Chapter 3. Java Servlets

3.1. Getting a Servlet Environment

You need a servlet container to run servlets. A servlet container uses a Java virtual machine to run servlet code as requested by a web server. The servlet container is also responsible for managing other aspects of the servlet lifecycle : user sessions, classloading, servlet contexts, servlet configuration information, servlet persistence, and temporary storage.

Because Tomcat is the reference implementation for the Servlet API, all the examples in this chapter have been tested with it. Tomcat 4.x supports the 2.3 API, and Tomcat 5.x supports the 2.4 API. Since Tomcat falls under the Apache umbrella, distribution is free, and you can download a copy (including, if you like, full source code) from http://jakarta.apache.org.Binary installations are available for Windows and several Unix flavors.

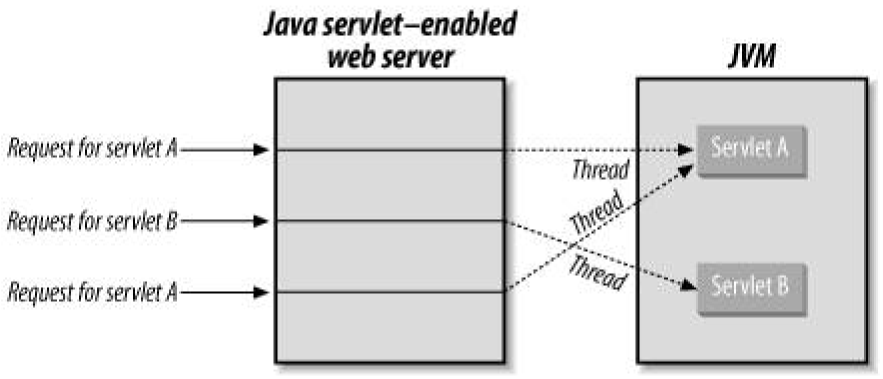
3.2. Servlet Basics

The Servlet API consists of two packages, `javax.servlet` and `javax.servlet.http`. The `javax` is left over from an earlier stage of Java package naming conventions. As mentioned (and as indicated by the fact that this chapter appears in Part I of this book), servlets are a standard part of J2EE.

3.2.1. The Servlet Lifecycle

When a client makes a request involving a servlet, the server loads and executes the appropriate Java classes. Those classes generate content, and the server sends the content back to the client. In most cases, the client is a web browser, the server is a web server, and the servlet returns standard HTML. From the web browser's perspective, this isn't any different from requesting a page generated by a CGI script or, indeed, a static HTML file. On the server side, however, there is

Figure 3-1. The servlet lifecycle



an important difference: persistence.[*] Instead of shutting down at the end of each request, the servlet can remain loaded, ready to handle subsequent requests. Figure 3-1 shows how this all fits together.

[*] Note that we use "persistent" to mean "enduring between invocations," not "written to permanent storage."

The request processing time for a servlet can vary, but it is typically quite fast when compared to a similar CGI program. The real performance advantage of a servlet is that you incur most of the startup overhead only once. Most of the I/O-intensive resources (such as database connection pools) your application will need can be created by the servlet container at startup and shared by all your servlets. Instead of connecting to the databaseor just loading your codethousands of times a day, the container loads it only once. When the container loads a servlet for the first time, it calls the `init (ServletConfig)` method, which is allowed to complete before the servlet is asked to respond to any

requests. After the `init( )` method runs, the servlet container marks the servlet as available. For each incoming connection directed at a particular servlet, the container calls the `service( )` method on the servlet to process the request. The `service( )` method can have access to all the resources created in the `init( )` method. The servlet's `destroy( )` method is called to clean up resources when the server shuts down.

The `init( )` method used to be an important way to create I/O-intensive resources. In more recent versions of the Servlet API, it has largely been supplanted by resource setup at the application level (via `ServletContextListener`, discussed later in this chapter) and at the container level.

Because servlets are persistent, you can actually remove a lot of filesystem and/or database access altogether. For example, to implement a page counter, you can simply store a number in a static variable rather than consulting a file (or database) for every request. Using this technique, you need to read and write to the disk only occasionally to preserve state. Since a servlet remains active, it can perform other tasks when it is not servicing client requests, such as running a background processing thread (i.e., where clients connect to the servlet to view a result) or even acting as an RMI host, enabling a single servlet to handle connections from multiple types of clients. For example, if you write an order processing servlet, it can accept transactions from both an HTML form and an applet using RMI.

The Servlet API includes numerous methods and classes for making application development easier. Most common CGI tasks require a lot of fiddling on the programmer's part; even decoding HTML form parameters can be a chore, to say nothing of dealing with cookies and session tracking. Libraries exist to help with these tasks, but they are, of course, decidedly nonstandard. You can use the Servlet API to handle most routine tasks, thus cutting development time and keeping things consistent for multiple developers on a project.

3.2.2. Writing Servlets

The three core elements of the Servlet API are the `javax.servlet.Servlet` interface, the `javax.servlet.GenericServlet` class, and the `javax.servlet.http.HttpServlet` class. Normally, you create a servlet by subclassing one of the two classes, although if you are adding servlet capability to an existing object, you may find it easier to implement the interface.

The `GenericServlet` class is used for servlets that don't implement any particular communication protocol. Here's a basic servlet that demonstrates servlet structure by printing a short message:

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

public class BasicServlet extends GenericServlet {
  public void service(ServletRequest req, ServletResponse resp)
    throws ServletException, IOException {

    resp.setContentType("text/plain");
    PrintWriter out = resp.getWriter(  );
   // We won't use the ServletRequest object in this example
    out.println("Hello.");
  }
}
```

`BasicServlet` extends the `GenericServlet` class and implements one method: `service( )`. Whenever a server wants to use the servlet, it calls the `service( )` method, passing `ServletRequest` and `ServletResponse` objects (we'll look at these in more detail shortly). The servlet tells the server what type of response to expect, gets a `PrintWriter` from the response object, and transmits its output.

Why Two Classes?

The designers of the original Servlet API expected that it would used for a variety of protocols above and beyond simple HTTP, hence the decision to create both a `GenericServlet` and an `HttpServlet` class. In practice, this never happened: APIs for other protocols use different mechanisms for content management. Prior to Version 2.3, generic servlets were often used to filter content from other servlets, a role now assumed by the `Filter` interface. Most servlets subclass `HttpServlet`.

3.2.3. HTTP Servlets

The `HttpServlet` class is an extension of `GenericServlet` that includes methods for handling HTTP-specific data.[*] `HttpServlet` provides a number of methods, such as `doGet( )`, `doPost( )`, and `doPut( )`, to handle particular types of HTTP requests (GET, POST, and so on). These methods are called by the default implementation of the `service( )` method, which figures out what kind of request is being made and then invokes the appropriate method. Here's a simple `HttpServlet`:

[*] `HttpServlet` is an abstract class, implemented by your servlet classes.

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

public class HelloWorldServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter(  );

    out.println("<html>");
    out.println(
        "<head><title>Have you seen this before?</title></head>");
    out.println(
        "<body><h1>Hello, World!</h1><h6>Again.</h6></body></html>");
  }
}
```

`HelloWorldServlet` demonstrates many essential servlet concepts. First, `HelloWorldServlet` extends `HttpServlet`. This is standard practice for an HTTP servlet. `HelloWorldServlet` defines one method, `doGet( )`, which is called whenever anyone requests a URL that points to this servlet. The `doGet( )` method is actually called by the default `service( )` method of `HttpServlet`. The `service( )` method is called by the web server when a request is made of `HelloWorldServlet`; the method determines what kind of HTTP request is being made and dispatches the request to the appropriate `do`*`XXX`*`( )` method (in this case, `doGet( )`). `doGet( )` is passed two objects, `HttpServletRequest` and `HttpServletResponse`, that contain information about the request and provide a mechanism for the servlet to produce output, respectively.

The `doGet( )` method itself does three things. First, it sets the output type to `text/html`, which indicates that the servlet produces standard HTML as its output. Second, it calls the `getWriter( )` method of the `HttpServletResponse` parameter to get a `java.io.PrintWriter` that points to the client. Finally, it uses the stream to send some HTML back to the client. This isn't really a whole lot different from the `BasicServlet` example, but it gives us all the tools we'll need later on for more complex web applications. We do have to explicitly set the content type, as there is no default setting, even for HTTP servlets where one might reasonably expect `text/html`.

If you define a `doGet( )` method for a servlet, you may also want to override the `getLastModified( )` method of `HttpServlet`. The server calls `getLastModified( )` to find out if the content delivered by a servlet has changed. The default implementation of this method returns a negative number, which tells the server that the servlet doesn't know when its content was last updated, so the server is forced to call `doGet( )` and return the servlet's output. If you have a servlet that changes its display data infrequently (such as a servlet that verifies uptime on several server machines once every 15 minutes), you should implement `getLastModified( )` to allow browsers to cache responses. `getLastModified( )` should return a `long` value that represents the time the content was last modified as the number of milliseconds since midnight, January 1, 1970, GMT (returned by calling the `getTime( )` method `java.util.Date`).

A servlet should also implement `getServletInfo( )`, which returns a `String` that contains information about the servlet, such as name, author, and version (just like `getAppletInfo( )` in applets). This method is called by the web server and generally used for logging purposes. It's rarely implemented in the real world, but it really is good practice to do so.

3.3. Web Applications

Now that we've seen a basic servlet, we can step back for a moment and talk about how servlets are integrated into the servlet container. A web application consists of a set of resources, including servlets, static content, JSP files, and class libraries, installed in a particular path on a web server. This path is called the servlet context, and all servlets installed within the context are given an isolated, protected environment to operate in, without interference from (or the ability to interfere with) other software running on the server.

A servlet context directory tree contains several different types of resources. These include class files and jar files (which aren't exposed to clients connecting via web browsers), JSP files (which are processed by the JSP servlet before being fed back to the client), and static files, such as HTML documents and JPEG images, which are served directly to the browser by the web server.

The context has a virtual component, too. For each context, the servlet container will instantiate separate copies of servlets and will create a private address space that can be accessed via the `ServletContext` class. Servlets running in the same context can use this class to communicate with each other. We'll discuss this more later.

The simplest servlet installations create just a single context, rooted at /, which is the top of the web server path tree. Servlets and static content are installed within this context. Most of the time, you'll create multiple servlet contexts , rooted lower down on the directory tree. A catalog application, for example, could be rooted at /catalog, with all of the application paths below the context root.

If you write a web application that will be installed on multiple web servers, it isn't safe to assume the context root will be fixed. If the path of a resource within your application is /servlet/CatalogServlet, and it's installed within the /catalog context, rather than writing:

```
out.println("<a href=\"/catalog/servlet/CatalogServlet\">");
```

you should write:

```
out.println(
  "<a href=\"" + request.getContextPath(  ) + "/servlet/CatalogServlet\">");
```

This approach works regardless of the context path installed within the web server. Of course, in most cases, you shouldn't be writing HTML directly out of the servlet anywaysee Chapters 4, 5, and 19 on JSP, JSF, and Struts for more on the separation of content and business logic in web applications.

3.3.1. Structure of Web Applications

On disk, a web application consists of a directory. The directory contains a subdirectory called WEB-INF and whatever other content is required for the application. The WEB-INF directory contains a *classes* directory (containing application code), a *lib* directory (containing application jar files), and a file called web.xml. The web.xml file contains all of the configuration information for the servlets within the context, including names, path mappings, initialization parameters, and context-level configuration information.

The procedure for installing a web application into a servlet container varies from product to product, but it generally consists of selecting a context root and pointing the server to the directory containing the web application.[*]

> [*] Web applications can be packaged into jar file equivalents called war files. Just use the jar utility that comes with the JDK to pack up the web application directory (including the WEB-INF subdirectory) and give the resulting file a .war extension, or use the `war` target in an Ant script.

3.3.2. Mapping Requests with a Context

Servlets are installed within the servlet container and mapped to URIs. This is done either via global properties that apply to all servlets or by specific, servlet-by-servlet mappings. In the first case, a client invokes a servlet by requesting it by name. Some servers map servlets to a /servlet/ or /servlets/ URL. If a servlet is installed as `PageServlet`, a request to /servlet/PageServlet would invoke it. Servlets can also be individually mapped to other URIs or to file extensions. `PageServlet` might be mapped to /pages/page1 or to all files with a .page extension (using *.page).

All of these mappings exist below the context level. If the web application is installed at /app, the paths entered into the browser for the examples earlier would be /app/servlet/PageServlet, /app/pages/page1, or /app/file.page. Of course, if they are entered into a browser, these paths represent only the end of the full URL, which must also include the protocol, host, and port information (e.g., http://myserver.org:8080).

To illustrate, imagine the servlet mappings (all are below the context root) in Table 3-1.

Table 3-1. Example servlet mappings

Mapping	Servlet
*/store/furniture/**	`FurnitureServlet`
*/store/furniture/tables/**	`TableServlet`
/store/furniture/chairs	`ChairServlet`
**.page*	`PageServlet`

The asterisk serves as a wildcard. URIs matching the pattern are mapped to the specified servlet, provided that another mapping hasn't already been used to deal with the URL. This can get a little tricky when building complex mapping relationships, but the Servlet API does require servers to deal with mappings consistently. When the servlet container receives a request, it always maps it to the appropriate servlet in the following order:

1. Exact path matching

   A request to /store/furniture/chairs is served by `ChairServlet`.

2. Prefix mapping

   A request to /store/furniture/sofas is served by `FurnitureServlet`. The longest matching prefix is used. A request to /store/furniture/tables/dining is served by `TableServlet`.

3. Extension

   Requests for /info/contact.page are served by `PageServlet`. However, requests for /store/furniture/chairs/about.page is served by `FurnitureServlet` (since prefix mappings are checked first, and `ChairServlet` is available only for exact matches).

If no appropriate servlet is found, the server returns an error message or attempts to serve content on its own. If a servlet is mapped to the / path, it becomes the default servlet for the application and is invoked when no other servlet is found.

3.3.3. Context Methods

Resources within a servlet context (such as HTML files, images, and other data) can be accessed directly via the web server. If a file called index.html is stored at the root of the /app context, then it can be accessed with a request to /app/index.html. Context resources can also be accessed via the `ServletContext` object, which is accessed via the `getresource( )` and `getresourceAsStream( )` methods . A full list of available resources can be accessed via the `getresourcePaths( )` method. In this case, an `InputStream` containing the contents of the index.html file can be retrieved by calling `getresourceAsStream("/index.html")` on the `ServletContext` object associated with the /app context.

The `ServletContext` interface provides servlets with access to a range of information about the local environment. The `getInitParameter( )` and `getInitParameterNames( )` methods allow servlets to retrieve context-wide initialization parameters. `ServletContext` also includes a number of methods that allow servlets to share attributes. The `setAttribute( )` method allows a servlet to set an attribute that can be shared by any other servlets that live in its `ServletContext`, and `removeAttribute( )` allows a servlet to remove an attribute from the application context. The `getAttribute( )` method, which previously allowed servlets to retrieve hardcoded server attributes, provides access to attribute values, while `getAttributeNames( )` returns an `Enumeration` of all the shared attribute names.

The servlet container is required to maintain a temporary working directory on disk for each servlet context. This directory is accessed by retrieving the `javax.servlet.context.tempdir` attribute, which consists of a `java.io.File` object pointing to the temporary directory. Each servlet must have its own temporary directory. The servlet container is not required to maintain its contents across restarts, so if you store something there, it may not be there if the servlet container restarts.

3.4. Servlet Requests

When a servlet handles a request, it typically needs specific information about the request so that it can respond appropriately. Most frequently, a servlet retrieves the value of a form variable and uses that value in its output. A servlet may also need access to information about the environment in which it is running. For example, a servlet may need to authenticate the user who is accessing the servlet.

The `ServletRequest` and `HttpServletRequest` interfaces provide access to this kind of information. When a servlet is asked to handle a request, the servlet container passes it a request object that implements one of these interfaces. With this object, the servlet can determine the actual request (e.g., protocol, URL, type), access parts of the raw request (e.g., headers, input stream), and get any client-specific request parameters (e.g., form variables, extra path information). For instance, the `getProtocol ( )` method returns the protocol used by the request, while `geTRemoteHost( )` returns the name of the client host. The interfaces also provide methods that let a servlet get information about the server (e.g., `getServername( )`, `getServerPort( )`). As we saw earlier, the `getParameter( )` method provides access to request parameters such as form variables. There is also the `getParameterValues( )` method, which returns an array of strings that contains all the values for a particular parameter. This array generally contains only one string, but some HTML form elements (as well as non-HTTP-oriented services) do allow multiple selections or options, so the method always returns an array, even if it has a length of 1.

`HttpServletRequest` adds a few more methods for handling HTTP-specific request data. For instance, `getHeaderNames( )` returns an enumeration of the names of all the HTTP headers submitted with a request, while `getHeader( )` returns a particular header value. Other methods handle cookies and sessions, as we'll discuss later.

Example 3-1 shows a servlet that restricts access to users who are connecting via the HTTPS protocol, using digest-style authentication and coming from a government site (a domain ending in .gov). The restrictions are implemented by checking information pulled from the servlet request using many of the methods described earlier.

Example 3-1. Checking request information to restrict servlet access

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

public class SecureRequestServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter(  );

    out.println("<html>");
    out.println("<head><title>Semi-Secure Request</title></head>");
    out.println("<body>");

    String remoteHost = req.getRemoteHost(  );
    String scheme = req.getScheme(  );
    String authType = req.getAuthType(  );

    if((remoteHost == null) || (scheme == null) || (authType == null)) {
      out.println("Request Information Was Not Available.");
      return;
    }

    if(scheme.equalsIgnoreCase("https") && remoteHost.endsWith(".gov")
       && authType.equals("Digest")) {
      out.println("Special, secret information.");
    }
    else {
      out.println("You are not authorized to view this data.");
    }

    out.println("</body></html>");
  }
}
```

3.4.1. Forms and Interaction

The problem with creating a servlet like our earlier `HelloWorldServlet` is that it doesn't do anything we can't already do with HTML. If we are going to bother with a servlet at all, we should do something dynamic and interactive with it. In many cases, this means processing the results of an HTML form. To make our example less impersonal, let's have it greet the user by name. The HTML form that calls the servlet using a GET request might look like this:

```
<html>
<head><title>Greetings Form</title></head>
<body>
<form method=get action="/servlet/HelloServlet">
What is your name?
<input type=text name="username" size=20>
<input type=submit value="Introduce Yourself">
</form>
</body>
</html>
```

This form submits a form variable named `username` to /servlet/HelloServlet. The `HelloServlet` itself does little more than create an output stream, read the `username` form variable, and print a nice greeting for the user. Here's the code:

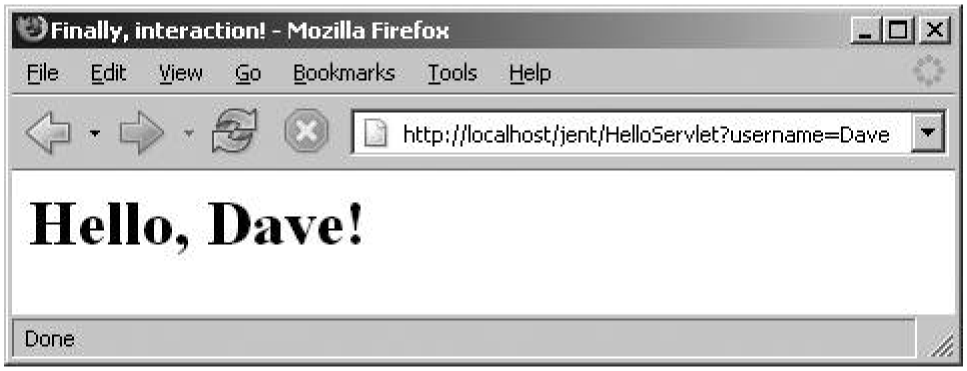
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HelloServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter(  );

    out.println("<html>");
    out.println("<head><title>Finally, interaction!</title></head>");
    out.println("<body><h1>Hello, " + req.getParameter("username")
                   + "!</h1>");
    out.println("</body></html>");
  }
}
```

All we've done differently is use the `getParameter( )` method of `HttpServletRequest` to retrieve the value of a form variable.[*] When a server calls a servlet, it can also pass a set of request parameters. With HTTP servlets, these parameters come from the HTTP request itselfin this case, in the guise of URL-encoded form variables. Note that a `GenericServlet` running in a web server also has access to these parameters using the simpler `ServletRequest` object. When `HelloServlet` runs, it inserts the value of the `username` form variable into the HTML output, as shown in Figure 3-2.

[*] For those interested in ancient history, in Java Web Server 1.1, the `getParameter( )` method was deprecated in favor of `getParameterValues( )`, which returns a `String` array rather than a single string. However, after an extensive write-in campaign, Sun took `getParameter( )` off the deprecated list for Version 2.0 of the Servlet API, so you can

safely use this method in your servlets. This is not an issue with later versions of the API, and the fact that we even talk about it dates us.

Figure 3-2. Output from HelloServlet


Finally, interaction! - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/jent/HelloServlet?username=Dave
Done

Hello, Dave!

3.4.2. POST, HEAD and Other Requests

As mentioned earlier, `doGet( )` is just one of a collection of enabling methods for HTTP request types. `doPost( )` is the corresponding method for POST requests . The POST request is designed for posting information to the server, although in practice it is also used for long parameterized requests and larger forms, to get around limitations on the length of URLs.

If your servlet is performing database updates, charging a credit card, or doing anything that takes an explicit client action, you should make sure this activity is happening in a `doPost( )` method because POST requests aren't idempotent, which means that they aren't safely repeatable, and web browsers treat them specially. For example, a browser can't bookmark or, in some cases, reload a POST request. On the other hand, GET requests are idempotent, so they can safely be bookmarked, and a browser is free to issue the request repeatedly without necessarily consulting the user. You can see why you don't want to charge a credit card in a GET method!

To create a servlet that can handle POST requests, all you have to do is override the default `doPost( )` method from `HttpServlet` and implement the necessary functionality in it. If necessary, your application can implement different code in `doPost( )` and `doGet( )`. For instance, the `doGet( )` method might display a postable data entry form that the `doPost( )` method processes. `doPost( )` can even call `doGet( )` at the end to display the form again.

The less common HTTP request types, such as HEAD, PUT, TRACE, and DELETE, are handled by other `do`*`XXX`*`( )` dispatch methods. A HEAD request returns HTTP headers only, PUT and DELETE allow clients to create and remove resources from the web server, and TRACE returns the request headers to the client. Since most servlet programmers don't need to worry about these requests, the `HttpServlet` class includes a default implementation of each corresponding `do`*`XXX`*`( )` method that either informs the client that the request is unsupported or provides a minimal implementation. You can provide your own versions of these methods, but the details of implementing PUT or DELETE functionality go rather beyond our scope.

3.5. Servlet Responses

In order to do anything useful, a servlet must send a response to each request that is made to it. In the case of an HTTP servlet, the response can include three components: a status code, any number of HTTP headers, and a response body.

The `ServletResponse` and `HttpServletResponse` interfaces include all the methods needed to create and manipulate a servlet's output. We've already seen that you specify the MIME type for the data returned by a servlet using the `setContentType( )` method of the response object passed into the servlet. With an HTTP servlet, the MIME type is generally `text/html`, although some servlets return binary data: a servlet that loads a GIF file from a database and sends it to the web browser should set a content type of `image/gif`, while a servlet that returns an Adobe Acrobat file should set it to `application/pdf`.

`ServletResponse` and `HttpServletResponse` each define two methods for producing output streams, `getOutputStream( )` and `getWriter( )`. The former returns a `ServletOutputStream`, which can be used for textual or binary data. The latter returns a `java.io.PrintWriter` object, which is used only for textual output. The `getWriter( )` method examines the content type to determine which charset to use, so `setContentType( )` should be called before `getWriter( )`.

`HttpServletResponse` also includes a number of methods for handling HTTP responses . Most of these allow you to manipulate the HTTP header fields. For example, `setHeader( )`, `setIntHeader( )`, and `setDateHeader( )` allow you to set the value of a specified HTTP header, while `containsHeader( )` indicates whether a certain header has already been set. You can use either the `setStatus( )` or `sendError( )` method to specify the status code sent back to the client. `HttpServletResponse` provides a long list of integer constants that represent specific HTTP status codes (we'll see some of these shortly). You typically don't need to worry about setting a status code, as the default code is 200 ("OK"), meaning that the servlet sent a normal response. However, a servlet that is part of a complex application structure (such as serving XML content to an AJAX-based dynamic web interface) may need to use a variety of status codes. Finally, the `sendRedirect( )` method allows you to issue a page redirect. Calling this method sets the `Location` header to the specified location and uses the appropriate status code for a redirect.

3.5.1. Request Dispatching

Request dispatching allows a servlet to delegate request handling to other components on the server. A servlet can either forward an entire request to another servlet or include bits of content from other components in its own output. In either case, this is done with a `RequestDispatcher` object that is obtained from the `ServletContext` via the `getrequestDispatcher( )` method (also available via the `HttpServletRequest` object). When you call this method, you specify the path to the servlet to which you are dispatching the request. The path should be relative to the servlet context. If you want to dispatch a request to /servlet/TargetServlet within the /app context (which is accessed from a user's browser by /app/servlet/TargetServlet), request a dispatcher for /servlet/TargetServlet.

When you dispatch a request, you can set request attributes using the `setAttribute( )` method of `ServletRequest` and read them using the `getAttribute( )` method. A list of available attribute names is returned by `getAttributeNames( )`. Rather than taking only `String` objects (like parameters), an attribute may be any valid Java object.

`RequestDispatcher` provides two methods for dispatching requests: `forward( )` and `include( )`. To forward an entire request to another servlet, use the `forward( )` method. When using `forward( )`, the `ServletRequest` object is updated to include the new target URL. If a `ServletOutputStream` or `PrintWriter` has already been retrieved from the `ServletResponse` object, the `forward( )` method throws an `IllegalStateException`.

The `include( )` method of `RequestDispatcher` causes the content of the dispatchee to be included in the output of the main servlet, just like a server-side include. To see how this works, let's look at part of a servlet that does a keep-alive check on several different servers. The `ServerMonitorServlet` referenced in this example relies on the `serverurl` attribute to determine which server to display monitoring information for:

```
out.println("Uptime for our servers");
// Get a RequestDispatcher to the ServerMonitorServlet
RequestDispatcher d = getServletContext( ).
     getRequestDispatcher("/servlet/ServerMonitorServlet");
req.setAttribute("serverurl", new URL("http://www1.company.com"));
d.include(req, res);
req.setAttribute("serverurl", new URL("http://www2.company.com"));
d.include(req, res);
```

Request dispatching is obviously different from issuing an HTTP redirect (via the `sendRedirect( )` method of `HttpResponse`) since everything takes place within a single server request. Using a forward rather than a redirect gives a better user experience since content is displayed in the browser without two round-trips to the server. However, when you forward a request, the apparent URL (from the browser's perspective) remains the original request URL. This can break relative links and sometimes leads to confusing caching behavior on the browser. It is a good practice to avoid relative links wherever possibleencode the full server-side path, starting with the context name, instead.

3.5.2. Error Handling

Sometimes things just go wrong. When that happens, it's nice to have a clean way out. The Servlet API gives you two ways of to deal with errors: you can manually send an error message back to the client or you can throw a `ServletException`. The easiest way to handle an error is simply to write an error message to the servlet's output stream. This is the appropriate technique to use when the error is part of a servlet's normal operation, such as when a user forgets to fill in a required form field.

3.5.2.1. Status codes

When an error is a standard HTTP error, you should use the `sendError( )` method of `HttpServletResponse` to tell the server to send a standard error status code. `HttpServletResponse` defines integer constants for all the major HTTP status codes . Table 3-2 lists the most common status codes. For example, if a servlet can't find a file the user has requested, it can send a 404 ("File Not Found") error and let the browser display it in its usual manner. In this case, we can replace the typical `setContentType( )` and `getWriter( )` calls with something like this:

```
response.sendError(HttpServletResponse.SC_NOT_FOUND);
```

If you want to specify your own error message (in addition to the web server's default message for a particular error code), you can call `sendError( )` with an extra `String` parameter:

```
response.sendError(HttpServletResponse.SC_NOT_FOUND,
                   "It's dark. I couldn't find anything.");
```

Table 3-2. Some common HTTP status codes

Constant	Code	Default message	Meaning
`SC_OK`	200	OK	The client's request succeeded, and the server's response contains the requested data. This is the default status code.
`SC_NO_CONTENT`	204	No Content	The request succeeded, but there is no new response body to return. A servlet may find this code useful when it accepts data from a form, but wants the browser view to stay at the form. It avoids the "Document contains no data" error message.
`SC_MOVED_PERMANENTLY`	301	Moved Permanently	The requested resource has permanently moved to a new location. Any future reference should use the new location given by the `Location` header. Most browsers automatically access the new location.
`SC_MOVED_TEMPORARILY`	302	Moved Temporarily	The requested resource has temporarily moved to another location, but future references should still use the original URL to access the resource. The temporary new location is given by the `Location` header. Most browsers automatically access the new location.
`SC_UNAUTHORIZED`	401	Unauthorized	The request lacked proper authorization. Used in conjunction with the `WWW-Authenticate` and `Authorization` headers.
`SC_NOT_FOUND`	404	Not Found	The requested resource is not available.
`SC_INTERNAL_SERVER_ERROR`	500	Internal Server Error	An error occurred inside the server that prevented it from fulfilling the request.
`SC_NOT_IMPLEMENTED`	501	Not Implemented	The server doesn't support the functionality needed to fulfill the request.
`SC_SERVICE_UNAVAILABLE`	503	Service Unavailable	The server is temporarily unavailable, but service should be restored in the future. If the server knows when it will be available again, a `Retry-After` header may also be supplied.

3.5.2.2. Servlet exceptions

The Servlet API includes two `Exception` subclasses, `ServletException` and its derivative, `UnavailableException`. A servlet throws a `ServletException` to indicate a general servlet problem.

When a server catches this exception, it can handle the exception however it sees fit.

`UnavailableException` is a bit more useful, however. When a servlet throws this exception, it is notifying the server that it is unavailable to service requests. You can throw an `UnavailableException` when some factor beyond your servlet's control prevents it from dealing with requests. To throw an exception that indicates permanent unavailability, use something like this:

```
throw new UnavailableException(this,
            "This is why you can't use the servlet.");
```

`UnavailableException` has a second constructor to use if the servlet is going to be temporarily unavailable. With this constructor, you specify how many seconds the servlet is going to be unavailable, as follows:

```
throw new UnavailableException(120, this, "Try back in two minutes");
```

One caveat: the servlet specification does not mandate that servers actually try again after the specified interval. If you choose to rely on this capability, you should test it first with the container you plan to deploy on.

3.5.2.3. A file-serving servlet

Example 3-2 demonstrates both of these error-handling techniques, along with another method for reading data from the server. `FileServlet` reads a pathname from a form parameter and returns the associated file. Note that this servlet is designed to return only HTML files. If the file can't be found, the servlet sends the browser a 404 error. If the servlet lacks sufficient access privileges to load the file, it sends an `UnavailableException` instead. Keep in mind that this servlet is a teaching exercise: you should not deploy it on your web server. (For one thing, any security exception renders the servlet permanently unavailable, and for another, it can serve files from the root of your hard drive.)

Example 3-2. Serving files

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

public class FileServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    File r;
    FileReader fr;
    BufferedReader br;
    try {
      r = new File(req.getParameter("filename"));
      fr = new FileReader(r);
      br = new BufferedReader(fr);
      if(!r.isFile(  )) {  // Must be a directory or something else
        resp.sendError(resp.SC_NOT_FOUND);
        return;
      }
    }
    catch (FileNotFoundException e) {
      resp.sendError(resp.SC_NOT_FOUND);
      return;
    }
    catch (SecurityException se) { // Be unavailable permanently
      throw(new UnavailableException(
        "Servlet lacks appropriate privileges."));
    }

    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter(  );
    String text;
    while( (text = br.readLine(  )) != null)
      out.println(text);

    br.close(  );
  }
}
```

3.6. Custom Servlet Initialization

At the beginning of this chapter, we talked about how a servlet's persistence can be used to build more efficient web applications. This is accomplished via class variables and the `init( )` method. When a server loads a servlet for the first time, it calls the servlet's `init( )` method and does not make any service calls until `init()` has finished. In the default implementation, `init( )` simply handles some basic housekeeping, but a servlet can override the method to perform whatever one-time tasks are required. This often means doing some sort of I/O-intensive resource creation, such as opening a database connection. You can also use the `init( )` method to create threads that perform various ongoing tasks. For instance, a servlet that monitors the status of machines on a network might create a separate thread to periodically ping each machine. When an actual request occurs, the service methods in the servlet can use the resources created in `init( )`. Thus, the status monitor servlet might display an HTML table with the status of the various machines.

The default `init(ServletConfig)` implementation is not a do-nothing method, so you should remember to always call the `super.init(ServetConfig)` method. If you override the parameterless version, you don't have to invoke the superclass's methodthe server calls the parameterized version, which calls the parameterless method. You can access the `ServletConfig` object using the `getServletConfig()` method.

The server passes the `init( )` method a `ServletConfig` object, which can include specific servlet configuration parameters (for instance, the list of machines to monitor). `ServletConfig` encapsulates the servlet initialization parameters, which are accessed via the `getInitParameter( )` and `getInitParameterNames( )` methods. `GenericServlet` and `HttpServlet` both implement the `ServletConfig` interface, so these methods are always available in a servlet and they access the servlet's `ServletConfig` to retrieve these parameters. Init parameters are set using the `<init-param>` elements for the `<servlet>` in the *web.xml* deployment descriptor (see

Every servlet also has a `destroy( )` method that can be overwritten. This method is called when, for whatever reason, a server unloads a servlet. You can use this method to ensure that important resources are freed or that threads are allowed to finish executing unmolested. Unlike `init( )`, the default implementation of `destroy( )` is a do-nothing method, so you don't have to worry about invoking the superclass's `destroy( )` method.

Example 3-3 shows a counter servlet that saves its state between server shutdowns. It uses the `init( )` method to first try to load a default value from a servlet initialization parameter. Next the `init( )` method tries to open a file named /data/counter.dat and read an integer from it. When the servlet is shut down, the `destroy( )` method creates a new counter.dat file with the current hit count for the servlet.

Example 3-3. A persistent counter servlet

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

public class LifeCycleServlet extends HttpServlet {

  int timesAccessed;

  public void init(ServletConfig conf) throws ServletException {

    super.init(conf);

    // Get initial value
    try {
      timesAccessed = Integer.parseInt(getInitParameter("defaultStart"));
    }
    catch(NullPointerException e) {
      timesAccessed = 0;
    }
    catch(NumberFormatException e) {
      timesAccessed = 0;
    }

    // Try loading from the disk
    DataInputStream ds = null;
    try {
      File r = new File("./data/counter.dat");
      DataInputStream ds = new DataInputStream(new FileInputStream(r));
      timesAccessed = ds.readInt(  );
    }
    catch (FileNotFoundException e) {
      // Handle error
    }
    catch (IOException e) {
      // This should be logged
    }
    finally {
      if (ds !=null)
         try {
             ds.close(  );
         } catch (IOException ignored) {}
  }

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
```

```
    throws ServletException, IOException {

    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter(  );

    timesAccessed++;

    out.println("<html>");
    out.println("<head>");
    out.println("<title>Life Cycle Servlet</title>");
    out.println("</head><body>");

    out.println("I have been accessed " + timesAccessed + " time[s]");
    out.println("</body></html>");
  }

  public void destroy(  ) {

    // Write the Integer to a file
    File r = new File("./data/counter.dat");
    DataOutputStream dout = null;
    try {
      dout = new DataOutputStream(new FileOutputStream(r));
      dout.writeInt(timesAccessed);
    }
    catch(IOException e) {
      // This should be logged
    }
    finally {
      try dout.close(  ); catch (IOException ignored) {}
    }
  }
}
```

3.6.1. Servlet Context Initialization

Version 2.3 of the Servlet API added support for application-level events using a listener-style interface. Classes that implement the `ServletContextListener` interface can be associated with a servlet context and are notified when the context is initialized or destroyed. This provides programmers with the opportunity to create application-level resources, such as database connection pools, before any servlets are initialized and to share single resources among multiple servlets using the `ServletContext` attribute functionality.

`ServletContextListener` contains two methods, `contextInitialized( )` and `contextDestroyed ( )`, which take a `ServletContextEvent`. Context listeners are associated with their context in the web.xml file for the web application (see Appendix A for details on configuring contexts in the *web.xml* file). Example 3-4 defines a listener that creates a hashtable of usernames and unencrypted passwords and associates it as a context attribute. We use it in a later example.

Example 3-4. A servlet context listener

```
import javax.servlet.ServletContextListener;
import javax.servlet.ServletContextEvent;

public class ContextResourceLoader implements ServletContextListener {
  public void contextInitialized(ServletContextEvent sce) {
    java.util.Hashtable users = new Hashtable(  );
    users.put("test", "test");
    users.put("admin", "bob3jk");
    sce.getServletContext(  ).setAttribute("enterprise.users", users);
  }

  public void contextDestroyed(ServletContextEvent sce) {
     // This is where we clean up resources on server shutdown/restart
  }
}
```

Obviously, a real application would manage usernames and passwords in a more robust fashion, such as by storing them in a database. In this case, we can count on the JVM to properly garbage-collect the `Hashtable` object. If we do something more complex (such as maintaining a pool of connections to a relational database), we would use the `contextDestroyed(  )` method to make sure those resources are properly freed.

3.7. Security

Servlets don't have to handle their own security arrangements. Instead, they can rely on the capabilities of the web server to limit access where required. The security capabilities of most web servers are limited to basic on-or-off access to specific resources, controlled by username and password (or digital certificate), with possible encryption using SSL. Most servers are limited to basic authentication, which transmits passwords more or less in the clear, while some support the more advanced digest authentication protocol, which works by transmitting a hash of the user's password and a server-generated value rather than the password itself. Both of these approaches look the same to the user; the familiar "Enter Username and Password" window pops up in the web browser.

Recent versions of the Servlet API take a much less hands-off approach to security. The web.xml file can define which servlets and resources are protected and which users have access. The user access model is the J2EE User-Role model, in which users can be assigned one or more roles. Users with a particular role are granted access to protected resources. A user named Admin might have both the Administrator role and the User role while users Bob and Ted might have only the User role. (See Chapter 10 for more details about J2EE security.)

In addition to basic, digest, and SSL authentication, the web application framework allows for HTML form-based logins. This approach allows the developer to specify an HTML or JSP page containing a form like the following:

```
<form method="post" action="j_security_check">
<input type="text" name="j_username">
<input type=password" name="j_password">
<input type="submit" value="Log In">
</form>
```

Note that form-based authentication is insecure and works only if the client session is being tracked via cookies or SSL signatures.

The `HttpServletRequest` interface includes a pair of basic methods for retrieving standard HTTP user authentication information from the web server. If your web server is equipped to limit access, a servlet can retrieve the username with `getremoteUser( )` and the authentication method (basic, digest, or SSL) with `getAuthType( )`. Version 2.2 of the Servlet API added the `isUserInRole( )` and `getUserPrincipal( )` methods to `HttpServletRequest`. `isUserInRole( )` allows the program to query whether the current user has a particular role (useful for dynamic content decisions that cannot be made at the container level). The `getUserPrincipal( )` method returns a `java.security.Principal` object identifying the current user.

3.8. Servlet Filters

Version 2.3 of the Servlet API introduced a new method of handling requests using the `javax.servlet.Filter` class. When filters are used, the servlet container creates a filter chain, which consists of zero or more `Filter` objects and a destination resource, either a servlet or another resource available on the web server (such as an HTML or JSP file).

Filters are installed in the server and associated with particular request paths (just like servlets). When a filtered resource is requested, the servlet constructs a filter chain and calls the `doFilter( )` method of the first filter in the filter chain, passing a `ServletRequest`, a `ServletResponse`, and the `FilterChain` object. The filter can then perform processing on the request. Filters are often used to implement logging, control security, or set up connection-specific objects. A filter can also wrap the `ServletRequest` and `ServletResponse` classes with its own versions, overriding particular methods. For instance, one of the example filters included with the Tomcat server adds support for returning compressed output to browsers that support it.

After the filter has processed the response, it can call the `doFilter( )` method of the `FilterChain` to invoke the next filter in the sequence. If there are no more filters, the request is passed on to its ultimate destination. After calling `doFilter()`, the filter can perform additional processing on the response received from farther down the chain.

In the event of an error, the filter can stop processing, returning to the client whatever response has already been created or forwarding the request to a different resource.

Example 3-5 provides a form-based authentication filter that could be customized to provide additional functionality. It works by intercepting each request and checking the `HttpSession` for an attribute called `enterprise.login`. If that attribute contains a `Boolean.TRUE`, access is permitted. If not, the filter checks for request parameters named `login_name` and `login_pass` and searches for a match in a hashtable containing valid username/password pairs. If valid login credentials are found, filter chain processing continues. If not, the user is served a login page located at /login.jsp, retrieved via a `RequestDispatcher`.[*]

[*] This isn't a highly secure system. Unless the client has connected via SSL, the username/password combination is transmitted unencrypted over the Internet. Also, successful logins leave the `login_name` and `login_pass` parameters in the request when processing it, potentially making them available to a malicious JSP file or servlet. This can

be an issue when designing a shared security scheme for dynamic content created by a group of different users (such as at an ISP). One way to get around this is to create a custom `HttpServletRequest` wrapper that filters out the `login_name` and `login_pass` parameters for filters and resources further down the chain.

Astute readers will note that we try to retrieve the user's hashtable from a servlet context attribute. We showed how to set this attribute at the web application level earlier in the chapter. In case you don't have that set up, the filter's `init( )` method will create its own if it can't find one in the context.

Example 3-5. AuthenticationFilter

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Hashtable;

public class AuthenticationFilter implements Filter {

  private Hashtable users = null;
  public void init(FilterConfig config)
    throws javax.servlet.ServletException {

    users = (Hashtable)config.getServletContext( ).getAttribute(
                               "enterprise.users");
    if(users == null) {
       users = new Hashtable(5);
       users.put("test", "test");
    }
  }

  public void doFilter(
    ServletRequest req, ServletResponse res, FilterChain chain)
    throws java.io.IOException, javax.servlet.ServletException {

    HttpServletRequest request = (HttpServletRequest)req;
    HttpSession sess = request.getSession(true);

    if(sess != null) {
      Boolean loggedIn = (Boolean)sess.getAttribute("enterprise.login");
      if (loggedIn != Boolean.TRUE) {
        String login_name = request.getParameter("login_name");
        String login_pass = request.getParameter("login_pass");
        if((login_name != null) && (login_pass != null))
          if(users.get(login_name).toString( ).equals(login_pass)) {
            loggedIn = Boolean.TRUE;
            sess.setAttribute("enterprise.login", Boolean.TRUE);
            sess.setAttribute("enterprise.loginname", login_name);
          }
      }

      if (loggedIn == Boolean.TRUE) {
        chain.doFilter(req, res);
      }  else {
        request.setAttribute("originaluri", request.getRequestURI( ));
        request.getRequestDispatcher("/login.jsp").forward(req, res);
      }
    }
  }
  public void destroy(  ) {
    // Code cleanup would be here
  }
}
```

Here's the JSP page used to display the login form. The important thing to note is that the form submits back to the original URI. The filter uses the `setAttribute( )` method of `HttpServletRequest` to specify the URI to post the form back to; the filter is then reapplied, and if the user has provided appropriate credentials, access to the resource is granted. For more on JSP, see Chapter 4.

```
<html><body bgcolor="white">

<% out.print ("<form method=post
action=\""+request.getAttribute("originaluri").toString(  ) +"\">"); %>

Login Name: <input type=text name="login_name"><br>
Password: <input type=password name="login_pass">
<input type=submit value="Log In">
</form>

</body></html>
```

When configuring the filter (we'll see an example in the next section), map it only to the paths you wish to protect. Mapping it to /* will not work since that would also protect the login.jsp file. If you did want to protect your whole application, you could build the login form into the filter, but that's not good practiceyou have to recompile for every change.

3.8.1. Filters and Request Dispatchers

In the original filter specification, a filter would run only in response to an original request from a remote client. Requests created internally via the `RequestDispatcher` class would not trigger new filter chains. The example in the previous section relies on this behaviorotherwise the filter would never be able to display the login.jsp file.

Servlet API 2.4 gives us a little more flexibility here. When declaring a filter mapping in web.xml, we can indicate whether it runs on requests only (the default) or whether it runs on includes, forwards, or any combination of the three. For example:

```
<filter-mapping>
   <filter-name>Authentication Filter</filter-name>
   <url-pattern>/secrets/*</url-pattern>
   <dispatcher>FORWARD</dispatcher>
   <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

This configuration causes the filter to run on request forwards as well as request dispatches, ensuring that security is applied even if an unprotected portion of the application redirects the user to a protected portion. However, use this power with care: many applications, including the various web application frameworks, make extensive use of request forwarding, which could cause the filter to run several times in the course of a single user request.

3.9. Thread Safety

In a typical scenario, only one copy of any particular servlet or filter is loaded into the server's runtime at any given time. Each servlet might, however, be called upon to deal with multiple requests at the same time. This means that a servlet needs to be threadsafe. If a servlet doesn't use any class variables (that is, any variables with a scope broader than the service method itself), it is generally already threadsafe. If you are using any third-party libraries or extensions, make sure that those components are also threadsafe. However, a servlet that maintains persistent resources needs to make sure that nothing untoward happens to those resources. Imagine, for example, a servlet that maintains a bank balance using an `int` in memory.[*] If two servlets try to access the balance at the same time, you might get this sequence of events:

[*] Hey, bear with us on this. It's an example.

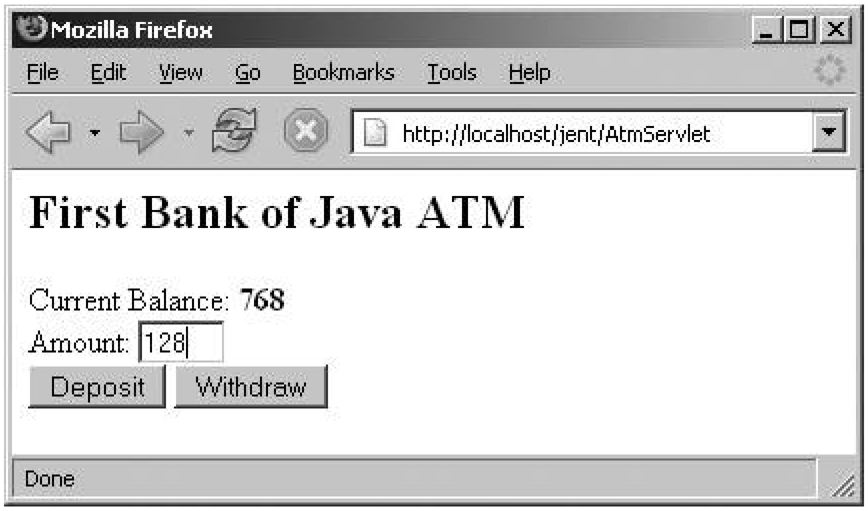
1. User 1 connects to the servlet to make a $100 withdrawal.
2. The servlet checks the balance for User 1, finding $120.
3. User 2 connects to the servlet to make a $50 withdrawal.
4. The servlet checks the balance for User 2, finding $120.
5. The servlet debits $100 for User 1, leaving $20.
6. The servlet debits $50 for User 2, leaving -$30.
7. The programmer is fired.

Obviously, this is incorrect behavior, particularly that last bit. We want the servlet to perform the necessary action for User 1, and then deal with User 2 (in this case, by giving him an insufficient funds message). We can do this by surrounding sections of code with synchronized blocks. While a particular synchronized block is executing, no other sections of code that are synchronized on the same object (usually the servlet or the resource being protected) can execute. For more information on thread safety and synchronization, see Java Threads by Scott Oaks and Henry Wong (O'Reilly).

Example 3-6 implements the ATM display for the First Bank of Java. The `doGet( )` method displays the current account balance and provides a small ATM control panel for making deposits and withdrawals, as shown in Figure 3-3.[]

[] Despite the fact that Java is a very large island, there's still only one account.

Figure 3-3. The First Bank of Java ATM display


Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/jent/AtmServlet

First Bank of Java ATM

Current Balance: **768**

Amount: 128

Deposit Withdraw

The control panel uses a POST request to send the transaction back to the servlet, which performs the appropriate action and calls `doGet( )` to redisplay the ATM screen with the updated balance.

Example 3-6. An ATM servlet

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

public class AtmServlet extends HttpServlet {

  Account act;

  public void init(ServletConfig conf) throws ServletException {
    super.init(conf);
    act = new Account(  );
    act.balance = 0;
  }

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    resp.setContentType("text/html");
    PrintWriter out = resp.getWriter(  );

    out.println("<html><body>");
    out.println("<h2>First Bank of Java ATM</h2>");
    out.println("Current Balance: <b>" + act.balance + "</b><br>");
    out.println("<form method=post>");
    out.println("Amount: <input type=text name=AMOUNT size=3><br>");
    out.println("<input type=submit name=DEPOSIT value=\"Deposit\">");
    out.println("<input type=submit name=WITHDRAW value=\"Withdraw\">");
    out.println("</form>");
    out.println("</body></html>");
  }

  public void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
```

```
    int amt=0;
    try {
      amt = Integer.parseInt(req.getParameter("AMOUNT"));
    }
    catch (NullPointerException e) {
      // No Amount parameter passed
    }
    catch (NumberFormatException e) {
      // Amount parameter was not a number
    }

    synchronized(act) {
      if((req.getParameter("WITHDRAW") != null) && (amt < act.balance))
        act.balance = act.balance - amt;
      if((req.getParameter("DEPOSIT") != null) && (amt > 0))
        act.balance = act.balance + amt;
    } // End synchronized block

    doGet(req, resp);                   // Show ATM screen
  }

  public void destroy(  ) {
    // This is where we would save the balance to a file
  }

  class Account {
    public int balance;
  }
}
```

The `doPost( )` method alters the account balance contained within an `Account` object `act` (since `Account` is so simple, we've defined it as an inner class). In order to prevent multiple requests from accessing the same account at once, any code that alters `act` is synchronized on `act`. This ensures that no other code can alter `act` while a synchronized section is running.

The `destroy( )` method is defined in the `AtmServlet`, but it contains no actual code. A real banking servlet would obviously want to write the account balance to disk before being unloaded. And if the servlet were using JDBC to store the balance in a database, it would also want to destroy all its database-related objects.

Servlet API 2.4 deprecates the `SingleThreadModel` interface, which was a tag interface to identify a servlet that could serve only one request at a time. Since `SingleThreadModel` could not resolve all potential threading issues, it has been deprecated and should not be used in new development.

3.10. Cookies

Very few software features have caused as much public confusion and outcry as the HTTP cookie.

Ethical and moral considerations aside, cookies allow a web server to store small amounts of data on client systems. Cookies are generally used to store basic user identification or configuration information. Because a cookie's value can uniquely identify a client, cookies are often used for session tracking (although, as we'll see shortly, the Servlet API provides higher-level support for this).[*]

[*] The cookie standard is spelled out in RFC 2965, available at http://rfc.net/rfc2965.html.

To create a cookie, the server (or, more precisely, a web application running on the server) includes a `Cookie` header with a specific value in an HTTP response. The browser then transmits a similar header with that value back to the server with subsequent requests, which are subject to certain rules. The web application can use the cookie value to keep track of a particular user, handle session tracking, and so forth. Because cookies use a single `Cookie` header, the syntax for a cookie allows for multiple name/value pairs in the overall cookie value.

The Servlet API includes a class, `javax.servlet.http.Cookie`, that abstracts cookie syntax and makes cookies easy to work with. In addition, `HttpServletResponse` provides an `addCookie( )` method and `HttpServletRequest` provides a `getCookies( )` method to aid in writing cookies to and reading cookies from the HTTP headers, respectively. To find a particular cookie, a servlet needs to read the entire collection of values and look through it:

```
Cookie[] cookies;
cookies = req.getCookies(  );
String userid = null;

for (int i = 0; i < cookies.length; i++)
  if (cookies[i].getName(  ).equals("userid"))
    userid = cookies[i].getAttribute(  );
```

A cookie can be read at any time but can be created only before any content is sent to the client. This is because cookies are sent using HTTP headers. These headers can be sent to the client before the regular content. Once any content has been written to the client, the server can flush the output and send the headers at any time, so you can't create any new cookies safely. You must create new cookies before sending any content. Here's an example of creating a cookie:

```
String userid = createUserID(  );     // Create a unique ID
Cookie c = new Cookie("userid", userid);
resp.addCookie(c);                  // Add the cookie to the HTTP headers
```

Note that a web browser is required to accept only 20 cookies per site and 300 total per user, and the browser can limit each cookie's size to 4096 bytes.

Cookies can be customized to return information only in specific circumstances. In particular, a cookie can specify a particular domain, a particular path, an age after which the cookie should be destroyed, and whether the cookie requires a secure (HTTPS) connection. A cookie is normally returned only to the host that specified it. For example, if a cookie is set by server1.company.com, it isn't returned to server2.company.com. You can get around this limitation by setting the domain to .company.com with the `setDomain( )` method of `Cookie`. By the same token, a cookie is generally returned for pages only in the same directory as the servlet that created the cookie, or it's returned under that directory. We can get around this limitation using `setPath( )`. Here's a cookie that is returned to all pages on all top-level servers at company.com:

```
String userid = createUserID(  );   // Create a unique ID
Cookie c = new Cookie("userid", userid);
c.setDomain(".company.com");  // *.company.com, but not *.web.company.com
c.setPath("/");               // All pages
resp.addCookie(c);            // Add the cookie to the HTTP headers
```

3.11. Session Tracking

Very few web applications are confined to a single page, so having a mechanism for tracking users through a site can often simplify application development. The Web, however, is an inherently stateless environment. A client makes a request, the server fulfills it, and both promptly forget about each other. In the past, applications that needed to deal with a user through multiple pages (for instance, a shopping cart) had to resort to complicated dodges to hold onto state information, such as hidden fields in forms, setting and reading cookies, or rewriting URLs to contain state information.

The Servlet API provides classes and methods specifically designed to handle session tracking . A servlet can use the session-tracking API to delegate most of the user-tracking functions to the server. The first time a user connects to a session-enabled servlet, the servlet simply creates a `javax.servlet.http.HttpSession` object. The servlet can then bind data to this object, so subsequent requests can read the data. After a certain amount of inactive time, the session object is destroyed.

A servlet uses the `getSession( )` method of `HttpServletRequest` to retrieve the current session object. This method takes a single `boolean` argument. If you pass `TRue` and there is no current session object, the method creates and returns a new `HttpSession` object. If you pass `false`, the method returns `null` if there is no current session object. For example:

```
HttpSession thisUser = req.getSession(true);
```

When a new `HttpSession` is created, the server assigns a unique session ID that must somehow be associated with the client. Since clients differ in what they support, the server has a few options that vary slightly depending on the server implementation. In general, the server's first choice is to try to set a cookie on the client (which means that `getSession( )` must be called before you write any other data back to the client). If cookie support is lacking, the API allows servlets to rewrite internal links to include the session ID, using the `encodeURL( )` method of `HttpServletResponse`. This is optional, but recommended, particularly if your servlets share a system with other, unknown servlets that may rely on uninterrupted session tracking. However, this on-the-fly URL encoding can become a performance bottleneck because the server needs to perform additional parsing on each incoming request to determine the correct session key from the URL. (The performance hit is so significant that many servers disable URL encoding by default.)

To use URL encoding, run all your internal links through `encodeURL( )`. If you have a line of code like this:

```
out.println("<a href=\"/servlet/CheckoutServlet\">Check Out</a>");
```

you should replace it with:

```
out.print("<a href=\"");
out.print(resp.encodeURL("/servlet/CheckoutServlet");
out.println("\">Check Out</a>");
```

When cookies can't be used, the server will encode the session ID into the path by appending `;jsessionid=` and the session ID, like this:[*]

> [*] Earlier versions of the Servlet API left the session encoding algorithm up to the container developer. Later releases standardized the system to prevent portability problems for applications that also use path encoding.

```
<a href="/app/CheckoutServlet;jsessionid=12AB39J">Check Out</a>"
```

In addition to encoding your internal links, you need to use `encodeRedirectURL( )` to handle redirects properly. This method works in the same manner as `encodeURL( )`.[ ]

> [ ] These methods were introduced in Servlet API 2.1, replacing two earlier methods named `encodeUrl( )` and `encodeRedirectUrl( )`. This was done to bring the capitalization scheme in line with other Java APIs.

You can access the unique session ID via the `getID( )` method of `HttpSession`. This is enough for most applications, since a servlet can use some other storage mechanism (i.e., a flat file, memory, or a database) to store the unique information (e.g., hit count or shopping cart contents) associated with each session. However, the API makes it even easier to hold onto session-specific information by allowing servlets to bind objects to a session using the `setAttribute( )` method of `HttpSession`. Once an object is bound to a session, you can use the `getAttribute()` method to retrieve it.

Objects bound using `setAttribute( )` are available to all servlets running on the server. The system works by assigning a user-defined name to each object (the `String` argument); this name is used to identify objects at retrieval time. In order to avoid conflicts, the general practice is to name bound objects with names of the form *`applicationname.objectname`*. For example:

```
session.setAttribute("myservlet.hitcount", new Integer(34));
```

Now that object can be retrieved with:

```
Integer hits = (Integer)session.getAttribute("myservlet.hitcount")
```

Example 3-7 demonstrates a basic session-tracking application that keeps track of the number of visits to the site by a particular user. It works by storing a counter value in an `HttpSession` object and incrementing it as necessary. When a new session is created (as indicated by `isNew( )`, which returns `true` if the session ID has not yet passed through the client and back to the server) or the counter object is not found, a new counter object is created.

Example 3-7. Counting visits with sessions

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

public class VisitCounterServlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    PrintWriter out = resp.getWriter(  );
    resp.setContentType("text/html");
    HttpSession thisUser = req.getSession(true);
    Integer visits;

    if(!thisUser.isNew(  )) {          // Don't check newly created sessions
      visits = (Integer)thisUser.getAttribute("visitcounter.visits");
      if(visits == null)
        visits = new Integer(1);
      else
        visits = new Integer(visits.intValue(  ) + 1);
    }
    else
      visits = new Integer(1);

    // Put the new count in the session
    thisUser.setAttribute("visitcounter.visits", visits);

    // Finally, display the results and give them the session ID too
    out.println("<html><head><title>Visit Counter</title></head>");
    out.println("<body>You have visited this page " + visits + " time[s]");
    out.println("since your last session expired.");
    out.println("Your Session ID is " + thisUser.getId(  ));
    out.println("</body></html>");
  }
}
```

3.11.1. HttpSessionBindingListener

Sometimes it is useful to know when an object is being bound to or unbound from a session object. For instance, in an application that binds a JDBC `java.sql.Connection` object to a session (something that, by the way, is ill-advised in all but very low traffic sites), it is vitally important that the `Connection` be explicitly closed when the session is destroyed.

The `javax.servlet.http.HttpSessionBindingListener` interface handles this task. It includes two methods, `valueBound( )` and `valueUnbound( )`, that are called whenever the object that implements the interface is bound or unbound from a session, respectively. Each of these methods receives an `HttpSessionBindingEvent` object that provides the name of the object being bound or unbound and the session involved in the action. The following is an object that implements the

`HttpSessionBindingListener` interface in order to make sure that a database connection is closed properly:

```
class ConnectionHolder implements HttpSessionBindingListener {

  java.sql.Connection dbCon;

  public ConnectionHolder(java.sql.Connection con) {
    dbCon = con;
  }

  public void valueBound(HttpSessionBindingEvent event) {
    // Do nothing
  }

  public void valueUnbound(HttpSessionBindingEvent event) {
    dbCon.close(  );
  }
}
```

Programmers should also be aware of the `HttpSessionListener` interface, which responds to session creation and destruction events.

3.12. Databases and Non-HTML Content

Most web applications need to communicate with a database, either to generate dynamic content or collect and store data from users, or both. With servlets, this communication is easily handled using the JDBC API described in Chapter 8. Thanks to JDBC and the generally sensible design of the servlet lifecycle, servlets are an excellent intermediary between a database and web clients.

Most of the general JDBC principles discussed in Chapter 8 apply to servlets. In our example, we create a database connection within the servlet's `init(  )` method. Larger applications will generally prefer to use a database connection pool, managed by the servlet container. Connection pools and `DataSource` objects (for container-managed database connections) are discussed in Chapter 8. Another option would be to use the object relational capabilities provided by Hibernate in your servlet. Hibernate is discussed in Chapter 20.

So far, all our servlets have produced standard HTML content. Of course, this is all most servlets ever do, but it's not all that they can do. Say, for instance, that your company stores a large database of PDF documents within an Oracle database, where they can be easily accessed. Now say you want to distribute these documents on the Web. Luckily, servlets can dish out any form of content that can be defined with a MIME header. All you have to do is set the appropriate content type and use a `ServletOuputStream` if you need to transmit binary data. Example 3-8 shows how to pull Adobe Acrobat documents from an Oracle database.

Example 3-8. A servlet that serves PDF files from a database

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

public class DBPDFReader extends HttpServlet {

  Connection con;

  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    try {
      Class.forName("oracle.jdbc.driver.OracleDriver");
      con = DriverManager.getConnection("jdbc:oracle:thin:@DBHOST:1521:ORCL",
                                        "user", "passwd");
    }
    catch (ClassNotFoundException e) {
      throw new UnavailableException("Couldn't load OracleDriver");
    }
    catch (SQLException e) {
      throw new UnavailableException("Couldn't get db connection");
    }
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    try {
      res.setContentType("application/pdf");
      ServletOutputStream out = res.getOutputStream(  );

      Statement stmt = con.createStatement(  );

      /* This is dangerous for production code. Request
         parameters should never be injected into SQL without
         preprocessing. We're keeping syntax simple for this
         example. */
      ResultSet rs = stmt.executeQuery(
        "SELECT PDF FROM PDF WHERE PDFID = " + req.getParameter("PDFID"));

      if (rs.next(  )) {
        BufferedInputStream pdfData =
          new BufferedInputStream(rs.getBinaryStream("PDF"));
        byte[] buf = new byte[4 * 1024];  // 4K buffer
        int len;
        while ((len = pdfData.read(buf, 0, buf.length)) != -1) {
          out.write(buf, 0, len);
        }
      }
      else {
        res.sendError(res.SC_NOT_FOUND);
      }
      rs.close(  );
      stmt.close (  );
    }
    catch(SQLException e) {
      // Report it
    }
  }
}
```