the Ant task invocation.

By default, the generated servlet classes and Java code will be placed in a temporary directory structure located under the domain root directory. The name of this temporary directory depends on the names of the server, enterprise application, and web application, and it typically looks something like `servers/myserver/tmp/_WL_user/_appsdir_myapp_dir/wx8qxk/jsp_servlet`. This default location may be overridden using the `<working-dir>` option in the `weblogic.xml` descriptor file.

Chapter Review

In this chapter we reviewed the key concepts related to web applications in WebLogic Server and presented a number of important best practices designed to improve the quality and performance of your web applications.

Most of this chapter has been at the detailed design and implementation level, the *trees*, in a sense. In the next two chapters we step back and look at the *forest* for a few minutes by examining the importance of the overall web application architecture, the selection of a suitable presentation template technique, and the application of a model-view-controller pattern and framework for form and navigation handling.