Chapter 5: Servlet Sessions

In this chapter

- What Is Session Tracking?
- Using Hidden Form Fields
- Working with Cookies
- URL Rewriting
- Session Tracking with the Servlet API

What Is Session Tracking?

Session tracking is the capability of a server to maintain the current state of a single client's sequential requests. The HTTP protocol used by Web servers is *stateless*. This means that every transaction is autonomous. This type of stateless transaction is not a problem unless you need to know the sequence of actions a client has performed while at your site.

For example, an online video store must be able to determine each visitor's sequence of actions. Suppose a customer goes to your site to order a movie. The first thing he does is look at the available titles. When he has found the title he is interested in, he makes his selection. The problem now is determining who made the selection. Because each one of the client's requests is independent of the previous requests, you have no idea who actually made the final selection.

Note

You could use HTTP authentication as a method of session tracking, but each of your customers would need an account on your site. This is fine for some businesses, but would be a hassle for a high-volume site. You probably could not get every user who simply wants to browse through the available videos to open an account.

In this chapter, you will look at several different ways to determine the actions that a particular client has taken. You will examine hidden form fields, cookies, URL rewriting, and the built-in session tracking functionality found in the servlet API.

Using Hidden Form Fields

Using hidden form fields is one of the simplest session tracking techniques. Hidden form fields are HTML input types that are not displayed when read by a browser. The following sample HTML listing includes hidden form fields:

```
<HTML>
<BODY>

<FORM ACTION="someaction" METHOD="post">

<INPUT TYPE="hidden" NAME="tag1" VALUE="value1">
```

```
<INPUT TYPE="hidden" NAME="tag2" VALUE="value2">

<INPUT TYPE="submit">

</FORM>

</BODY>
</HTML>
```

When you open this HTML document in a browser, the input types marked as hidden will not be visible. They will, however, be transmitted in the request.

Let's create a simple example that shows how this technique works. You'll create a servlet that can service both `POST` and `GET` methods. In the `doGet()` method, you'll build a form that contains hidden fields and an action that points to the servlet's `doPost()` method. The `doPost()` method will then parse the hidden values sent in the request and echo them back to the client. The example is found in Listing 5.1.

Listing 5.1 `HiddenFieldServlet.java`

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

public class HiddenFieldServlet extends HttpServlet {

 public void init(ServletConfig config)
  throws ServletException {

  super.init(config);
 }

 //Process the HTTP Get request
 public void doGet(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {

  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<html>");
  out.println("<head><title>HiddenFieldServlet" +
   "</title></head>");
  out.println("<body>");

  // Create the Form with Hidden Fields
  out.println("<FORM ACTION=" +
   "\"/djs/servlet/HiddenFieldServlet\" METHOD=\"POST\">");

  // These values would be uniquely generated
  out.println("<INPUT TYPE=\"hidden\" NAME=" +
   "\"user\" VALUE=\"James\">");

  out.println("<INPUT TYPE=\"hidden\" NAME=" +
   "\"session\" VALUE=\"12892\">");

  // These are the currently selected movies
```

```
  out.println("<INPUT TYPE=\"hidden\" NAME=" +
   "\"movie\" VALUE=\"Happy Gilmore\">");

  out.println("<INPUT TYPE=\"hidden\" NAME=" +
   "\"movie\" VALUE=\"So I Married an Axe Murderer\">");

  out.println("<INPUT TYPE=\"hidden\" NAME=" +
   "\"movie\" VALUE=\"Jaws\">");

  out.println("<INPUT TYPE=\"submit\" VALUE=" +
   "\"Submit\">");
  out.println("</FORM>");

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

 //Process the HTTP Post request
 public void doPost(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {

  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<html>");
  out.println("<head><title>HiddenFieldServlet" +
   "</title></head>");
  out.println("<body>");

  // Get the hidden inputs and echo them
  String user = request.getParameter("user");
  String session = request.getParameter("session");

  out.println("<H3>" + user +
   ", the contents of your Shopping Basket are:</H3><BR>");

  String[] movies = request.getParameterValues("movie");

  if ( movies != null ) {

   for ( int x = 0; x < movies.length; x++ ) {

    out.println(movies[x] + "<BR>");
   }
  }

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

 //Get Servlet information
 public String getServletInfo() {

  return "HiddenFieldServlet Information";
 }

}
```

When you have this servlet installed, open your browser to the servlet's URL. The URL on my local box

is listed as follows:

```
http://localhost/djs/servlet/HiddenFieldServlet
```

When the servlet is loaded, you should only see a Submit button. If you view the current HTML source, you will see a listing similar to this snippet:

```
<html>
<head><title>HiddenFieldServlet</title></head>
<body>
<FORM ACTION="/djs/servlet/HiddenFieldServlet" METHOD="POST">
<INPUT TYPE="hidden" NAME="user" VALUE="James">
<INPUT TYPE="hidden" NAME="session" VALUE="12892">
<INPUT TYPE="hidden" NAME="movie" VALUE="Happy Gilmore">
<INPUT TYPE="hidden" NAME="movie" VALUE="So I Married an Axe Murderer">
<INPUT TYPE="hidden" NAME="movie" VALUE="Jaws">
<INPUT TYPE="submit" VALUE="Submit">
</FORM>
</body></html>
```

Notice the hidden fields. Now click the Submit button. The form invokes the `doPost()` method of the `HiddenFieldServlet`. This method parses the hidden fields out of the request and displays them in a "shopping cart" listing. Figure 5.1 shows the results of the `HiddenFieldServlet`'s `doPost()` method.

Figure 5.1
Output of HiddenFieldServlet.

You can see that hidden form fields have their advantages. They are easy to implement and are supported by most browsers. This technique also has its disadvantages. The hidden fields must be created in a particular sequence. You are not able to click the Back button on your browser without losing the additional fields added to the current page. You are also restricted to dynamically generated documents.

Working with Cookies

One of the more elegant solutions to session tracking is the use of persistent cookies. Netscape first introduced cookies in one of the company's first versions of Netscape Navigator.

A *cookie* is a keyed piece of data that is created by the server and stored by the client browser. Browsers maintain their own list of unique cookies. This makes cookies a very viable solution for session

The Servlet API provides built-in support for cookies. It does this through the use of the `Cookie` class and the `HttpServletResponse.addCookie()` and `HttpServletRequest.getCookies()` methods.

The `Cookie` class encapsulates persistent cookies as defined by RFC 2109. The prototype for the `Cookie`'s constructor takes a `String` representing the unique name of the cookie and a `String` representing the value of the cookie, and it is listed as follows:

```
public Cookie(String name, String value)
```

The `Cookie` class also provides accessors used to get and set the values of the cookie. Listing 5.2

contains an example of using cookies to perform session handling.

Listing 5.2 `CookieServlet.java`

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

public class CookieServlet extends HttpServlet {
//Initialize global variables

 public void init(ServletConfig config)
  throws ServletException {

  super.init(config);
 }

 private String getCurrentUser(String value) {

  String userName = new String("");

  // This would normally be a Select from a database or
  // other storage area.
  if ( value.equals("564XX892") ) {

   userName = new String("Bob");
  }
  return userName;
 }

 //Process the HTTP Get request
 public void doGet(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {

  // Get the list of Cookies stored in the request
  Cookie[] cookieList = request.getCookies();
  String user = null;
  String responseString = null;

  if ( cookieList != null ) {

   // Cookies found, let's get the session id
   for ( int x = 0; x < cookieList.length; x++ ) {

    String name = cookieList[x].getName();

    if ( name.equals("session_id") ) {

     // Get the user based on the session id
     user = getCurrentUser(cookieList[x].getValue());
     break;
    }
   }
  }

  if ( user == null ) {
```

```
   // Let's create a cookie that represents a unique
   // session id.
   response.addCookie(new Cookie("session_id", "564XX892"));
   responseString = new String("Welcome to our site, " +
    "we have created a session for you.");
  }
  else {

   responseString = new String("Hello : " + user);
  }

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

  out.println("<html>");
  out.println("<head><title>CookieServlet</title></head>");

  out.println("<body>");

  out.println(responseString);

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

 //Get Servlet information
 public String getServletInfo() {

  return "CookieServlet Information";
 }

}
```

Every time the `CookieServlet` services a request, it checks for cookies in the `HttpServletRequest`. It does this by calling the `HttpServletRequest.getCookies()` method. If the request does contain cookies, the servlet will iterate over the list of cookies looking for a cookie with the name `session_id`.

If the request contains no cookies or the list of cookies does not contain a cookie named `session_id`, you create one and add it to the response. The code snippet that does this is listed as follows:

```
response.addCookie(new Cookie("session_id", "564XX892"));
```

Note

Cookies are stored in the response as HTTP headers. Therefore, you must add cookies to the response before adding any other content.

The best way to test this functionality is to open your browser to the `CookieServlet`. The first time it runs, you should get a response that says "Welcome to our site, we have created a session for you." After you get this message, click the Refresh button. You should see a new response that says "Hello : Bob." The servlet can now identify the user "Bob" by the session ID stored as a cookie.

Note

If you have trouble running this example, make sure the use of cookies is enabled in your browser.

URL Rewriting

If your browser does not support cookies, URL rewriting provides you with another session tracking alternative. *URL rewriting* is a method in which the requested URL is modified to include a session ID. There are several ways to perform URL rewriting. You are going to look at one method that is provided by the Servlet API. Listing 5.3 shows an example of URL rewriting.

Listing 5.3 `URLRewritingServlet.java`

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

public class URLRewritingServlet extends HttpServlet {
//Initialize global variables

 public void init(ServletConfig config)
  throws ServletException {

  super.init(config);
 }

 //Process the HTTP Get request
 public void doGet(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {

  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<html>");
  out.println("<head><title>URL Rewriting</title></head>");
  out.println("<body>");

  // Encode a URL string with the session id appended
  // to it.
  String url = response.encodeRedirectURL(
   "http://localhost:8000/servlet/checkout?sid=5748");

  // Redirect the client to the new URL
  response.sendRedirect(url);

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

 //Get Servlet information
```

```
 public String getServletInfo() {

  return "URLRewritingServlet Information";
 }

}
```

This servlet services a `GET` request and redirects the client to a new URL. This new URL has the string `sid=5748` appended to it. This string represents a session ID. When the servlet that services the redirection receives the request, it will be able to determine the current user based on the appended value. At that point, the servlet can perform a database lookup on the user and her actions based on this ID.

Two methods are involved in this redirection. The first is `HttpServletResponse.encodeRedirectURL()`, which takes a `String` that represents a redirection URL and encodes it for use in the second method. The second method used is the `HttpServletRequest.sendRedirect()` method. It takes the `String` returned from the `encodeRedirectString()` and sends it back to the client for redirection.

The advantage of URL rewriting over hidden form fields is the capability to include session tracking information without the use of forms. Even with this advantage, it is still a very arduous coding process.

Session Tracking with the Servlet API

The Servlet API has its own built-in support for session tracking. The `HttpSession` object provides this functionality. In this section, I focus on four of the `HttpSession`'s session tracking methods.

The first method is the `setAttribute()` method. The `setAttribute()` method binds a name/value pair to store in the current session. If the name already exists in the session, it is replaced. The method signature for `setAttribute()` is listed as follows:

```
public void setAttribute(String name, Object value)
```

The next method is the `getAttribute()` method, which is used to get an object that is stored in the session. The `getAttribute()` method takes a string representing the name that the desired object is bound to. Its signature is listed as follows:

```
public Object getAttribute(String name)
```

The third session method returns an array of the current bound names stored in the session. This method is convenient if you want to remove all the current bindings in a session. Its signature is listed as follows:

```
public String[] getAttributeNames()
```

The last session method is the `removeAttribute()` method. As its name suggests, it removes a binding from the current session. It takes a string parameter representing the name associated with the binding. Its method signature is listed as follows:

```
public void removeAttribute(String name)
```

Now that I have discussed the `HttpSession` object, let's take a look at an example of how to use it. In this example, you will service a request that contains a list of movies to add to a user's account. You will then parse the submitted list, add it to the customer's session, and redisplay it for approval. When the customer approves the list, they will click the Proceed to Checkout button to commit the transaction. Listing 5.4 contains the source for this example.

Listing 5.4 `HttpSessionServlet.java`

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

public class HttpSessionServlet extends HttpServlet {

 public void init(ServletConfig config)
  throws ServletException {

  super.init(config);
 }

 //Process the HTTP Get request, this method
 // will handle the checkout
 public void doGet(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {

  String[] movies = null;

  // Get a handle to the HttpSession Object
  // if there is no session create one
  HttpSession session = request.getSession(true);

  // Get the movies list object bound to the
  // name "Movies"
  if ( session != null ) {

   movies = (String[])session.getAttribute("Movies");
  }

  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<html>");
  out.println("<head><title>Session Servlet</title></head>");
  out.println("<body>");

  // Iterate over the movies array, displaying the
  // current list of movies stored in the session
  out.println("<H2>Thank you for purchasing:</H2>");
  for ( int x = 0; x < movies.length; x++ ) {

   out.println(movies[x] + "<BR>");
  }
  out.println("</body></html>");
  out.close();
 }

 //Process the HTTP Post request
```

```
 public void doPost(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {

  // Parse the movies selected
  String movies[] = request.getParameterValues("Movies");

  // Get a handle to the HttpSession Object
  // if there is no session create one
  HttpSession session = request.getSession(true);

  // add the list of movies to the session
  // binding it to the String "Movies"
  if ( session != null ) {

   session.setAttribute("Movies", movies);
  }

  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<html>");
  out.println("<head><title>Session Servlet</title></head>");
  out.println("<body>");

  out.println("<H2>Contents of Shopping Cart</H2>");

  // Display the submitted movie array
  for ( int x = 0; x < movies.length; x++ ) {

   out.println(movies[x] + "<BR>");
  }
  // Create a form to submit an order
  out.println("<FORM action=/djs/servlet/HttpSessionServlet " +
   "METHOD=GET>");
  out.println("<input type=\"Submit\" name=\"add\" value=" +
   "\"Proceed to Checkout\"></FORM>");

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

 //Get Servlet information
 public String getServletInfo() {

  return "HttpSessionServlet Information";
 }
}
```

To invoke this servlet, you need to create an HTML file that will make a `POST` request containing a list of selected movies. The HTML file that contains this form is in Listing 5.5.

Listing 5.5 HtmlSessionServlet.html

```
<HTML>
<HEAD>
<TITLE>
Movie List
</TITLE>
</HEAD>
```

```
<BODY>
<H2>Select From Available Movies</h2>

<FORM ACTION=http://localhost/djs/servlet/HttpSessionServlet method=POST>

 <SELECT NAME="Movies" SIZE="5" MULTIPLE>
  <OPTION SELECTED>Air Force One</OPTION>
  <OPTION>Happy Gilmore</OPTION>
  <OPTION>So I Married an Axe Murderer</OPTION>
  <OPTION>Austin Powers</OPTION>
  <OPTION>Pure Luck</OPTION>
 </SELECT><BR>
 <INPUT TYPE="Submit" NAME="add" VALUE="Add Movies">

</FORM>

</BODY>
</HTML>
```

To see how this example works, load this HTML page in a browser. You should see a screen similar to Figure 5.2.

Figure 5.2
The Movie Selection List screen.

When this page is loaded, select a couple of the movies in the list and click the Add Movies button. You should now see a screen containing the list of movies you selected. Figure 5.3 displays an example of this output.

Figure 5.3
The Contents of Shopping Cart screen.

To understand how this first part works, you need to examine the `doPost()` method. This is the method that services the `POST` request sent by your HTML document.

The first thing the `doPost()` method does is get the list of submitted movies from the request. It then tries to get a reference to the `HttpSession` object stored in the `HttpServletRequest`. This is done by calling the `HttpServletRequest.getSession()` method. The code snippet that performs this is listed in the following:

```
// Get a handle to the HttpSession Object
// if there is no session create one
HttpSession session = request.getSession(true);
```

The `getSession()` method takes one parameter. This parameter is a Boolean value that, if `true`, tells the method to create an `HttpSession` if one doesn't exist.

When you have a reference to the `HttpSession` object, you can add your movie list to it. You do this by calling the `HttpSession.setAttribute()` method, passing it the name `"Movies"` and the object to be bound to it: `movies`. The movie list is now stored in the client's session. The last thing you do in the `doPost()` method is redisplay the list of selected movies and ask the user to click Proceed to Checkout.

Note

Sessions do expire. Therefore, you will need to consult your server's documentation to determine the length of time a session is valid.

Now you are going to look at the really cool part. Click the Proceed to Checkout button. You should see a screen similar to Figure 5.4, which tells you "Thank you for purchasing:" and displays the movies you selected.

Figure 5.4
The Thank You screen.

The request performed by this form simply calls the same servlet using the `GET` method. If you look at the URL your browser now points to, you will notice there is no movie data encoded in the URL string.

Look at the `doGet()` method to see exactly how this is done. The first thing you do is get a reference to the `HttpSession` object, which is done exactly as before with the `getSession()` method. When you have a reference to the session, you can get the list of movies stored in the session. You do this by calling the `HttpSession.getAttribute()` method, passing it the name bound to the `movies` object. The following code snippet shows how this is done:

```
// Get the movies list object bound to the
// name "Movies"
if ( session != null ) {

movies = (String[])session.getAttribute("Movies");
}
```

Note

Make sure that you downcast your stored object back to its original type. While in the `HttpSession`, it is stored as an object.

When you have the list of movies, thank the customer for the purchase and redisplay the list of ordered movies. That is all there is to it. As you have seen, the Servlet API provides you with a very elegant and simple-to-use method of maintaining persistent sessions.

Summary

In this chapter, we covered several methods that you can integrate into your servlets to handle persistent sessions. We talked about the hidden form fields, persistent cookies, URL rewriting, and the Servlet API's built-in session handling support. At this point, you should be able to examine your session

handling requirements and determine which method satisfies your needs. You should also be able to implement a session handling solution based on your decision.

In the next chapter, we will look at communication between Java applets and servlets using a method call HTTP tunneling.