chapter 3

Advanced Servlet Techniques

In This Chapter

- Using sessions and storing state
- Using cookies for long-term storage
- Filtering HTTP requests
- Understanding WebLogic Server deployment issues
- Additional best practices for Web application development

Chapter 2 discusses the basics of handling HTTP requests and how to manage collections of Web resources. This chapter takes servlets a step further and discusses how to maintain information between HTTP requests, how to improve code reuse with filters, and how to plan Web application deployments on WebLogic to improve performance and reliability.

Servlets and Web Sessions

The Hypertext Transfer Protocol (HTTP) is, by design, a stateless protocol. However, many Web applications require information, previously gathered from the user, to process a request. With this information, a servlet can remember a user's name, items placed in a shopping cart, or a shipping address. The notion of saving information across multiple requests from a single client is typically called a Web session. Linking this information from one request to another is session tracking.

Store Information in a Session

In the early days of the Web, developers found many ways to maintain session information. Browsers and servlet containers provided different mechanisms to store, retrieve, and track information from one request to the next. Developers had to create their own session management schemes. Fortunately, the servlet standard defines a common mechanism to maintain sessions. This common mechanism, the servlet session, hides most of the technical complexities and allows developers to concentrate on writing Web applications. Servlets place information into a session and retrieve this information in subsequent requests. In this way, sessions add state information to the otherwise stateless HTTP protocol.

Sessions work by storing state information in a local object and creating a reference to that object. WebLogic Server passes this reference back to the browser to be returned in future requests. When the browser does make another request, it includes the reference. WebLogic Server matches the request to the appropriate session object using this reference and provides this session object to the Web application. This way, when a the servlet stores a user's account number in the session, the account number is still there when the user clicks the B*uy now* button.

The servlet session hides most, but not all, technical details. First, WebLogic Server must store the session object between requests. WebLogic Server offers a number of options to store this information, each with its own set of tradeoffs. The section on deployment considerations, later in this chapter, provides more options regarding session storage.

Second, WebLogic Server must pass the reference or session ID to the browser. Typically, WebLogic Server automatically selects between two mechanisms: URL rewriting and HTTP cookies. Again, each technique has tradeoffs. Developers can configure both the selection between these mechanisms and implementation details for either mechanism. In practice, the configuration of the client typically determines the selection between URL rewriting and cookies.

The class `javax.servlet.http.HttpSession` defines servlet sessions. You can access the `HttpSession` using the `getSession()` method of the `HttpServletRequest`. You can also set and get attributes on the `HttpSession` object. The `HttpSession` enables your application to maintain session information without dealing with all the details of tracking sessions.

What to Put in Your Session

There are a number of things you could store in a session object, including the following:

- Fields a user has entered that will be used to generate a subsequent page.
- Virtual shopping carts to hold the items that a user is currently interested in purchasing.
- A history of resources a user has viewed during the current session. For example, if you are building an e-commerce site, you'll want to see what pages the user has visited so you can decide what other pages might be of interest to him or her.

There also are things that you should not store in a session object. Do not store long-term data, such as a user record or account balance. Also, don't store transactional data in a session. Sessions can never provide the reliability necessary to record transactions. The session is a temporary object, so only store information that you can recover.

Session ID

WebLogic Server generates a long, randomly generated, and unique session identification number for each session. The servlet engine uses this number to track sessions across HTTP requests.

For each new browser session, WebLogic Server automatically creates a temporary HTTP cookie, a text object that can be placed on the client's machine and included in future requests to the server. WebLogic Server automatically assigns a session ID number, stores the session identification number in this temporary cookie, and attempts to place the cookie in the client browser. If successful, the browser sends this cookie back to the server along with each subsequent request. WebLogic Server automatically maps the session identifier to the `HttpSession` object it is tracking for that user. WebLogic Server locates the appropriate `HttpSession` and provides this session object to the servlet within the `HttpRequest` object.

Some clients do not support cookies, and users may consider cookies an invasion of privacy and reject them. If WebLogic Server fails to place a cookie on the client, it reverts to URL rewriting. URL rewriting works by adding the session ID to each URL in subsequent requests.

When storing session information, security is a concern. Even though the session information is maintained with the server, we are concerned with the session ID stored on the client. If the session ID is short, an attacker may be able to guess a valid session ID and masquerade as a valid user. The longer the session ID, the harder it is for an attacker to guess. By default, WebLogic Server generates a 52-digit session ID.

For most situations, the default session ID length is acceptable. In other cases, shorter session IDs are required. For example, some wireless devices (using WAP, or the Wireless Application Protocol) limit the entire URL to 128 characters. In this situation, use the `weblogic.xml` deployment descriptor to set the length of the session ID. The best practice is to use the longest session ID possible to ensure security.

Using Sessions

Accessing the Session Object

From one request to the next, the session object is available for the lifetime of the session. Access the `HttpSession` object from the `HttpServletRequest` object using the `getSession()` method, like this:

```
HttpSession session = request.getSession (false);
```

The method takes a `boolean` parameter to specify whether the method may create a new session. If the value is true, WebLogic creates a new session object if one does not already exist. If the value is false, the server does not create a new session object but instead returns `null`. For reasons that become apparent later, you typically want to set a value of false when you call this method.

Creating New Sessions

The `HttpSession` object provides an `isNew()` method—use it with caution. The best way to determine whether your user already has a session is to get the current session. If no session exists, you can force the user to log in and, only after a successful login, create a new session object. This sequence is a good practice because it prevents users from bypassing your security mechanisms.

Acquire the session object by calling the method `getSession(false)`. If the user does not already have a session, the method returns `null`. You can then force users to log in each time they begin a new session. In this case, redirect the client browser to the login page for your Web application when the `getSession(false)` returns `null`. If the session exists, or is not `null`, you can continue using the session:

```
HttpSession session = request.getSession (false);

if (session==null) {
    // We  Send a Redirect
    responseObj.sendRedirect("http://www.learnweblogic.com/login");
}
// continue processing
```

What NOT *to* Do

You might be tempted to do the following:

```
// An Example of What Not to Do:
HttpSession session = request.getSession (true);
// Check to See If the Session Is New
if (session.isNew()) {
    // We  Send a Redirect
    responseObj.sendRedirect("http://www.learnweblogic.com/login");
}
```

Don't do it.

In this code, we ask for the session object as usual. However, because we set the parameter value to `true` in the `getSession()` method, a new session is created if one does not already exist. While this code seems to do the right thing, it introduces a security problem. Someone can create a large number of session objects by repeatedly requesting the URL. These sessions accumulate as memory on your server for each request, so an attacker can tie up significant server resources. Ultimately, an attacker can destabilize the server.

Best Practice: Use sessions to mark login status. Create a new session only on login and look for a session to confirm login status. When the user logs out, invalidate his or her session. Use filters to apply security checks consistently throughout your Web application.

Storing and Accessing Session Data

Session data is stored in the `HttpSession` object using name/value pairs. For each pair, a `String` name is used to reference a value `Object`. The value can be an instance of any Java class.

The servlet specification defines four methods that you can use to access the values in the `HttpSession` object:

- `public Object getAttribute (String name)`—Returns the object bound with the specified name in this session, or `null` if no object is bound under the name.
- `public void setAttribute (String name, Object attribute)`—Binds an object to this session using the name specified.
- `public Enumeration getAttributeNames()`—Returns an Enumeration of string objects containing the names of all the objects bound to this session.
- `public void removeAttribute(String name)`—Removes the object bound with the specified name from this session.

Best Practice: Keep your sessions small. Your servlet's performance can degrade if the session objects become too large. Keep your attributes small. When an attribute is changed, WebLogic Server must replicate the attribute. If your attributes are small, you minimize the required replication.

Best Practice: Make sure all attribute objects implement the `Serializable` interface. WebLogic Server may serialize your session to move it to another instance.

URL Rewriting

WebLogic Server automatically generates temporary cookies to store the session ID, but what if a client does not accept cookies? URL rewriting provides a way to deal with these clients. URL rewriting works by modifying the HTML code sent to the client. WebLogic Server adds a session ID to the end of each URL on links back to your application. WebLogic Server can use this session ID in place of cookies to match the request to the right session.

URL rewriting requires that the servlet call an encoding method in the servlet response object for URLs in the generated HTML page. For URL rewriting to work reliably, all servlets in a Web application must use rewriting on all URLs. The methods to rewrite URLs take two forms:

- `public String encodeURL(String url)`—Includes the logic required to encode the session information in the returned URL. WebLogic Server automatically determines whether the session ID needs to be encoded in the URL. If the browser supports cookies, or if session tracking is turned off, URL encoding is unnecessary. To encode the URL, add the following code to your servlet:

  ```
  // Add the Session Information Via URL Encoding
  myHttpServletResponse.encodeURL(thisURL);
  ```

  So, if your original URL was

  ```
  <a href="foo.jsp">bar</a>
  ```

  you would use the `encodeURL()` method on that URL:

  ```
  out.println("<a href=\"" +
  response.encodeURL("<a href=\"foo.jsp\">") + "\">bar</a>");
  ```

- `public String encodeRedirectURL(String url)`—Performs the same task as the previous method, except it is specially geared to redirects in the response object. For redirection, you should have the following in your servlet:

  ```
  // Sending a Redirect with URL Encoded session ID
  myHttpServletResponse.sendRedirect
          (myHttpServletResponse.encodeRedirectUrl(anotherURL));
  ```

URL rewriting is also important when using Web frames and redirects. Depending on timing and the type of request, the browser may not send the cookie with every request. If you notice that your Web site mysteriously loses contact with the user sessions when frames are being used, you should force URL rewriting to solve the problem. See the section "Configure Sessions" in this chapter for specifics on forcing URL rewriting.

At the beginning of a session, WebLogic Server uses URL rewriting while it determines whether the client supports cookies. If the client browser does support cookies, then WebLogic Server stops rewriting URLs. If not, WebLogic Server continues to use URL rewriting.

By default, WebLogic uses both cookies and URL rewriting. To override this default behavior, set session parameters in the `weblogic.xml` deployment descriptor for the Web application. See the section "Configure Sessions" to set `CookiesEnabled` and `URLRewritingEnabled` session parameters.

Best Practice: Always rewrite URLs in your servlets and JSPs for links back to the Web application.

Testing URL R*ewriting*

Once you activate URL rewriting, the clients that you point at your server see a difference. In fact, you see something like this:

```
http://www.shinn.com/index.html;jsessionid=1234!a!b
```

The session ID number is encoded at the end of the URL for each page. You may notice that the session ID appears to be longer than specified. In the unrealistic example above, the session ID is four characters long. The values following this ID are used by WebLogic Server to support clustering.

Lifecycle of a Session

Servlets create new a new session object using `getSession(true)`. Following this call, WebLogic Server creates an HTTP session object and begins to pass the session ID to the browser. WebLogic Server passes this session ID to the Web browser as long as the session is active and WebLogic Server can match the request to the right session. The session remains active until it expires from lack of use or the servlet explicitly deactivates the session. To match the request to an active session, WebLogic Server requires the session ID. Under most circumstance, the browser continues to pass this ID until the browser is restarted.

Session Duration

If the client does not make any requests to the WebLogic Server for a specified period of time, WebLogic Server automatically invalidates and removes the HTTP session. By default, WebLogic Server times sessions out after one hour. You may set a timeout interval in the WebLogic Server configuration or in servlet code. If the user requests a page after WebLogic Server has closed the session, then the user has to log in again. After a session times out, any information you have stored in the HTTP session object is lost.

WebLogic Server discards timed-out session objects, but also the session is lost if the client loses its reference to the session. For instance, if the user shuts down the Web browser, the browser deletes the cookies storing the session ID. If the user restarts the browser, he must log in again. If URL rewriting is active and the user leaves the site, the HTML containing the session ID is lost. When a client loses its session ID, WebLogic Server cannot match the HTTP request with the `HttpSession` object. The user must log in, and the old session object on the server will eventually time out. As we see in the next section, the servlet may also explicitly end a session.

Finally, the `HttpSession` objects are managed by the WebLogic Server and should not be referenced outside the scope of an HTTP request (after the `service()` or `do<request type>()` method returns). Instead, reference the attribute objects stored in the session. If you must reference the `HttpSession` outside the scope of the request, then `clone()` the session first, and pass the copy.

Invalidating a Session

Under certain circumstances, such as logging out, you should explicitly invalidate a session. Use the `invalidate()` method of the `HttpSession` object, like this:

```
// Create the Session Object
HttpSession session = request.getSession (false);
// Invalidate the Session
session.invalidate();
```

WebLogic Server clears the HTTP session and invalidates it for further use. The next time the user visits your servlet, no session will be available. Your Web application may require the user to log in again and create a new session object.

Sessions, Inactivity, and Time

The `HttpSession` object supports other useful methods:

- `public long getCreationTime()`—Returns a long integer that represents when the session was created. To print the creation time of the object,

```
DateFormat df =
DateFormat.getDateTimeInstance(DateFormat.DEFAULT,DateFormat.FULL);
HttpSession session = request.getSession (true);
// Get the Creation Time and Print it in human-readable form
System.out.println(df.format(new Date(session.getCreationTime())));
```

- `public String getId()`—Returns a long integer of the ID number that WebLogic Server has associated with the session. To print the ID number of the servlet session,

```
HttpSession session = request.getSession (true);
// Get the Creation Time and Print it
system.out.println(session.getID());
```

- `public long getLastAccessedTime()`—Returns a long integer that represents the last time the session object was accessed. If the session is new, the last accessed time is the same as the creation time.
- `public int getMaxInactiveInterval()`—Returns an integer that represents the number of seconds that the session can be inactive before it is automatically removed by WebLogic Server.
- `public void setMaxInactiveInterval(int interval)`—Sets the number of seconds that the session can be inactive before the session is invalidated.

Configure Sessions

Use the Web application's deployment descriptor files to configure WebLogic Server management of sessions. The standard deployment descriptor, `web.xml`, provides a single configuration parameter: `session-timeout`. The following lines from a `web.xml` file specify a 5-minute timeout for sessions.

```
    <session-config>
        <session-timeout>5</session-timeout>
    </session-config>
```

Specify the `session-timeout` in minutes. A value of 0 or –1 means that the session never times out. Do not use this setting: over time, these permanent sessions will consume significant resources and overload the system. If you need information to persist beyond a user session, use cookies or use session lifecycle events to trigger permanent storage of the session information (both approaches are described later in this chapter). The value –2 instructs WebLogic Server to look to the *weblogic.xml* file for a timeout setting; this behavior is the default.

Use the WebLogic-specific deployment descriptor, `weblogic.xml`, to achieve greater control over session management. The `session-descriptor` element may contain several `session-param` elements; each `session-param` defines a parameter using a name/value pair. In the following example, the length of the session ID is shortened, `CookiesEnabled` is set to false, and `URLRewritingEnabled` is set to true. As a result, WebLogic Server generates a short session ID and uses only URL rewriting.

```
<session-descriptor>
    <session-param>
        <param-name>IDLength</param-name>
        <param-value>16</param-value>
    </session-param>
    <session-param>
        <param-name>CookiesEnabled</param-name>
        <param-value>false</param-value>
    </session-param>
    <session-param>
        <param-name>URLRewritingEnabled</param-name>
        <param-value>true</param-value>
    </session-param>
</session-descriptor>
```

Use the following parameters to configure session management for WebLogic Server:

- `TrackingEnabled`—Use this setting to turn on or off session tracking in WebLogic Server. If tracking is turned off, then WebLogic Server creates the session but does not save the session between requests. [`true`|`false`; default `true`].
- `TimeoutSecs`—This parameter specifies a default for how long a session remains active. WebLogic considers a session invalid if it remains inactive for the specified number of seconds. Your servlet code can override this value programmatically. If specified, the `session-timeout` parameter in `web.xml` takes precedence over this value. [1…`Integer.MAX_VALUE`; default 3600].
- `InvalidationIntervalSecs`—WebLogic Server checks for timed-out session objects on a regular basis, normally at least once a minute. This parameter tunes the frequency of the check. [1...604,800; default 60].
- `IDLength`—Tune the length of the session ID using this parameter. [8…`Integer.MAX_VALUE`; default 52].

- `URLRewritingEnabled`—Use this setting to enable or disable session tracking using URL rewriting. [`true` | `false`; default `true`].
- `CookiesEnabled`—Use this setting to enable or disable session tracking using cookies. [`true` | `false`; default `true`].
- `CookieMaxAgeSecs`—This parameter sets the timeout in seconds for the cookie used to track sessions on the client. After the specified time, the client deletes the cookie. Several values have special meaning. If set to –1, the client removes the cookie when the user exits the client. If set to 0, the cookie expires immediately. If set to `Integer.MAX_VALUE`, the cookie never expires. [–1, 0, 1…`Integer.MAX_VALUE`; default –1].
- `CookieDomain`—Set the domain of the cookie used to track the session. By default, the value is set by the domain configuration of the WebLogic Server.
- `CookieName`, `CookieComment`, `CookiePath`—A number of parameters allow customization of the cookies used to track sessions. See WebLogic Server documentation for more details: *http://e-docs.bea.com/wls/docs81/webapp/weblogic_xml.html*.
- `ConsoleMainAttribute`—The WebLogic Server Admin console offers a session-tracking function. The console can use one of the session attributes as an identifier. Use this property to specify the identifying session attribute.

Additional attributes are described in the section "Configuration of Session Persistence" later in this chapter.

Best Practice: Be aggressive when setting session timeouts. Inactive sessions can be a drag on WebLogic Server performance.

Best Practice: Set the `CookieDomain` parameter. Some browsers will not send cookies received from HTTP responses with HTTPS requests because the two occur on different ports. Ensure the cookie is not lost by consistently setting the cookie's domain.

Session Example

You can easily add sessions to your servlets. As an example, let's build a servlet that counts the number of times you have visited a given page. To do this, create a session and insert a counter object into the session. Each time the user revisits the page, we increment the counter object.

The following listing demonstrates use of the servlet session:

```
package com.learnweblogic.examples.ch3;

import java.io.*;
import java.text.DateFormat;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionServlet extends HttpServlet {

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

```
    throws ServletException, IOException {
        // Get the session object. Don't do this in real life.
        // Use getSession(false) and redirect the request to log in on
failure.
        HttpSession session = req.getSession(true);

        // set content type first, then obtain a PrintWriter
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();

        // then write the data of the response
        out.println("<HEAD><TITLE>SessionServlet Output</TITLE></
HEAD><BODY>");
        out.println("<h1>SessionServlet Output</h1>");

        // Retrieve the count value from the session
        Counter cntr = (Counter)
        session.getAttribute("sessiontest.counter");

        // Increment the counter. If the counter is not
        // currently contained in the session, create it
        if (null!=cntr) {
            cntr.increment();
        } else {
            cntr = new Counter();
            session.setAttribute("sessiontest.counter", cntr);
        }

        // And print out number of times the user has hit this page:
        out.println("You have hit this page <b>" + cntr.toString() +
        "</b> times.<p>");
       out.println("Click <a href=" + res.encodeURL("/SessionServlet") +
        ">here</a>");
       out.println(" to ensure that session tracking is working even " +
        "if cookies aren't supported.<br>");
        out.println("<p>");

        // Finally, demonstrate some of the more common methods in the
        // HttpSession object surrounding sessions:
        out.println("<h3>Request and Session Data:</h3>");
        out.println("Session ID in Request: " +
        req.getRequestedSessionId());
        out.println("<br>Session ID in Request from Cookie: " +
        req.isRequestedSessionIdFromCookie());
        out.println("<br>Session ID in Request from URL: " +
        req.isRequestedSessionIdFromURL());
        out.println("<br>Valid Session ID: " +
```

```
        req.isRequestedSessionIdValid());
        out.println("<h3>Session Data:</h3>");
        out.println("New Session: " + session.isNew());
        out.println("<br>Session ID: " + session.getId());
        DateFormat df =
DateFormat.getDateTimeInstance(DateFormat.DEFAULT,DateFormat.FULL);
        out.println("<br>Creation Time: " +
        df.format(new Date(session.getCreationTime())));
        out.println("<br>Last Accessed Time: " +
        df.format(new Date(session.getLastAccessedTime())));
        out.println("</BODY>");
    }

    class Counter {
        private int counter;
        public Counter() {
            counter = 1;
        }

        public void increment() {
            counter++;
        }

        public String toString() {
            return String.valueOf(counter);
        }
    }
}
```

The output of this servlet looks something like Figure 3–1.

If you click refresh to visit this page multiple times, you'll see that the counter increments each time by incrementing the HTTP session object. If you restart the Web browser, the counter restarts at 1 because on restart, the browser lost its session ID. WebLogic Server recognizes the request as the beginning of a new session and starts over with a new session object and a new counter.

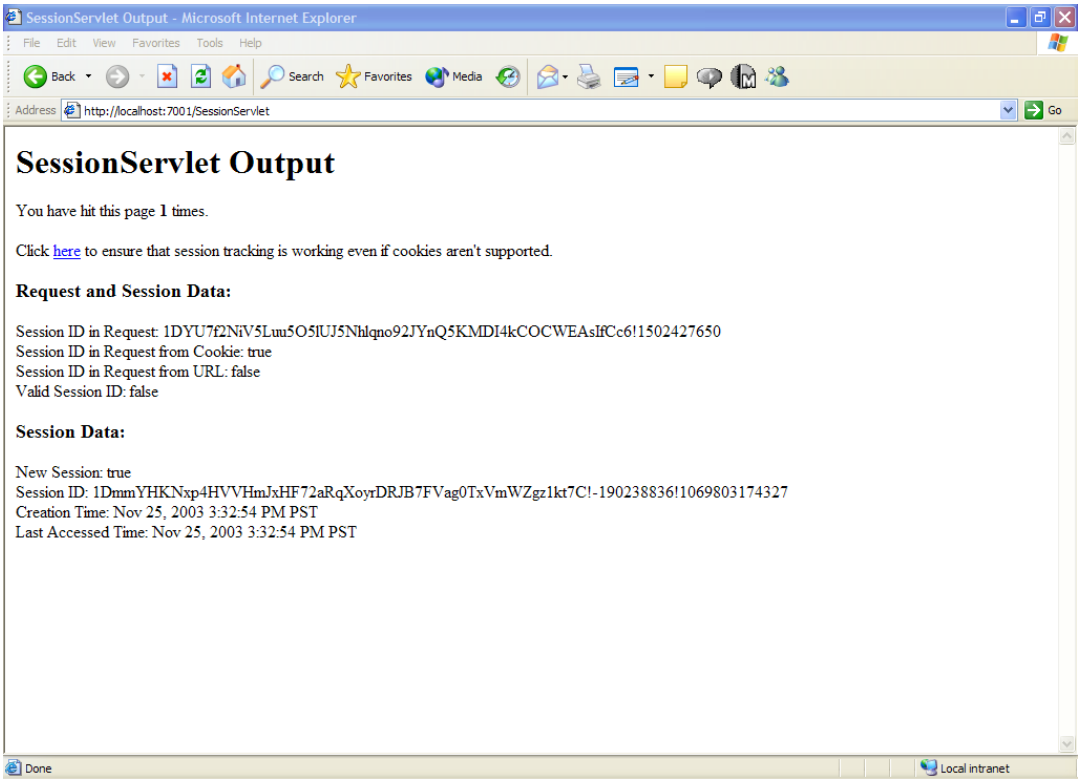

Figure 3–1
Servlet output.

Deploying the Session Servlet in WebLogic Server

In order to deploy the session servlet, you can use the code included on the CD-ROM accompanying this book. It is located in the file named *SessionServlet.war*, which is located in the subdirectory */examples/ch3*. It is deployed using the same process used for the `HelloServlet` example.

Session Events

Chapter 2 describes Web application events. These events can trigger code for various Web application events, like starting applications or binding attributes to the `ServletContext`. Similarly, developers can trigger code for session events. Developers can set triggers for similar session events, including session creation and attribute binding. These triggers are useful in initializing resources in session objects, releasing resources from session objects, or permanently storing session information for future reference.

Session Lifecycle Events

To configure code to handle session lifecycle events, write a class implementing either the `HttpSessionListener` or `HttpSessionActivationListener` and configure, and regis-

ter the class in the `web.xml` deployment descriptor. The `HttpSessionListener` defines the following methods:

- `public void sessionCreated(HttpSessionEvent event)`—WebLogic Server calls this method in registered listener classes after a new session is created.
- `public void sessionDestroyed(HttpSessionEvent event)`—WebLogic Server calls this method in registered listener classes before an existing session is released. The session is released either because it has timed out or because it has been invalidated.

WebLogic Server may move a session from one server to another within a cluster. In order to move the session, WebLogic Server serializes the session object in a process called *passivation*. The target server then restores or activates the session. Some attributes of a session may not be serializable (e.g., a JDBC connection). The `HttpSessionActivationListener` provides handlers for both activation and passivation:

- `void sessionWillPassivate(HttpSessionEvent event)`—WebLogic Server calls this method before it serializes the session prior to migrating the session to another instance.
- `void sessionDidActivate(HttpSessionEvent event)`—WebLogic Server calls this method after it restores the session on the target server instance.

The `HttpSessionEvent` object, passed to all these methods, provides only one method: `getSession()`. This method returns a reference to the session that has just been created or is about to be destroyed.

Register `HttpSessionListener` classes using the same listener syntax used for Web applications event listeners. Assuming that the class `com.mycorp.SessionLifecycleListener` implements `HttpSessionListener`, add the following lines into the `web.xml` deployment descriptor to register the listener:

```
<listener>
  <listener-class>
    com.mycorp.SessionLifecycleListener
  </listenerclass>
</listener>
```

The listeners are called in the same order as they appear in the `web.xml` deployment descriptor. Again WebLogic Server checks the interfaces implemented by each listener in order to route events properly. As a result of this dynamic type-checking, a single listener may implement multiple interfaces. For each listener interface a class implements, WebLogic Server calls the appropriate event handlers. This behavior can come in handy in a number of scenarios. In the following example, a counter is initialized with Web application startup and updated with each session created or destroyed:

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

public class SessionCounter implements HttpSessionListener,
ServletContextListener {
```

```
    private static int totalSessionCount, activeSessionCount;
    /** Creates a new instance of SessionContextCounter */
    public SessionContextCounter() {
        totalSessionCount = activeSessionCount = 0;
    }

    public void contextInitialized(ServletContextEvent event) {
        ServletContext sc = servletContextEvent.getServletContext();
        sc.setAttribute("SessionContextCounter" ,new SessionCounter());
    }

    public void contextDestroyed(ServletContextEvent event) {}

    public void sessionCreated(HttpSessionEvent event) {
        totalSessionCount++;
        activeSessionCount++;
    }

    public void sessionDestroyed(HttpSessionEvent event) {
        activeSessionCount--;
    }
}
```

Session Binding Events

The `HttpSessionAttributeListener` interface works like the previous listener interfaces. Implement the interface and register the class in the `web.xml` deployment descriptor. WebLogic Server calls the event handlers of implementing classes when an attribute is added, removed, or modified in a session object:

- `public void attributeAdded(HttpSessionBindingEvent event)`—WebLogic Server calls this method when an attribute is added to the session.
- `public void attributeRemoved(HttpSessionBindingEvent event)`—WebLogic Server calls this method when an attribute value is removed from the session.
- `public void attributeReplaced(HttpSessionBindingEvent event)`—WebLogic Server calls this method when an attribute value is replaced. The attribute is replaced rather than added when the method `setAttribute()` is called on the session using an existing attribute name.

Each of these handlers takes an `HttpSessionBindingEvent` as a parameter. The event class encapsulates the attribute change using the following accessors:

- `public String getName()`—Returns the name of the modified attribute.
- `public HttpSession getSession()`—Returns a reference to the modified session.
- `public Object getValue()`—Returns the value of the modified value when the value is either added or replaced.

The `HttpSessionAttributeListener` monitors changes to session attributes. The servlet specification also provides the ability to monitor the objects that may be bound to sessions. Like the attribute listener, classes that implement the `HttpSessionBindingListener` interface

receive notifications of binding events. Unlike the previous listeners, classes implementing the `HttpSessionBindingListener` are not registered in the `web.xml` deployment descriptor.

When application code calls the `setAttribute()` method on a session, the call includes both name and value parameters. If the value object implements `HttpSessionBindingListener`, WebLogic Server calls its `valueBound()` method. The `HttpSessionBindingListener` defines the following methods:

- `public void valueBound(HttpSessionBindingEvent event)`—WebLogic Server calls this method when the implementing value is bound to an `HttpSession`.
- `public void valueUnbound(HttpSessionBindingEvent event)`—WebLogic Server calls this method when an attribute value is unbound from an `HttpSession`. The value can be unbound if either the application calls the `removeAttribute()` method of the session or the session is destroyed (due to either an explicit call to `invalidate()` or a session timeout).

As discussed earlier, you should always configure sessions to time out. However, session event listeners, including the `HttpSessionListener` and the `HttpSessionBindingListener`, provide a way to persist session attributes. Persist the attribute using the `sessionDestroyed()` or `valueUnbound()` handler methods. When the user logs in again, restore the session using the data you saved from the previous session.

Baking Your Own Cookies

WebLogic Server uses cookies to track sessions, but developers may use cookies independent of HTTP sessions. Cookies provide a useful mechanism for storing long-term information about users. For example, you might want your application to recognize users when they return to your site. You might prepopulate the user's ID so that she does not have to enter it. Developers typically use cookies to implement handy features like this, common on many e-commerce sites.

In order to recognize users over longer periods of time, add a cookie into the HTTP response. If the user permits cookies, the browser stores the cookie on the user's machine and includes the cookie in future requests, even in future sessions. The application checks HTTP requests to see whether the cookie is included and uses the data found in the cookie. In the example above, this means that the user need only remember her password for authentication.

A cookie contains the following properties:

- A cookie name
- A single value
- Optional additional attributes such as a comment, a maximum age, a version number, and path and domain qualifiers

Cookies can store information for a long period of time, but this does not mean you should store all your data in cookies. As discussed in the previous section on sessions, browsers may not accept cookies. In addition, cookies are particularly bad places to store sensitive data: values can be intercepted, modified, or faked. For long-term storage of sensitive data, use another mechanism; the best practice is to use a database. See Chapter 5, "WebLogic Server JDBC and JTA," to use WebLogic Server to access a database using J2EE standards.

Cookies vs. Servlet Sessions

Servlet sessions are time-limited. Typically, servlet sessions last only as long as the lifetime of the browser session. Once the user exits the Web browser or the session times out, WebLogic Server automatically discards the `HttpSession` object. When you want your Web application to automatically recognize a user beyond the scope of a single session, use a cookie.

Here are some cases in which you should use servlet sessions:

- Track a user shopping cart.
- Cache data such as account balances that the user might look up more than once during a session.
- Store references or addresses of resources on which your application relies.

Problems best solved with a cookie might include these:

- Store convenience information such as a user's login ID.
- Store preference information such as the user's choice of language or color.
- Store an identifier used to track usage of your application.

Cookies have several characteristics: for one, you can use more than one cookie. Browsers impose limits on the use of browsers. The typical browser supports 20 cookies for each Web server, 300 cookies total, and may limit cookie size to 4KB each.

Making Cookies

Create a cookie using the `Cookie` constructor with two parameters: a name and a value. The value parameter represents the information that you want to store. This value may include the user's name or any arbitrary string value. To create a cookie, put the following in your servlet:

```
Cookie myCookie = new Cookie("Cookie name", "63");
```

Then, add your cookie to the HTTP response:

```
response.addCookie(myCookie);
```

The values stored in cookies can only be strings. Therefore, you should text-encode all data that you store in a cookie. Once you add a cookie to your response object, WebLogic Server automatically places it in the client browser. The client browser stores the cookie to the local computer disk for future requests.

Retrieving Cookies

When WebLogic Server invokes a servlet, the cookies sent by the browser are included in the `HttpServletRequest` object. These cookies are retrieved as an array of cookie objects using the following code:

```
Cookie[] cookies = request.getCookies();
```

Once you have the cookie objects, you can search for your specific cookie. You can use the `getName()` method to look at each cookie's name:

```
// Assign the Name of the First
// Cookie in The Array to String foo
```

```
String foo = Cookies[0].getName();
```

If the user's browser does not accept cookies it does not include cookies in subsequent requests. In this case, the result of `request.getCookies()` returns `null` (or an array without the cookie you added). In general, the servlet author has no way to determine in advance whether a particular client will accept cookies.

Useful Methods for Dealing with Cookies

There are a number of other methods that you can use with cookies:

- `public String getValue()` and `public void setValue(String newValue)`—Enable you to access the value stored in the cookie and set the value stored in the cookie respectively.
- `public void setMaxAge(int expiry)` and `public int getMaxAge()`—Set and get the number of seconds from the creation of a cookie to when the cookie expires (the cookie's age). The value 0 tells the browser to expire immediately. The value –1 means that the cookie expires when the Web browser exits—the default configuration. To make your cookies last indefinitely, set the value to the maximum possible integer:

  ```
  // Set the cookie myCookie to last indefinitely
  myCookie.setMaxAge(Integer.MAX_VALUE);
  ```

- `public void setDomain(String domain)`—Manually set the cookie's domain using this method. Browsers use the domain to select cookies to include in HTTP requests. The domain is typically the name of the server (e.g., `www.learnweblogic.com`). If the domain begins with a dot (`.`), the cookie should be returned with requests to any server in the specified DNS (Domain Name System) zone. For example, browsers will send a cookie with the domain `.learnweblogic.com` to a server named `www.learnweblogic.com`.

Best Practice: Set cookie domains. Some browsers will not send cookies received from HTTP responses with HTTPS requests because the two occur on different ports. Ensure the cookie is not lost by consistently setting the cookie's domain.

- `public void setComment(String comment)` and `public String getComment()`—Can be used to set the comment field in your cookie and read the comment. This is very useful for occasions when individuals are browsing their cookie store on their Web browser. To set the comment field in a cookie named myCookie, place the following code in your servlet:

  ```
  // Set the Comment Field in Cookie myCookiemyCookie.setComment("gumby999");
  ```

- `public void setSecure(boolean)` and `public boolean getSecure()`—Enable you to set the security settings for the cookie. The following code requires that the cookie be sent only over a secure channel such as SSL:

  ```
  // Require That the Cookie myCookie
  ```

```
// Only Be Sent over Secure Channels
myCookie.setSecure(true);
Custom Cookies for Personalization
```

The first step in implementing a custom personalization cookie is to add the mechanism to set and look for the cookie in your pages. The following is a trivial `doGet()` method that indicates how to look for a cookie to prepopulate the login field:

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

This example uses a session object to represent being logged in. We create and invalidate sessions to set the user's login status.

```
// Get the session if one exists
HttpSession session = req.getSession(false);
```

The local variables `cookieFound` and `thisCookie` represent the cookie named `LoginCookie`. The `for` loop cycles through each of the cookies provided by the browser. If `cookieFound` is true, the `thisCookie` variable will reference the cookie.

```
// Try to retrieve the cookie from the request.
boolean cookieFound = false;
Cookie thisCookie = null;
Cookie[] cookies = req.getCookies();
if(null != cookies) {
    // Look through all the cookies and see if the
    // cookie with the login info is there.
    for(int i=0; i < cookies.length; i++) {
        thisCookie = cookies[i];
        if (thisCookie.getName().equals("LoginCookie")) {
            cookieFound = true;
            break;
}   }   }
```

If the user has performed a logout action, invalidate the current session.

```
// Logout action removes session, but the cookie remains
if(null != req.getParameter("Logout")) {
    if(null != session) {
        session.invalidate();
        session = null;
}   }
```

When the user is not logged in, check to see whether the user has provided login information. This code is for demonstration purposes only and will accept any login and password values.

```
        if(null == session) {       // If the user is not logged in
            String loginValue = req.getParameter("login");
            boolean isLoginAction = (null==loginValue)?false:true;
```

If no cookie exists, the code creates a new one with the login as its value. When the cookie exists, the servlet sets the changed value of the cookie. Note that we add the cookie again after we change the value of the cookie. This call informs WebLogic Server to send the cookie back to the client and updates the cookie's value.

```
        if(isLoginAction) {     // User is logging in
            if(cookieFound) {
                if(!loginValue.equals(thisCookie.getValue())) {
                    thisCookie.setValue(loginValue);
                    res.addCookie(thisCookie);
                }
            } else {
               // If the cookie does not exist, create it and set value
                thisCookie = new Cookie("LoginCookie",loginValue);
                res.addCookie(thisCookie);
            }
            // create a session to show that we are logged in
            session = req.getSession(true);
            displayLogoutPage(req, res);
```

In the case where we display the login page, use the value in the cookie to prepopulate the page's login field.

```
            } else {
               // Display the login page. If the cookie exists, set login
                if(cookieFound) {
                   req.setAttribute("login", thisCookie.getValue());
                }
                displayLoginPage(req, res);
            }
```

Finally, logged in users are provided the opportunity to log out.

```
        } else {
            // If the user is logged in, display a logout page
            displayLogoutPage(req, res);
        }
    }
```

Long-Term CookieServlet Example

In this example, we extend the preceding code to create a complete servlet. This servlet deviates only slightly from the preceding example. The `CookieServlet` example handles both GET and POST methods and extends the lifespan of the cookie to beyond the foreseeable future (roughly 65 years). Also, this version uses display methods to generate HTML pages.

```
package com.learnweblogic.examples;

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

public class CookieServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws IOException {
        handleRequest(req, res);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws IOException {
        handleRequest(req, res);
    }

   public void handleRequest(HttpServletRequest req, HttpServletResponse res)
        throws IOException {
        // Get the session if one exists
        HttpSession session = req.getSession(false);

        // Try to retrieve the cookie from the request.
        boolean cookieFound = false;
        Cookie cookie = null;
        Cookie[] cookies = req.getCookies();
        if (null != cookies) {
            // Look through all the cookies and see if the
            // cookie with the login info is there.
            for (int i = 0; i < cookies.length; i++) {
                cookie = cookies[i];
                if (cookie.getName().equals("LoginCookie")) {
                    cookieFound = true;
                    break;
                }
            }
        }

        // Logout action removes session, but the cookie remains
        if (null != req.getParameter("Logout")) {
```

```
            if (null != session) {
                session.invalidate();
                session = null;
            }
        }

        if (null == session) { // If the user is not logged in
            String loginValue = req.getParameter("login");
           boolean isLoginAction = (null == loginValue) ? false : true;

            if (isLoginAction) { // User is logging in
                if (cookieFound) { // If the cookie exists update the
value only if changed
                    if (!loginValue.equals(cookie.getValue())) {
                        cookie.setValue(loginValue);
                        res.addCookie(cookie);
                    }
                } else {
                   // If the cookie does not exist, create it and set value
                    cookie = new Cookie("LoginCookie", loginValue);
                    cookie.setMaxAge(Integer.MAX_VALUE);
                    res.addCookie(cookie);
                }
                // create a session to show that we are logged in
                session = req.getSession(true);
                session.setAttribute("login",loginValue);
                   req.setAttribute("login",loginValue);
                displayLogoutPage(req, res);
            } else {
               // Display the login page. If the cookie exists, set login
                if (cookieFound) {
                    req.setAttribute("login", cookie.getValue());
                }
                displayLoginPage(req, res);
            }
        } else {
            // If the user is logged in, display a logout page
           String loginValue = session.getAttribute("login").toString();
            req.setAttribute("login",loginValue);
           displayLogoutPage(req, res);
        }
    }

    public void displayLoginPage(
        HttpServletRequest req,
        HttpServletResponse res)
        throws IOException {
```

```
        String login = (String) req.getAttribute("login");
        if (null == login) {
            login = "";
        }
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html><body>");
        out.println(
            "<form method='POST' action='"
                + res.encodeURL(req.getRequestURI())
                + "'>");
        out.println(
          "login: <input type='text' name='login' value='" + login + "'>");
        out.println(
            "password: <input type='password' name='password' value=''>");
       out.println("<input type='submit' name='Submit' value='Submit'>");
        out.println("</form></body></html>");
    }

    public void displayLogoutPage(
        HttpServletRequest req,
        HttpServletResponse res)
        throws IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html><body>");
      out.println("<p>Welcome " + req.getAttribute("login") + "</p>");
        out.println(
            "<form method='GET' action='"
                + res.encodeURL(req.getRequestURI())
                + "'>");
        out.println(
            "Click <a href='"
                + res.encodeURL(req.getRequestURI())
                + "'>here</a> to reload this page.<br>");
        out.println("<input type='submit' name='Logout'
value='Logout'>");
        out.println("</form></body></html>");
    }
}
```

In a real-world situation, you want your application to authenticate users using more secure methods. In Chapter 13, "Application Security with WebLogic Server 8.1," we further discuss how to have your application log in an existing user.

Deploying the CookieServlet in WebLogic Server

To deploy the `CookieServlet`, use the code included on the CD-ROM accompanying this book, in */code/ch3/cookieServlet.war*.

Step 0: Installing the CookieServlet Application

Follow the steps used in previous examples to build and deploy the application code. Be sure to create a new directory for your work.

Step 1: Modifying Browser Settings

To view the `CookieServlet`, modify your Web browser to enable you to see better what happens with cookies. If you are using Internet Explorer, use Tools → Internet Options... to delete. Choose the Advanced option in the left-hand panel. You see a screen such as the one in Figure 3–2.

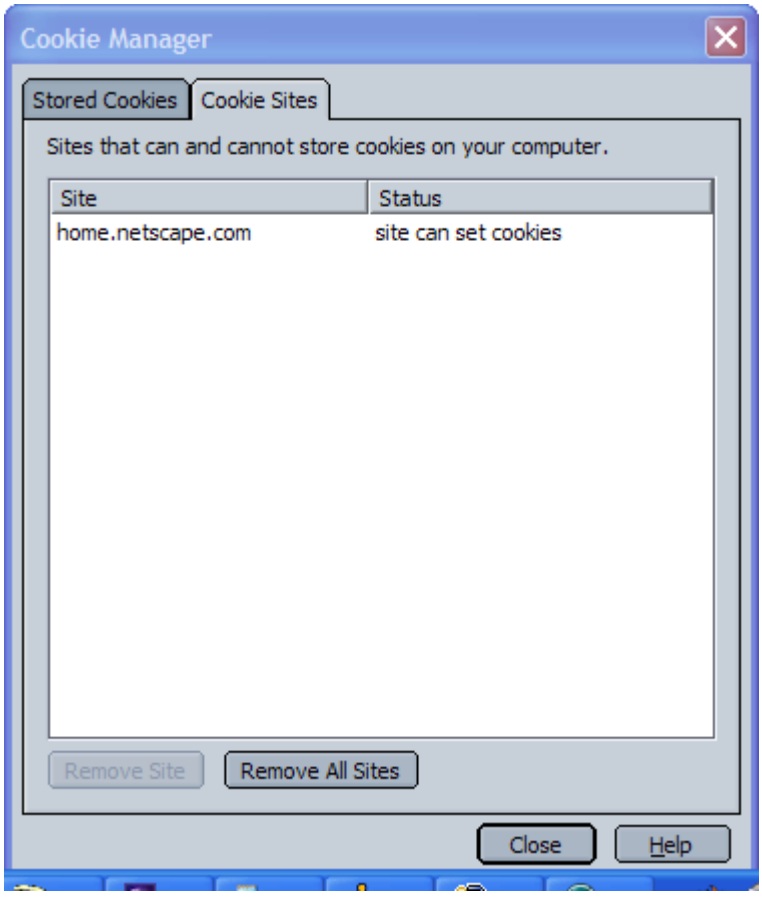

Figure 3–2
Netscape Navigator's preferences dialog.

Select the option to warn you before accepting cookies. This enables you to see any cookies that WebLogic Server intends to put into your browser. Microsoft Internet Explorer requires that you modify the settings and set your Internet security configuration for the browser. If you decide not to modify your browser settings, you won't see the details of the cookie placement—but the example still works.

Step 2: Visiting the CookieServlet

Visit the `CookieServlet` by pointing your Web browser at the deployment location for your WebLogic Server instance. If you have deployed on your local machine, at port 7001 (the default), you can view the example at *http://127.0.0.1:7001/CookieServlet*.

You should be immediately see the `CookieServlet` login screen (see Figure 3–3), which displays the following:

Fill in the User ID field (the servlet does not check the password) and press the submit button. You should see the cookie servlet display (see Figure 3–4).

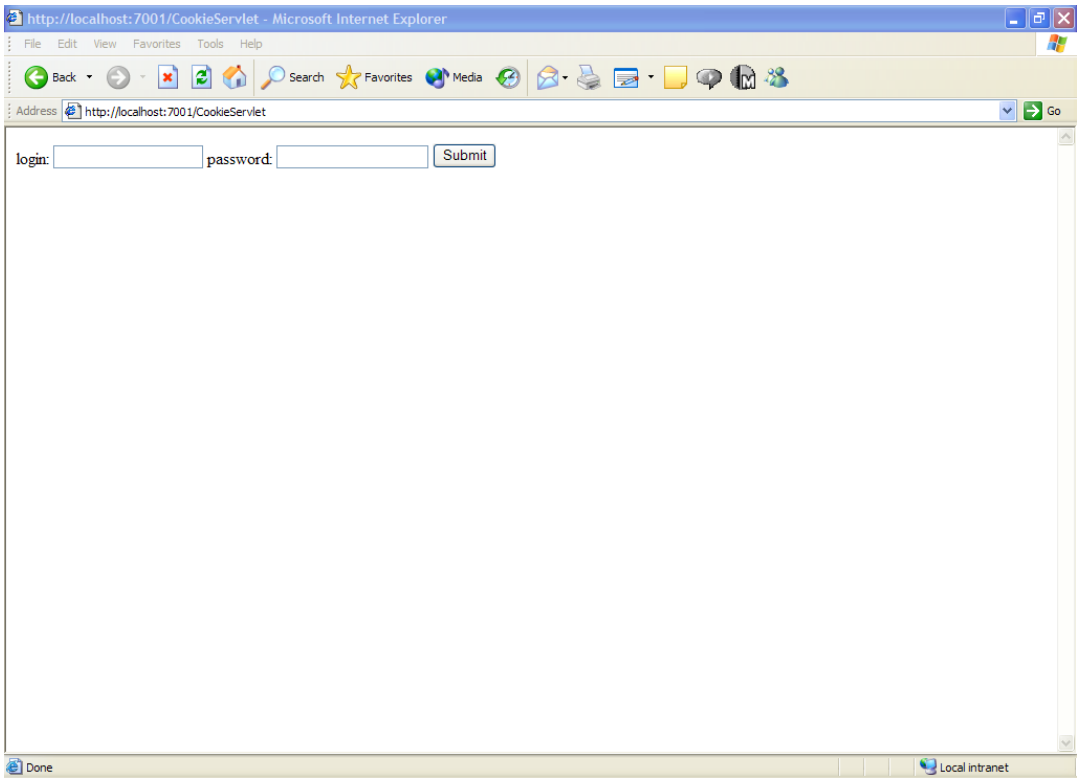

Figure 3–3
Initial response from CookieServlet.

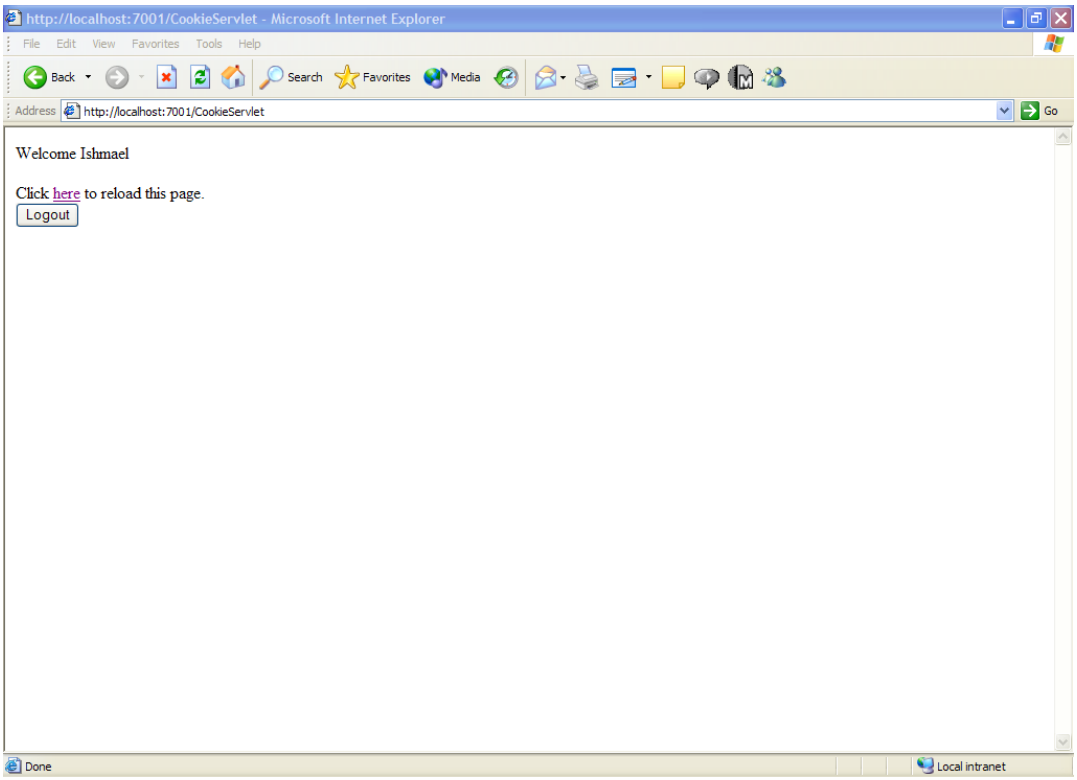

Figure 3–4
CookieServlet display after login.

Step 3: Revisiting the CookieServlet

Log out and visit the `CookieServlet` again, with the same Web browser, at *http://127.0.0.1:7001/CookieServlet*.

You should see the screen shown in Figure 3–5.

The `CookieServlet` recognizes the cookie in your Web browser and welcomes you with your previous ID.

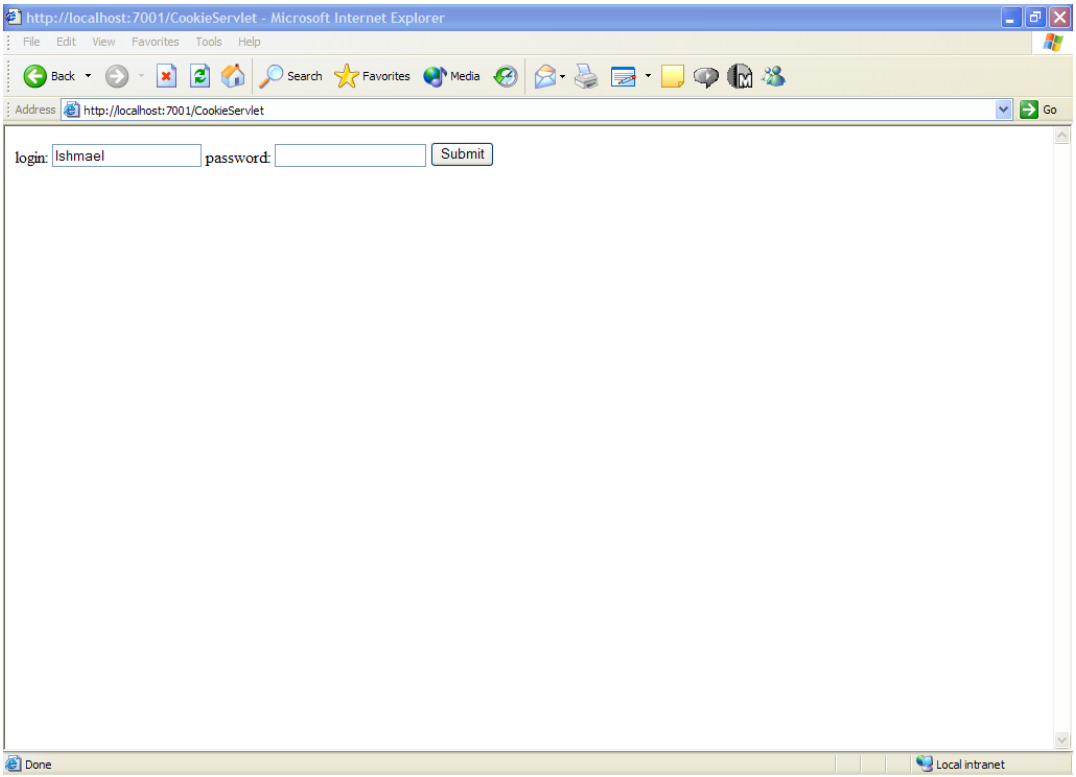

Figure 3–5
Revisiting the CookieServlet.

Filters

So far, we have discussed one way to handle HTTP events—Web browsers generate HTTP requests, and servlets process the requests. Filters augment the servlet request cycle. Filters intercept requests prior to servlet processing and responses after servlet processing. Filters can be used to perform functions across multiple servlets in a Web application, to add features to existing servlet code, and to create reusable functions for use in multiple Web applications.

For instance, a filter can be used to provide customized authentication. The filter intercepts requests prior to calling the servlet. The filter determines whether the user has sufficient privileges to access the requested servlet. Unauthorized users are rejected without the servlet ever knowing about the request. As a result, authentication code can be located in a single location, the filter, rather than scattered throughout the multiple servlets in the Web application.

Similarly, filters can log user actions, compress output streams, or even transform the servlet's output to meet the demands of a target browser. Filters can achieve this flexibility because they are not hardwired into servlets, but instead configured in the `web.xml` deployment descriptor to intercept requests.

Filter Interfaces

In addition to configuration similarities, filters and servlets share other characteristics. Both execute in a multithreaded environment. The same safeguards necessary to servlet development regarding careful use of class and instance variables apply equally to filters. Also, where servlets handle requests using the `doGet()` and `doPost()` methods, filters implement a `doFilter()` method.

Implement the `Filter` interface (`javax.servlet.Filter`) by writing `init()`, `destroy()`, and `doFilter()` methods:

- `public void init(FilterConfig config) throws ServletException`—WebLogic Server executes the `init()` method of registered filters during the initialization phase. Typically, the `init()` method stores the `FilterConfig` object for future use.
- `public void destroy()`—WebLogic Server executes the `destroy()` method of registered filters during the Web application's end of service phase.
- `public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException`—WebLogic Server calls the `doFilter()` method for selected filters. The `Filter` should handle the request or pass the request to the provided `FilterChain`.

The `init()` method takes a `FilterConfig` object. The `FilterConfig` provides the filter access to the `ServletContext`, and filter configuration parameters in the `web.xml` file. `FilterConfig` defines the following methods:

- `public String getFilterName()`—Use this method to access the name of the filter as set in the `web.xml` file.
- `public String getInitParameter(String name)`—Look up the value of an init parameter using this method and the name of the parameter. Parameters are defined in the `web.xml` file.
- `public Enumeration getInitParameterNames()`—Returns the names of all the parameters defined for the filter in an `Enumeration`.
- `public ServletContext getServletContext()`—Use the `FilterConfig` to access the Web application's `ServletContext`.

Also, the `doFilter()` method includes the familiar `ServletRequest` and `ServletResponse` objects and the new `FilterChain` object. The `FilterChain` interface defines only a `doFilter()` method but is at the center of filter flow control. In the following custom authentication example, a mean-spirited filter is used to limit access to logged-in users whose login name begins with the letter D. The filter itself intercepts the request prior to the servlet:

```
package com.learnweblogic.examples;

import javax.servlet.*;
import javax.servlet.http.*;

public class OnlyDsFilter implements Filter {

    FilterConfig config = null;

    public void doFilter(javax.servlet.ServletRequest req,
javax.servlet.ServletResponse res, javax.servlet.FilterChain chain)
```

```
throws java.io.IOException, javax.servlet.ServletException {
        if(req instanceof HttpServletRequest && res instanceof
HttpServletResponse) {
            HttpServletRequest httpReq = (HttpServletRequest)req;
            HttpServletResponse httpRes = (HttpServletResponse)res;
            String user = httpReq.getRemoteUser();
            if(null != user && !user.startsWith("D")) {
                httpRes.sendError(HttpServletResponse.SC_FORBIDDEN,
"Only D users allowed");
                return;
            }
        }
        chain.doFilter(req, res);
        return;
    }

    public void init(javax.servlet.FilterConfig filterConfig) throws
javax.servlet.ServletException {
        config = filterConfig;
    }

    public void destroy() {
    }
}
```

Configure Filters

Configure filters in the `web.xml` file. As with servlets, use the `filter` elements to define named filters, and `filter-mapping` elements to link this filter to specific requests. The `filter` element should contain a `filter-name`, and `filter-class` element. A bare-bones `filter` element looks like this:

```
<filter>
    <filter-name>OnlyDs</filter-name>
    <filter-class>com.learnweblogic.examples.OnlyDsFilter</filter-
class>
</filter>
```

The filter may also contain any number of `init-params` and the optional `description` element. The servlet specification also defines `icon` and `display-name` elements, but WebLogic Server ignores these. A more general version of the above filter might be configured as follows:

```
<filter>
    <filter-name>OnlyDs</filter-name>
    <description>OnlyDs limits access to resources by user name.</
description>
    <filter-class>com.learnweblogic.examples.AuthByName</filter-class>
```

```
    <init-param>
        <param-name>accept-pattern</param-name>
        <param-value>D.*</param-value>
    </init-param>
</filter>
```

This configuration might be used with a more general authentication filter. The parameter `accept-param` provides a regular expression to define which names are allowed access. The modifications to the above filter, necessary to use this parameter, are left to the reader.

Map requests to filters using the `filter-mapping` element in the `web.xml` file. The mapping ties either a URL pattern or named servlet to a filter. The `filter-mapping` element contains a `filter-name` and either a `url-pattern` or a `servlet-name`. WebLogic Server selects filters by matching `url-patterns` against request URLs, and then selects the appropriate filter using the `filter-name`. WebLogic Server may select multiple filters, and it calls them in the order they appear in the `web.xml` file.

After matching URLs, WebLogic Server selects filters using servlet names. Rather than using the request URL, WebLogic Server matches the selected servlet `servlet-name` in the `filter-mapping` to the servlet it has selected to run. Again, these are selected in the same order as listed in the `web.xml` file.

```
<filter-mapping>
    <filter-name>OnlyDs</filter-name>
    <url-pattern>/D/*</url-pattern>
</filter-mapping>

<filter-mapping>
    <filter-name>OnlyDs</filter-name>
    <servlet-name>SpecialDServlet</servlet-name>
</filter-mapping>
```

Intercepting Servlet Output

The code example `OnlyDsFilter` intercepts requests prior to servlet processing. To intercept the response from the servlet, include code after the call to `doFilter()`:

```
    public void doFilter(javax.servlet.ServletRequest req,
javax.servlet.ServletResponse res, javax.servlet.FilterChain chain)
throws java.io.IOException, javax.servlet.ServletException {
        chain.doFilter(req, res);
        // do post-processing here...
        return;
    }
```

A single filter may include code both before and after it calls the servlet. In fact, to modify servlet output, create a stand-in `HttpServletResponse` before calling `doFilter()`, and modify the servlet output after the `doFilter()` call.

The classes `HttpServletRequestWrapper` and `HttpServletResponseWrapper` provide a starting point for creating these stand-in objects. The wrapper classes take request or response objects in their constructors. The wrappers provide methods that simply call each of the methods in the `HttpServletRequest` or `HttpServletResponse`. Override methods in these wrappers to customize the behavior of the class.

Filter Example

In the following example, a filter replaces special tokens in the body of the HTTP response. The filter works like the Ant task named `filter`. The deployer of the filter configures both a token and value. The filter looks for the token, preceded and followed by the `@` sign, in the response body. The filter then replaces instances of the token and `@` signs, adjusts the response `content-length`, and returns the modified response body.

```
package com.learnweblogic.examples;

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

public class TokenReplacementFilter implements Filter {
    private String replToken, replValue;

    public void init(FilterConfig filterConfig)
    throws ServletException {
        replToken = filterConfig.getInitParameter("token.name");
        replValue = filterConfig.getInitParameter("token.value");
        if (null == replToken) {
            throw new ServletException("TokenReplacementFilter named " +
             filterConfig.getFilterName() +
             " missing token.name init parameter.");
        }
        if (null == replValue) {
            throw new ServletException("TokenReplacementFilter named " +
             filterConfig.getFilterName() +
             " missing token.value init parameter.");
        }
    }
    public void destroy() {}

    public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
        BufferedResponse resWrapper = new BufferedResponse(
        (HttpServletResponse)response);

        chain.doFilter(request, resWrapper);
```

```
        if (resWrapper.getOutputType() == BufferedResponse.OT_WRITER) {
            String resBody = new String(
           resWrapper.toByteArray(),resWrapper.getCharacterEncoding());
            if (resWrapper.getContentType().equals("text/html")) {
                resBody = resBody.replaceAll("@" + replToken + "@",
replValue);
                response.setContentLength(resBody.length());
            }
            PrintWriter writer = response.getWriter();
            writer.println(resBody);
        } else if (resWrapper.getOutputType() ==
BufferedResponse.OT_OUTPUT_STREAM) {
            ServletOutputStream out = response.getOutputStream();
            out.write(resWrapper.toByteArray());
        }
    }
```

The `init()` method reads `init-params` and sets the search and replace strings as instance variables within the filter. Notice that if either parameter is missing, the filter halts initialization of the Web application when it throws a `ServletException`. The `doFilter()` method creates a substitute `HttpServletResponse` and passes this object to the `FilterChain` for further processing. When the `FilterChain` returns, `doFilter()` extracts the body of the response, modifies the response body, and then writes this modified value to the original response object. Finally, the `doFilter()` method is careful to not replace strings if the servlet uses an `getOutputStream()` or if the `content-type` is not `text/html`.

The `BufferedResponse` class provides an `HttpServletResponse` to be used by a servlet. The response provides all the normal methods to the servlet but buffers servlet output for later processing by a filter.

```
    class BufferedResponse extends HttpServletResponseWrapper {

        public static final short OT_NONE = 0, OT_WRITER = 1,
OT_OUTPUT_STREAM = 2;

        private short outputType = OT_NONE;
        private PrintWriter writer = null;
        private BufServletOutputStream out = null;
        private String contentType;

        public BufferedResponse(HttpServletResponse response){
            super(response);
        }

       public ServletOutputStream getOutputStream() throws IOException {
            if (outputType == OT_WRITER) {
                throw new IllegalStateException();
            } else if (outputType == OT_OUTPUT_STREAM) {
```

```
            return out;
        } else {
            out = new BufServletOutputStream();
            outputType = OT_OUTPUT_STREAM;
            return (ServletOutputStream)out;
        }
    }

    public PrintWriter getWriter() throws IOException {
        if (outputType == OT_OUTPUT_STREAM) {
            throw new IllegalStateException();
        } else if (outputType == OT_WRITER) {
            return writer;
        } else {
            writer = new PrintWriter(
            new OutputStreamWriter(
            getOutputStream(),getCharacterEncoding()));
            outputType = OT_WRITER;
            return writer;
        }
    }

    public short getOutputType() {
        return outputType;
    }

    public String getContentType() {
        return contentType;
    }

    public void setContentType(String contentType) {
        this.contentType = contentType;
        super.setContentType(contentType);
    }

    public void flushBuffer() throws IOException {
        if (outputType == OT_WRITER) {
            writer.flush();
        } else if (outputType == OT_OUTPUT_STREAM) {
            out.flush();
        }
    }

    public void reset() {
        resetBuffer();
        outputType = OT_NONE;
        super.reset();
```

```
        }

        public void resetBuffer() {
            if(null != out) {
                out.reset();
            }
        }

        public byte[] toByteArray() {
            flushBuffer();
            if(null != out) {
                return(out.toByteArray());
            } else {
                return null;
            }
        }
    }
```

The `BufferedResponse` extends the `HttpServletResponseWrapper`. The wrapper provides a default implementation for all of the methods in the `HttpServletResponse`, which passes the request to the wrapped response. The `BufferedResponse` overrides several of these methods.

First, we review the `getWriter()` and `getOutputStream()` methods. These methods create `PrintWriter` or `OutputStream` objects that write to a byte array rather than into the response. The servlet writes to this temporary store. The filter accesses the array using the `toByteArray()` method and processes the array before writing it to the real response object.

The servlet specification is very clear that servlets may call only one of these two methods. The `BufferedResponse` enforces this restriction by storing the first selected method in the instance variable, `outputType`, and by checking this variable before returning an `OutputStream` or `PrintWriter`.

Again, the default behavior of the `HttpServletResponseWrapper` is to call the wrapped response object. As a result, the default behavior of methods `flushBuffer()`, `resetBuffer()`, and `reset()` is to update the original response object's output buffer. The `BufferedResponse` uses a temporary buffer, so method calls that update to the original buffer will have unexpected results. Instead, override these methods to update the local buffer. Chapter 2 includes the admonition to avoid flushing output streams in your servlet. Similarly, you should be careful to override these methods to ensure servlets do not prematurely flush the response output buffer.

Best Practice: Override `HttpResponseWrapper` methods `flushBuffer()`, `reset()`, and `resetBuffer()`.

This filter needs to know the `Content-Type` of the response. Normally, the response object does not provide a method to get this value. The `BufferedResponse` overrides the setter to mirror this value to a local variable and provides a getter for the same value.

The final piece of code is the `BufServletOutputStream`. The `getOutputStream()` method of `ServletResponse` returns a `ServletOutputStream`. The `ServletOutputStream` is declared abstract, so in order to create our own, we must provide a class implementing

`write(int i)`. In addition, we provide methods `toByteArray()` and `reset()` to gain access to and control over the underlying `ByteArrayOutputStream`:

```
    class BufServletOutputStream extends ServletOutputStream {
        ByteArrayOutputStream bufferedOut;

        public BufServletOutputStream() {
            bufferedOut = new ByteArrayOutputStream();
        }

        public void write(int i) throws IOException {
            bufferedOut.write(i);
        }

        public byte[] toByteArray() {
            return bufferedOut.toByteArray();
        }

        public void reset() {
            bufferedOut.reset();
        }
    }
}
```

To finish out the example, here is a `web.xml` file that configures the filter to replace `@version@` with `1.0` on all URLs in the servlet:

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
  <filter>
    <filter-name>ReplaceVersion</filter-name>
    <filter-class>com.learnweblogic.examples.TokenReplacementFilter</
filter-class>
    <init-param>
      <param-name>token.name</param-name>
      <param-value>version</param-value>
    </init-param>
    <init-param>
      <param-name>token.value</param-name>
      <param-value>1.0</param-value>
    </init-param>
  </filter>

  <filter-mapping>
```

```
    <filter-name>ReplaceVersion</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>
```

Deploying TokenReplacementFilter

In order to deploy the session servlet, you can use the code included on the CD-ROM accompanying this book. It is located in the file named *TokenReplacementFilter.war*, which is located in the subdirectory */examples/ch3*. It is deployed using the same process used for the `HelloServlet` example. Once deployed, access the servlet at *http://localhost*:7001/*TokenReplacementFilter*, and then at *http://localhost*:7001/*TokenReplacementFilter/replace*, to see the impact of the filter. See Figures 3–6 and 3–7 for the display of the servlet with and without the filter.

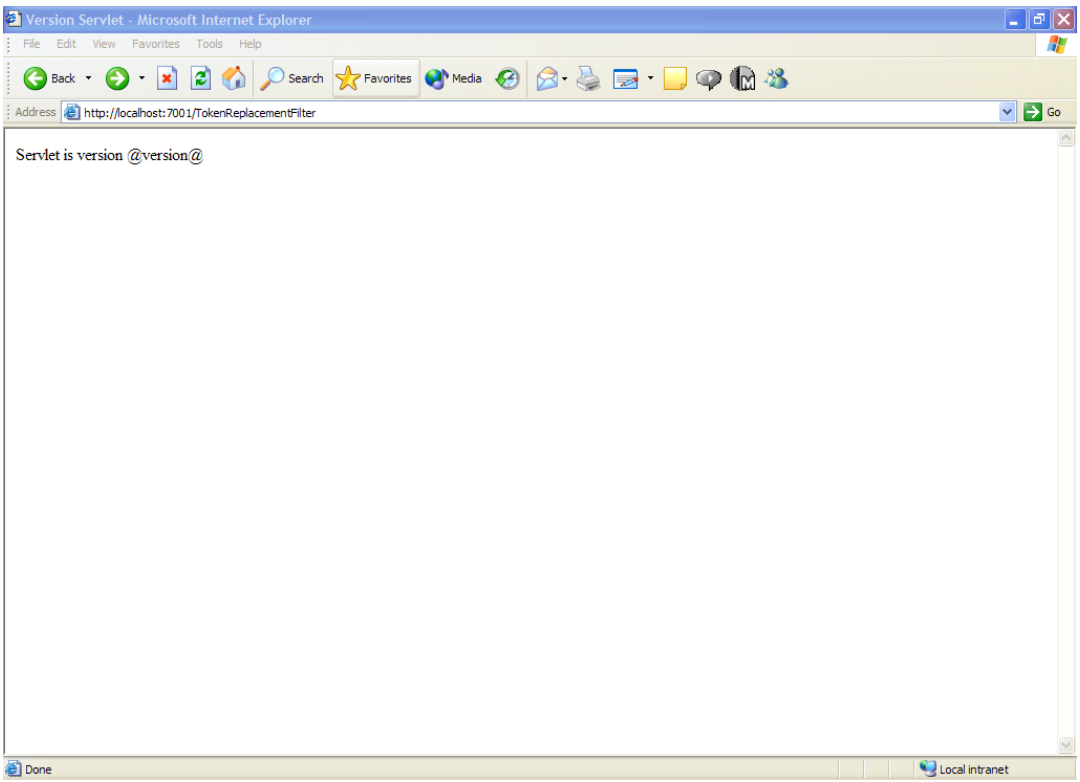

Figure 3–6
Version servlet display without filter.

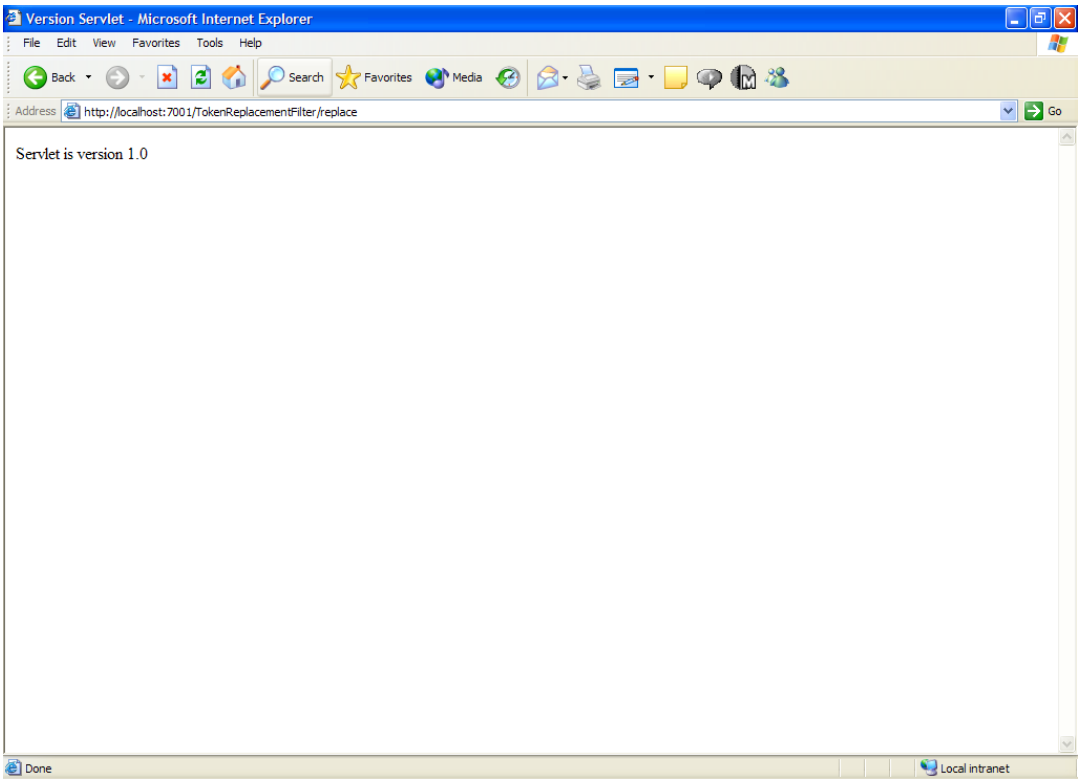

Figure 3–7
Version servlet display with filter.

Best Practice: Use filters to apply features to a Web application as a whole. Filters can reduce download times by compressing servlet output and can apply security policies to all resources in a Web application.

Using Servlets with WebLogic Server Clustering

Clustering, discussed in Chapter 12, is an important capability that provides redundancy and task balancing across multiple processes and/or machines. For WebLogic Server deployment, it is necessary to design your hardware environment to accommodate clustering. Similarly, you must design your servlets and Web applications to work well when clustered.

Persisting Session Information

When a Web application runs in a clustered environment, the cluster can recover from the failure of any one instance. When one instance fails, WebLogic Server directs client requests to one of the remaining instances. In many cases, redirecting the client request is sufficient. However, if the Web

application uses sessions to store information between requests, you must take special precautions to make the session available on the target instance.

You have five options for storing session information in a WebLogic Server deployment:

- Do not protect your session information. In the event of a failure, all user sessions on the failed instance are lost. Sessions are lost because the sessions are only stored in that WebLogic Server instance.
- Protect your session information in a database. In the event of a failure, all session information is persisted in the database and readily available. WebLogic Server saves the data in the database after each request, and with each new request, WebLogic Server recovers the session data.
- Protect your session information with in-memory replication. In this model, a given session is always directed to the same server in a cluster, typically referred to as the primary for that session. The cluster automatically chooses another secondary server to act as a hot backup to the primary server. This node is updated across the network by the primary node after every change to the `HttpSession` object. In other words, every time you call `setAttribute()` or `removeAttribute()`, WebLogic Server automatically and synchronously updates the copy of the `HttpSession` on the secondary server. In the event of a failure to either the primary or secondary server, a different server in the cluster automatically takes its place to handle the session.
- Protect your session information in a temporary file. Using this option, WebLogic Server can recover sessions of a failed instance if it has access to the file. In environments using networked file systems, WebLogic Server can recover sessions across machines.
- Protect your session information in a cookie. In this model, WebLogic Server stores session information in an HTTP cookie. WebLogic Server instances take no special precaution to recover from instance failure because information is stored on the client. However, the use of cookies imposes a number of limitations. Many of these limits are discussed in the "Baking Your Own Cookies" section of this chapter. Among these are that some browsers do not accept cookies, and the data stored in cookies is subject to manipulation. In addition, when cookies are used to store sessions, you must store only string values in the session.

Session Protection Performance Implications

WebLogic Server greatly improves its ability to recover from failure when it stores session information in a database or on the filesystem. However, this additional reliability comes with a performance cost. Unprotected session storage provides the highest level of performance. The overhead required to store session information adds work and degrades performance to varying degrees. The right match of performance, reliability, and features depends on your needs, including your application's characteristics, hardware configuration, and the cost of failure.

In general, in-memory replication imposes the least overhead. File, cookie, and database persistence share similar performance characteristics and are much slower than in-memory replication. In one test, in-memory replication rated 5.5 times faster than database replication. However, generic benchmarks provide little guidance for your application.

The size of a session has a huge impact on performance. A larger session slows response times and magnifies the impact of session protection. Make sure you limit the data stored in sessions. Also, your specific configuration will impact performance greatly. Another test has shown that for database storage, moving the database from the same machine as WebLogic Server to a remote machine will slow session management by 68 percent.

Choosing the Right Level of Protection

The right amount of session protection for your application depends on your needs. To what extent is performance critical? What is the cost of losing a user session? If performance is most important, you may choose not to protect your session information. If the loss of session information is merely inconvenient, then you should consider leaving session data unprotected.

For rock-solid protection of session information, database or file persistence is the best choice. You are much less likely to lose your session information, because every change in the session is saved as a transaction against the database or saved to a file. WebLogic Server can recover sessions stored to a database or filesystem as long as the target system is still available. Unfortunately, the reliability of storing sessions in a database or filesystem comes at the cost of slower performance.

If your session storage needs are minimal, and you can be sure your users' browsers will accept cookies, then cookie storage is an option. Because session information is stored on the client rather than the server, cookies are a reliable option. Unfortunately, cookie storage has all the performance issues of file and database storage, is insecure, limits functionality, and is not universally supported by clients.

WebLogic Server provides the additional option of in-memory replication. In-memory replication is highly reliable and provides better performance than database or file persistence. WebLogic Server performs in-memory replication by storing the session in memory and copying the session on another instance. If the primary instance fails, the user is directed to the secondary instance. In-memory replication is highly reliable, because the likelihood is low that multiple servers go down simultaneously. Yet, the performance cost of replication is significantly lower than file or database storage. In-memory replication provides a high-performance reliable alternative.

Best Practice: Begin with in-memory replication. For greater performance, stop protecting sessions. For greater reliability, store sessions to a database or the filesystem.

Configuration of Session Persistence

Configure the way WebLogic Server manages sessions using the `weblogic.xml` deployment descriptor. Use the following session parameters to customize WebLogic Server session management:

- `PersistentStoreType`—This parameter specifies where WebLogic Server stores session objects between user requests. For additional information regarding this setting, see the section "Using Servlets with WebLogic Server Clustering" in this chapter. Set the storage type to one of six values:
  - `memory`: Store session objects in memory.
  - `file`: Store session objects to files on the server. If specified, also set `PersistentStoreDir`.
  - `jdbc`: Uses a database to store persistent sessions. If specified, also set `PersistentStorePool`.
  - `replicated`: Same as memory, but session data is replicated across the clustered servers.
  - `cookie`: All session data is stored in a cookie in the user's browser. The use of cookies to store session data introduces additional limitations. See the discussion on deployment in this chapter.
  - `replicated_if_clustered`: If the Web application is deployed to clustered servers, use replicated. Otherwise, use memory. This value is the default.

- `PersistentStoreDir`—If the `PersistentStoreType` is `file`, this property specifies either an absolute path of the directory to use or a path relative to a WebLogic Server temporary directory. The system administrator must manually create the specified directory and set appropriate ownership and permissions for the directory.
- `PersistentStorePool`, `PersistentStoreTable`, `JDBCConnectionTimeoutSecs`—If the `PersistentStoreType` is `jdbc`, these properties specify the name of the JDBC Connection Pool, the name of the table in which to store sessions, and the time WebLogic Server waits to time out a JDBC connection. The pool has no default value and must be specified when using JDBC. By default, the table is named `wl_servlet_sessions`; use this property to override the default. The timeout defaults to 120 seconds.
- `CacheSize`—If the `PersistentStoreType` is `file` or `jdbc`, this parameter specifies how much memory to dedicate to cache session objects. WebLogic Server can access cached sessions much faster than if the server must read the session object from a file or database. However, under some circumstances, caching can degrade performance significantly (i.e., thrashing of virtual memory). If set to 0, caching is disabled. The default is 256 sessions cached. [0..`Integer.MAX_VALUE`; default 256].
- `PersistentStoreCookieName`—If the `PersistentStoreType` is cookie, this property names the cookie WebLogic Server uses to store session data.

Specify session parameters in the session descriptor as described in the section "Configure Sessions" earlier in this chapter.

Special Considerations for In-Memory Replication

There are a number of special considerations for using in-memory replication of session state:

- In a single server instance, nonserializable data placed into HTTP sessions works fine. Unfortunately, in-memory replication does not work with nonserializable data. You'll incur many debugging headaches, only to discover that a portion of the data is not being replicated.
- Large, complex objects bring down the server performance. There is a good amount of overhead for serializing/deserializing the data, in addition to the network costs.
- Don't put hash tables, vectors, and so forth in the session because WebLogic Server cannot detect changes to objects in them. WebLogic Server ends up replicating the whole object, even though you changed just one part.
- It is a good practice to make all custom objects implement the serializable interface.
- Data that doesn't change should be made static so that it is not replicated.

Best Practices for Servlets

A number of best practices enable your use of advanced servlet features to be as successful as possible.

Limit the Size of Sessions

If your application protects sessions, the size of the session becomes important to servlet performance. WebLogic Server takes much more time to save larger sessions, so take pains to limit the amount of information stored in sessions.

Be Smart About Session State

If you are using in-memory replication or any other mechanism to protect your session state, it is important to be smart about how you treat the session state objects. In order to maximize efficiency, you want to make sure that session information is updated only when needed. Minimizing the changes that are made to the session state minimizes the number of updates either to the database or to the other nodes in the cluster.

WebLogic Server is smart about what it needs to replicate for in-memory replication. WebLogic Server monitors the session object to see what objects are placed in it, so it is important to be smart about how you treat session state objects. In order to maximize efficiency, minimize the updates to the database or secondary server.

WebLogic Server only transmits updates to the database or secondary server. To take advantage of this, you should make session information stored in the session object as granular as possible. It is better to store many small objects than to store one large object, such as a hash table or vector. If you change the state of a complex object that is already in the session, you must use the `setAttribute` again to notify WebLogic Server of the change.

If cached information is stored in the user session, consider writing the cache object so that it does not serialize. WebLogic Server does not transmit objects that do not serialize to databases or secondary servers. Your application can cache expensive objects in sessions without imposing performance-sapping overhead. If a server instance fails, the application can recompute the cached values.

Finally, only include necessary information in sessions. If data does not change across user sessions or multiple accesses, put data in static variables. Static variables are not shared across cluster nodes.

Check Serializable Status of Session Objects

When WebLogic Server passivates sessions, it serializes session objects to transmit them to secondary servers or to a database. Check that all objects added to a session are serializable (unless you do not want the information persisted). Objects that are not serializable are lost during passivation. This loss results in disappearing data that can be difficult to trace.

Persist Important Information in a Database

Session objects are appropriate for transient data that is specific to a user session. This information may include navigation state and items in a shopping cart. You should not expect that session objects are available for very long periods of time. Instead, for information that you expect to store for long periods, your best strategy is to store information about the user in the database. A database is the best location for storing long-term data.

Set Session Timeouts Aggressively

Sessions consume resources in WebLogic Server. If the timeout is set too large, inactive sessions can become a drag on WebLogic Server performance. For that reason, set your inactive intervals to be as short as possible.

Activate URL Rewriting and Encode URLs

Many users configure Web browsers to refuse cookies. For this reason, you should always enable URL rewriting in your servlets. URL rewriting ensures that you maintain compatibility and the highest level of usability for all your users.

Servlet/JSP developers should always use the URL encoding methods when embedding URLs in their HTML. Careful encoding ensures that WebLogic Server can properly track user sessions using URL rewriting. However, encode only URLs referring back to the Web application. If you pass session data to external servers, you may make your session IDs vulnerable to attack.

Use In-Memory Replication for Protecting Sessions

WebLogic Server provides reliable and high-performance replication of user sessions. Start with in-memory replication and only switch to unprotected or to protection using a database or filesystem to meet specific needs.

Increase reliability even further using replication groups. Replication groups can be used ensure that primary and secondary server instances are not on the same machine or even in the same data center.

Use Sessions to Mark Login Status

Use sessions to mark whether a user is logged in. Create a new session only on login and use sessions as a marker to determine whether a user has logged in. When a user logs out, invalidate his session. Sessions help ensure that an attacker cannot work around security measures by not logging in.

Use filters to apply security checks and configure the application of the security filter in your `web.xml` deployment descriptor.

Set Cookie Domains

Some browsers do not send cookies received from HTTP responses with HTTPS requests, or vice versa, because HTTP and HTTPS services are hosted on separate ports. As a result, Web applications may lose cookies and sessions identified using cookies. To ensure that cookies work across HTTP and HTTPS requests, set the cookie domain.

Set the cookie domain for custom cookies using the `setDomain()` method. To ensure that WebLogic maintains sessions, set the `CookieDomain` session parameter:

```
<session-descriptor>
  <session-param>
    <param-name>CookieDomain</param-name>
    <param-value>mydomain.com</param-value>
  </session-param>
</session-descriptor>
```

Using Cookies to Track Users

You may want to use cookies to store user identification information and make it easier for users to log in. However, you should remember that cookies are not particularly secure or a reliable form of permanent storage. Use the following guidelines for tracking users with cookies:

- Do use cookies to make your application more convenient. Provide an option to prepopulate a user's login ID. Provide nonsensitive user-specific information even before login.
- Do not assume that the information in a cookie is correct. Multiple users may share a machine, or a single user may have multiple accounts. Always allow users to sign in as someone else.
- Do not assume that the user is on her own machine. A user may access your application from a shared machine. Ask the user whether your application should remember the user's ID.
- Always require passwords before providing sensitive information.

Use Filters to Improve Code Reuse

Filters encourage code reuse because a filter can add functionality to a servlet without modifying servlet code. This aspect is useful when the servlet code is difficult to modify. Filters are even more useful when functionality can be applied to multiple servlets or even across Web applications. Use filters to apply security policies, cache unchanged information, transform XML output for a specific target browser, or dynamically compress output.

Override HttpResponseWrapper Methods

Use filters with subclasses of the `HttpResponseWrapper` to postprocess servlet responses. If your subclass of `HttpResponseWrapper` overrides the output buffer, be sure to override all the methods that access the output buffer. In particular, override `flushBuffer()`, `reset()`, and `resetBuffer()` methods.

Resources

The working body that defines the HTTP standard is the World Wide Web Consortium or W3C. The W3C homepage is at *http://www.w3c.org*.

The specification for HTTP is available as an RFC. HTTP 1.1 is available at *ftp://ftp.isi.edu/in-notes/rfc2616.txt*.

The Java Community Process page for JSR 000053 provides the final specification for the Servlet 2.3 specification: *http://www.jcp.org/aboutJava/communityprocess/final/jsr053*.

To read the javadoc for the Servlet classes, refer to *http://java.sun.com/products/servlet/2.3/javadoc*.

Full documentation of the deployment descriptor used to configure WebLogic Server (weblogic.xml) is available at *http://edocs.bea.com/wls/docs81/webapp/weblogic_xml.html*.

The developers at Sun have created a number of example filters. Among these filters are a filter to transform XML using XSL-T and a filter to compress servlet output. See *http://java.sun.com/products/servlet/Filters.html*.

To achieve the highest degree of reliability with in-memory replication, use the BEA feature called *Replication Groups* to customize how WebLogic Server selects secondary servers for replication. Documentation is available at *http://edocs.bea.com/wls/docs81/cluster/failover.html*.

Putting It All Together

Most Web applications have a need to store information from one request to another. WebLogic Server provides support for servlet standards to provide session and cookie storage mechanisms to applications. WebLogic Server goes beyond the basics of the specification with support for cluster-

ing. Clustered instances of WebLogic Server can provide high-performance and highly reliable applications and maintain a positive user experience.

Also, tools that improve the organization of code can benefit programmer productivity software quality. Filters provide just this kind of tool. Using filters, developers can declaratively apply a feature to components of a Web application.

With the powerful tools of filters, cookies, reliable sessions, and the best practices in applying these elements, you should have the skills you need to write enterprise-strength servlets. Next, you will learn about the other major presentation technology, JavaServer Pages.