Building servlets with session tracking

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial tips

Should I take this tutorial?

This tutorial teaches techniques for building Internet applications using servlet and JSP technology. A key point is to enable session handling, so the servlet knows which user is doing what. The tutorial shows a URL bookmarking system in which multiple users access a system to add, remove, and update an HTML listing of bookmarks. The servlet uses JSP technology to handle the user interaction.

You should be familiar with Java programming concepts such as the structure of the language (packages, classes, and methods) and how object-oriented programming differs from procedural programming. Prior experience writing servlets is helpful. If you need a refresher on servlets, see another dW tutorial, " *Building Java HTTP servlets* ."

To write the servlets described in this tutorial, you simply need an editing environment. This can be as basic as an operating system editor. In a development environment, many people use IDEs because they have debuggers and other features specific to writing and testing code.

To compile the servlets, you'll need the Java compiler (`javac.exe`). You'll also need the Java server development kit (JSDK), which is typically an extension to the JDK. Most servers support servlets based on the JSDK Version 2.1; many now support the latest release, JSDK 2.2. Once you have written your servlets, you can test them with the utility included in the JSDK. The JSDK 2.0 has a utility called servletrunner, while JSDK 2.1 comes with a small HTTP server. JSP technology and servlets form a part of the Java 2 Platform Enterprise Edition, which supports and provides ready-built functionality.

To execute the servlets, you'll need an application server, such as WebSphere, or an HTTP server with a servlet engine installed. HTTP servers are usually included with the application servers. Most HTTP servers that are not included in an application server require installation of an additional servlet engine such as Apache JServ.

Navigation

Navigating through the tutorial is easy:

* Select Next and Previous to move forward and backward through the tutorial.
* When you're finished with a section, select the next section. You can also use the Main and Section Menus to navigate the tutorial.
* If you'd like to tell us what you think, or if you have a question for the author about the content of the tutorial, use the Feedback button.

Getting help

For questions about the content of this tutorial, contact the author, Jeanne Murray, at *jeannem@us.ibm.com* . If you have problems running the servlets, a direct contact with Barry Busler, at *barrybusler@us.ibm.com* , will get you a faster answer.

Jeanne Murray is on the developerWorks staff, a perch from which she cajoles others to write servlets for tutorials. Thanks to Barry Busler for his contributions. Jeanne's been with IBM in Research Triangle Park, NC, for 12 years, where she's worked on software design and development projects ranging from mainframes to miniature devices.

Section 2. First, the basics of session tracking

Why do you need session tracking?

Session tracking makes you a polite host or hostess - - it keeps you from having to ask "oh, is it you again?" every time you receive a request. This is especially important in today's e-commerce Web apps; a Web shopping cart, for example, needs to accurately reflect a user's choices at every step in the process.

Internet communication protocols consist of two types: stateful and stateless. The server associates a state with a connection; it remembers who the user is and handles all user requests.

* Stateful protocols, such as telnet and FTP, can process multiple operations before closing a connection. The server knows that all requests came from a single person.
* HTTP is a stateless protocol, which means every time a client asks for a Web page, the protocol opens a separate connection to the server, and the server doesn't know the context from one connection to the next. Each transaction is a single isolated transaction. Imagine if every time you added an item to your shopping cart, it forgot all your previous items, and couldn't even remember whose cart it was in the first place.

Some Web servers support persistent HTTP connections, meaning they keep a socket open for multiple client requests. But this isn't exactly the easy way out; rather, you'll have to go through all sorts of contortions to make sure the Web server supports them, that you comply with the number of connections allowed on a single socket (or change it if you can find the configuration setting), and generally get yourself all wrapped up in exception testing till the day dawns dark and dreary.

The session tracking API

Techniques for identifying a session and associating data with it, even over multiple connections, include using cookies, URL rewriting, and hidden form fields. These techniques, which are explained more fully on the next three panels, do require additional work for managing the information involved.

To eliminate the need for manually managing the session information within your code (no matter what session tracking technique you're using), you use the HttpSession class of the Java Servlet API. The HttpSession interface allows servlets to:

* View and manage information about a session
* Make sure information persists across multiple user connections, to include multiple page requests as well as connections

Session tracking techniques: Cookies

Cookies are probably the most common approach for session tracking. Cookies store information about a session in a human-readable file on the client's machine. Subsequent sessions can access the cookie to extract information. The server associates a session ID from the cookie with the data from that session. This gets more complicated when there are multiple cookies involved, when you have to decide when to expire the cookie, and when you have to generate so many unique session identifiers. Additionally, a cookie cannot grow more than 4K in size, and no domain can have more than 20 cookies.

Cookies pose some privacy concerns for users. Some people don't like the fact that a program can store and retrieve information about their habits. In addition, people are wary that sensitive information -- such as a credit card number -- could be stored in a cookie. *We* know this isn't good programming practice, but as for those other guys out there...) Unfortunately, it is easy to use cookies inappropriately.

As a result, some users disable cookies or delete them altogether. Therefore, you should not depend on them as your sole mechanism for session tracking.

Session tracking techniques: URL rewriting

URL rewriting is used to append data on the end of each URL that identifies the session. The server associates the identifier with data it has stored about the session.

The URL is constructed using an HTTP GET. It may include a query string containing pairs of parameters and values. For example: `http://www.server.com/getPreferences?uid=username&bgcolor=red&fgcolor=blue`. URLs can get quite lengthy.

URL rewriting is a good solution, especially if users have disabled cookies, in which case it becomes a most excellent solution. But be sure to consider the following:

* You have to be sure to append the information to every URL that references your site.
* Appending parameters brings up privacy issues; you may not want the actual data you are tracking to be visible.
* There's a loophole with this technique: users can leave the session and come back using a bookmark, in which case your session information is lost.

These problems are not insurmountable, just tedious to tackle.

Session tracking techniques: Hidden form fields in HTML

Hidden form fields store information about the session. The hidden data can be retrieved later by using the HTTPServletRequest object. When a form is submitted, the data is included in the Get or Post.

Before you get excited though: form fields can be used only on dynamically generated pages, so their use is limited. And there are security holes: people can view the HTML source to see the stored data.

The HttpSession object

No matter which technique(s) you've used to collect session data, you need to store it somewhere. In this tutorial, we use the `HttpSession` object to store the session data from our servlet. This object matches a user to session data stored on a server. The basic steps for using the `HttpSession` object are:

* Obtain a session object
* Read or write to it
* Either terminate the session by expiring it, or do nothing so it will expire on its own

A session persists for a certain time period, up to forever, depending on the value set in the servlet. A unique session ID is used to track multiple requests from the same client to the server.

Persistence is valid within the context of the Web application, which may be across multiple servlets. A servlet can access an object stored by another servlet; the object is distinguished by name and is considered bound to the session. These objects (called attributes when you set and get them) are available to other servlets within the scope of a request, a session, or an application.

Key methods used in HttpSession

`HttpSession` maintains and accesses the session information within a servlet so you don't have to manipulate the information directly in your code.

Important methods used in `HttpSession` include:

* `isNew()`. Returns true if the client doesn't yet know about the session. If the client has disabled cookies, then a session is new on each request.
* `getId()`. Returns a string containing the unique identifier assigned to this session. Useful when using URL rewriting to identify the session.
* `setAttribute()`. Binds an object to this session, using the name specified. (Note: this replaces the `setValue()` method of JSDK 2.1.)
* `getAttribute()`. Returns the object (with the specified name) bound in this session. (Note: this replaces the `getValue()` method of JSDK 2.1.)
* `setMaxInactiveInterval()`. Specifies the time between client requests before the servlet invalidates this session. A negative time indicates the session should never timeout.
* `invalidate()`. Expires the current session and unbinds the object bound to it.

Section 3. Data in, data out: Using JSP technology

Handling client interaction with JSP technology

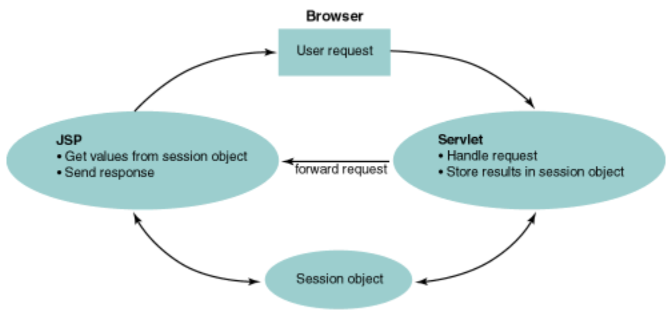
Now that you're familiar with session tracking and the HttpSession interface, we'll look at another important technology for building our bookmarking servlet: JavaServer Pages (JSP) technology.

JSP technology is used to manage the client interaction because it helps us separate the presentation logic from the application logic. JSP technology was developed to separate dynamic content from static HTML page design, meaning the page design can change without having to worry about changing the content. JSP technology can generate XML as easily as it can HTML, and the resulting file will send either XML or HTML to the client.

JSP technology from 10,000 feet

The contents of JSP files -- such as HTML tags, JSP tags, and scripts -- are basically translated into a servlet by the application server. This process translates both the dynamic and static elements declared within the JSP file into Java servlet code. The servlet generates a response and sends it back to the Web server.

JSP technology is a server-side technology, which means all this processing occurs in the server and is available to other components such as JavaBeans and Enterprise JavaBeans. In fact, servlets typically interact with JSP files using beans as intermediaries. The servlet does some computation, builds a bean with the results, and the JSP file does the formatting to present the information to the user. This is the cleanest separation between logic and presentation.



The basic steps for a servlet communicating with a JSP file to present information to a client are as follows:

* A request first goes to a servlet, which handles the request.
* The results are stored in a bean.
* The servlet forwards the request to a JSP file.
* The JSP file obtains data values from the bean and handles all presentation elements.

Storing the results in a bean makes them accessible to other JSP pages. An alternative approach is to store results in a session object, which is what we do in our bookmarking example in this tutorial. This makes the data specific to a user session, rather than globally available, which is appropriate for our application.

Now you know enough about JSP technology to be dangerous, er, that is, enough to understand this tutorial. For more indepth instruction, you can take another tutorial on developerWorks: " *Introduction to JavaServer Pages technology* ."

If we weren't just doing a demo...the role of JDBC

To keep this tutorial simple, we access a data file to read and update stored user information. In the real world, you would likely be using an external database to store your data. Then you would use JDBC (Java Database Connectivity) to access the data.

JSP files can call JDBC statements directly to access back-end systems. One issue that arises is that you don't want your Web page designers inadvertently modifying database access statements. Therefore, a better solution would be to create JavaBeans components to encapsulate the JDBC code. Also, JavaBeans components are a good alternative to isolating JDBC code that is repetitively executed.

Because JDBC is the standard way to access databases from servlets, we'll tell you a bit about it here. There is a comprehensive tutorial on this subject elsewhere on developerWorks (" *Java Database Connections* ") if you want to try it on your own.

JDBC from 10,000 feet

JDBC standardizes the mechanism for connecting to databases, the syntax for sending queries and committing transactions, and the structure for representing the results. The idea is, if your queries follow standard SQL syntax, you should be able to change database hosts, ports, and vendors with few changes to your code.

The basic steps for a servlet to access a relational database using JDBC are as follows:

* Load the class that references the appropriate JDBC driver.
* Specify the location of the database server.
* Get connected, which means pass the URL and the database username and password.
* Send, then execute, a query using SQL syntax.
* Process the results and return them to the browser. Numerous methods are available for accessing the results by row and column, and for dynamically discovering high-level information about a result (which is useful for ad hoc queries).
* Close the connection. Note that you only do this when you're *really* done; the overhead of opening connections time and time again will kill your performance, and reusing connections is an important technique for optimization. It is very important to remember to close your connection so that garbage collection will clean up the unused object. Sometimes databases have a limited number of connections available; you don't want any unused connections lying around.

Again, we are not using JDBC in this tutorial so that we may keep life simple. However, we've tried to structure the code and use naming conventions that make it easy for you to infer where you'd insert database functionality.

Section 4. Scenario: A URL bookmarking system

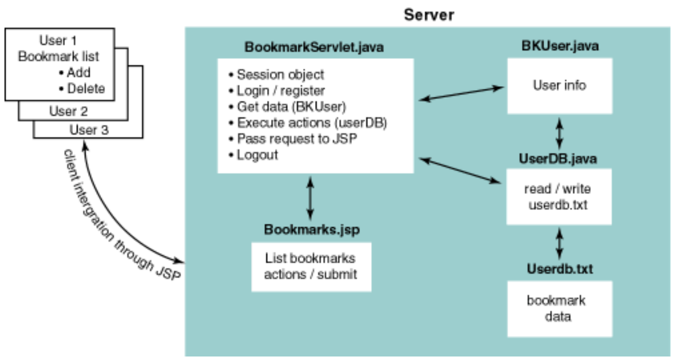
What the bookmarking system does

Our servlet demonstrates a URL bookmarking system in which multiple users access a system to add, remove, and update an HTML listing of bookmarks. The bookmarks are stored in a user information file. The servlet creates and tracks sessions and updates the bookmark file. It uses JSP technology to handle the user interaction.

The system consists of several servlets, all of which are described in detail in this tutorial (with the code):

* BookmarkServlet.java, which handles the user requests, performs the actions (add, delete, etc.), stores results in a session object, and passes the request to the JSP file
* BKUser.java, which manages the user information. This object manipulates information in the (pseudo) database format so it can be used by the Java code.
* UserDB.java, which, in combination with userdb.txt, stores the user information, to include user name and password and all URLs for their bookmarks. This is our simulated database.
* Bookmarks.jsp, which handles the user interaction by displaying data and providing the interface for the user actions.

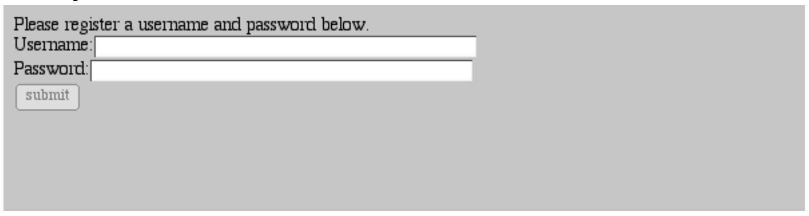
The components and the flow of information among them is shown here.



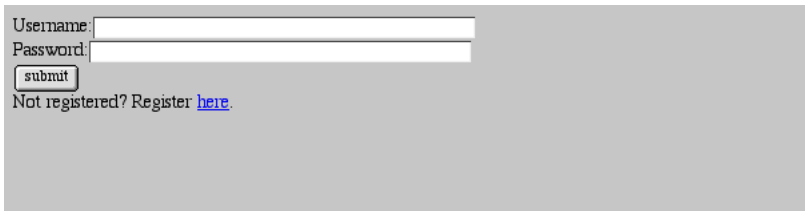
Tasks from the user's point of view

The user's tasks are to:

* Register with the system, as shown here.

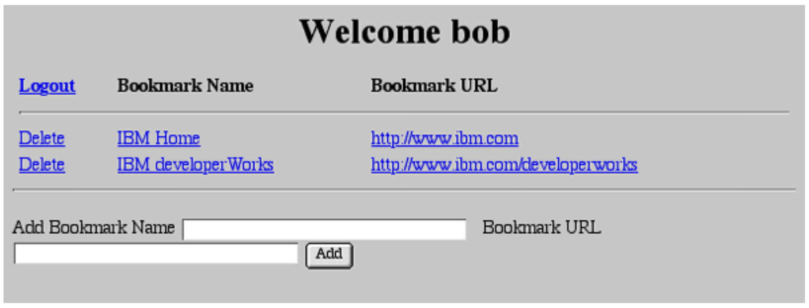

Please register a username and password below.
Username:
Password:

* Log in, as shown here.


Username:
Password:

Not registered? Register here.

* View, add, and delete bookmarks, as shown here.


Welcome bob

Logout	Bookmark Name	Bookmark URL
Delete	IBM Home	http://www.ibm.com
Delete	IBM developerWorks	http://www.ibm.com/developerworks

Add Bookmark Name Bookmark URL

Tasks from the servlet's point of view

The user has the easy part; the servlet does all the hard stuff. The servlet tasks are to:

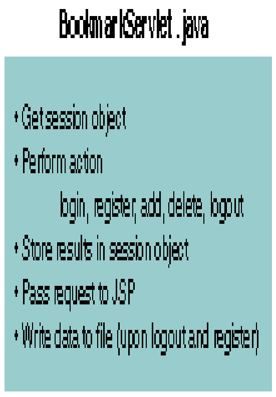
* Register users. Some will be new, others already known to the system. The servlet also validates user logins.
* Show each user his or her own bookmark list. The servlet keeps track of which user gets which list.
* Store changes to the lists when users make updates.
* Write the changes to the file when the user registers or logs out.
* Close the session when the user is no longer actively making updates.

On to the code...

The following sections walk you through the code. For the purposes of explaining the hows and whys, we've broken the code into chunks. (As a result, it may look as though all closing braces aren't matched up in the individual code snippets; this isn't truly the case.) You can view the code in its entirety in the *appendix* of this tutorial.

You may want to bring up the *servlet package documentation (Version 2.2)* in a secondary browser window as you go through the next sections.

Section 5. Handling requests: BookmarkServlet.java



BookmarkServlet.java

- Get session object
- Perform action
  - login, register, add, delete, logout
- Store results in session object
- Pass request to JSP
- Write data to file (upon logout and register)

This servlet's purpose in life

`BookmarkServlet.java` handles the requests and performs the actions, which include user registration, log in and log out, and the add and delete functions. It stores results in the session object and passes requests to the JSP file. Once a user logs out, the servlet writes the data to the userdb.txt file.

Declare a new class and initialize

The servlet starts with import statements, which give us access to other Java packages. Next, we need to declare a new class, BookmarkServlet, by extending the HttpServlet class (`javax.servlet.http.HttpServlet`). We also specify paths for the code, which makes it easier to maintain. (The paths we use here are specific to the Tomcat server we used for testing, which uses the default Sun `Web-inf/classes` directory.)

The `init()` method, which was new in JDSK 2.1, starts by calling the `super.init(ServletConfig)` method in order to register the ServletConfig for later use. This is typically done at the beginning of the `init()` method.

Next, we initialize UserDB, which we use to store the user and bookmark data. We will be reading the stored user information from a file called userdb.txt. (Note: This is where database connections would be opened and initialized if we were using JDBC.)

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

public class BookmarkServlet extends HttpServlet{

  private static final String pathtohtml="/examples/servlets/tut/";
  private static final String pathtojsp="/Web-inf/classes/";
  private static final String pathtoservlet = "";

public void init(ServletConfig cfig) throws ServletException{
  super.init(cfig);
  UserDB.initDB();
}
```

Set up the session object

Next we have the `doGet()` method, which passes GET requests to the `doPost()` method below it. This is a common technique in servlets; it ensures that if some program issues a POST request to this servlet (which could alter data), the servlet will enact the `doGet()`

method in response.

The `doPost()` method first sets up the session object, which will be used for all the actions. First we get the session associated with this client. If there is not one available, the true parameter on `getSession()` indicates to create a new one. Then, we determine what action is to be performed based on input to the servlet. The `getParameter() method` of the request tells us whether the user wants to log in, add a bookmark, or perform another operation. If we get invalid input, we redirect to an error page and exit the servlet.

```
 public void doGet(HttpServletRequest request,
                              HttpServletResponse response){
     this.doPost(request, response);
     }

 public void doPost(HttpServletRequest request,
                               HttpServletResponse response){

    HttpSession session = request.getSession(true);

    String action=request.getParameter("func");

    try{

       if(action==null){
         response.sendRedirect(pathtohtml+"error.html");
         return;
       }
```

Determine the action: Log in

The next set of statements are `if/else` statements, where we implement the action. If the user is logging in, the servlet gets the username and password through the BKUser interface. We use the `getParameter()` method on the request to the UserDB object.

If the user login is valid, the servlet stores the user information in the session object using the `setAttribute()` method. It then passes the request to the JSP file that provides the user interface.

If the login is not valid, the servlet sends the user to the login error page.

```
      if (action.equals("login")){

         BKUser user= UserDB.getUser(request.getParameter("username"),
                                                              request.getParameter("pas

         if(user!=null){
             session.setAttribute("user", user);
             passItOn(request, response, pathtojsp+"Bookmarks.jsp");
         }

         else{
             response.sendRedirect(pathtohtml+"error.html");
         }
      }
```

Determine the action: Add, delete bookmark

The next statements handle the actions for adding and deleting bookmarks. But before we can fulfill this request, we check to see if the user is already known to us. In both sets of statements, before we perform the action, we check to see if the session is a new one by using the `isNew()` method of the session object. If the user did indeed somehow get to the add or delete function without logging in first, we send them to the login page.

The servlet uses the BKUser interface to manage the user information, including the bookmark URL, which is represented by the href parameter. It then passes the information to the JSP file.

```
    else if (action.equals("add")){

           if(session.isNew()){
              response.sendRedirect(pathtohtml+"login.html");
           }
           else{
              BKUser user = (BKUser)session.getAttribute("user");
              user.addBookmark(request.getParameter("name"),
                                         request.getParameter("href"));
              session.setAttribute("user", user);
              passItOn(request, response, pathtojsp+"Bookmarks.jsp");
          }
    }

    else if (action.equals("delete")){

           if(session.isNew()){
                 response.sendRedirect(pathtohtml+"login.html");
           }
           else{
              BKUser user = (BKUser)session.getAttribute("user");
           user.delBookmark(request.getParameter("name"));
              session.setAttribute("user", user);
           passItOn(request, response, pathtojsp+"Bookmarks.jsp");
           }
     }
```

Determine the action: Register

This code handles the user registration, updating UserDB and setting up the user information in BKUser. It stores user information in the session object using the `setAttribute()` method.

```
     else if (action.equals("register")){

            UserDB.addUser(request.getParameter("username"),
                                  request.getParameter("password"));
            BKUser user= UserDB.getUser(request.getParameter("username"),
                                 request.getParameter("password"));

          if(user!=null){
                session.setAttribute("user", user);
                   passItOn(request, response, pathtojsp+"Bookmarks.jsp");
             UserDB.updateDB();
          }
          else{
            response.sendRedirect(pathtohtml+"error.html");
          }
      }
```

Determine the action: Logout

Here we determine the user has chosen to log out. We write to the userdb.txt file using the UserDB interface, then expire the session and redirect the user to the logout page. If we were truly connected to a database, then we would also close the database connection.

```
         else if (action.equals("logout")){
               UserDB.updateDB();
               session.invalidate();
               response.sendRedirect(pathtohtml+"logout.html");
            }
      }
      catch(Exception e){
         e.printStackTrace();
      }
    }
```

Enable forwarding to the JSP file

This last section of code enables forwarding request information to the JSP file. The `RequestDispatcher` method was introduced in SDK 2.1. It dispatches a request to either a static resource, such as an HTML page, or another dynamic resource, such as a servlet or JSP file. A request can either be forwarded or included, and the state can be made available to the dispatched resource.

In essence, the servlet cannot print any data before calling `forward()` because the target page will likely set one or more headers -- and this may not work if data has already been sent. (This also applies to the `include()` method. The page it includes cannot set any headers because the servlet calling the `include()` may have already printed data.)

When the `forward()` method returns, the calling servlet regains control. The output steam is then closed at this point.

```
 private void passItOn(HttpServletRequest request,
                              HttpServletResponse response, String nextpage)
     throws ServletException, IOException{
     RequestDispatcher disp = getServletContext().getRequestDispatcher(nextpage
     disp.forward(request, response);
     }

 public void destroy(){
     UserDB.closeDB();
     }
 }
```

Here's the code

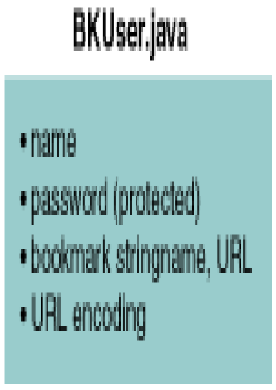
Here's the *BookmarkServlet.java code* .

You can also view the code listing in the *Appendix* of this tutorial.

If you have the ability to execute your servlet on a server, compile the code and put the class file in the servlet directory. Run the servlet using a URL with the following syntax. Remember that the class file name is case sensitive.

```
http://host/path/BookmarkServlet
```

Section 6. Manipulating user information: BKUser.java


BKUser.java

- name
- password (protected)
- bookmark stringname, URL
- URL encoding

This object's purpose in life

BKUser manages the user information and formats it so all parties involved can read it. For example, the data format housed in the userdb.txt file is unreadable by the BookmarkServlet object; BKUser takes care of encoding the URLs and sets up a hashtable. Both actions make it easier to parse the information.

Set up to read strings

The code starts with the housekeeping: import statements and class declaration. The java.util and java.net packages provide the enumeration and encoding functions needed for this object.

The `BKUser()` method is the constructor for the class. When a new BKUser object is created, this method initializesthe user information for the object by parsing the `dbstring`. We call it dbstring here as though we are accessing a database; in reality our system stores the data in a file (called userdb.txt). The add and delete actions are handled with `put()` and `remove()` methods.

```
import java.io.*;
import java.util.*;
import java.net.*;

public class BKUser{
      public String name;
      protected String password;
      public Hashtable bookmarks;

      public BKUser(String name, String password,
                           String dbstring){
            this.name=name;
            this.password=password;
            this.bookmarks=new Hashtable();
            if(dbstring!=null){
               this.readDBString(dbstring);
            }
      }

      public void addBookmark(String name, String href){
            bookmarks.put(name, href);
      }

      public void delBookmark(String name){
            bookmarks.remove(name);
      }
```

Format information for the data file

Here we convert the user information to the format for storage in the data file.

```
       public String getDBString(){
             String dbstring="";
             boolean firsttime=true;

             dbstring+=URLEncoder.encode(this.name) + " "
                                + URLEncoder.encode(this.password) + " ";
             for (Enumeration enum = bookmarks.keys() ; enum.hasMoreElements() ;
                   String bkname=(String)enum.nextElement();
                   String bkhref=(String)bookmarks.get(bkname);
                   if(firsttime){
                         dbstring+=URLEncoder.encode(bkname)+"!"
                                +URLEncoder.encode(bkhref);
                         firsttime=false;
                    }
                   else{
                         dbstring+="|"+URLEncoder.encode(bkname)+"!"
                                     +URLEncoder.encode(bkhref);
                   }
              }
              return dbstring;
        }
```

Extract user information

And here we get the user information from that data file format and decode it.

```
     public void readDBString(String dbstring){
        StringTokenizer sptok = new StringTokenizer(dbstring);
        this.name=URLDecoder.decode(sptok.nextToken());
        this.password=URLDecoder.decode(sptok.nextToken());
        StringTokenizer pptok = new StringTokenizer(sptok.nextToken(),"|");
        while (pptok.hasMoreTokens()) {
           StringTokenizer exctok = new StringTokenizer(pptok.nextToken(),"!");
           bookmarks.put(URLDecoder.decode(exctok.nextToken()),
                                URLDecoder.decode(exctok.nextToken()));
        }
     }
 }
```

Here's the code

Here's the *BKUser.java code* .

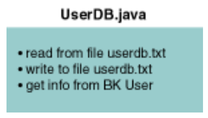
You can also view the code listing in the *Appendix* of this tutorial.

If you have the ability to execute your servlet on a server, compile the code and put the class file in the servlet directory. Run the servlet using a URL with the following syntax. Remember that the class file name is case sensitive.

```
http://host/path/BKUser
```

Section 7. Storing data: UserDB.java and userdb.txt

Purpose of these files



UserDB and userdb.txt form our "database." The UserDB object opens the data file and reads and writes the data; userdb.txt file is our data store.

The userdb.txt file has the following format.

```
username password mark1!url1|mark2!url2|...markn!urln
```

Each user's information is on a separate line. Here is an example for a user named Bob, who has the password pass.

```
bob pass IBM+Home!http%3A%2F%2Fwww.ibm.com|IBM+developerWorks!http%3A%2F%2Fwww.i
```

Make declarations

Here we declare the class UserDB and identify the data file, userdb.txt. We also enable the read and write functions.

```
 import java.io.*;
 import java.util.*;

 public final class UserDB{
   static final String dbfile="userdb.txt";
   static Vector users;

 public static void initDB(){
    users = new Vector();
    readFromFile(dbfile);
  }

  public static void closeDB(){
    writeToFile(dbfile);
    users=null;
  }

  public static void updateDB(){
    writeToFile(dbfile);
  }
```

Read the data

This function opens the data file and scans the user information into the vector `users`, which is of the type BKUser.

```
  public static synchronized void readFromFile(String filename){
      try{
        BufferedReader in = new BufferedReader(new FileReader(dbfile));
        String line;
        while((line=in.readLine())!=null){
          users.add(new BKUser(null, null, line.trim()));
        }
        in.close();
      }
      catch(Exception e){
        e.printStackTrace();
      }
  }
```

Write the data

This function writes the information to the file. All of the user information and bookmarks from the vector `users` is written to the file using the prescribed format.

```
username password mark1!url1|mark2!url2|...markn!urln
```

The `readFromFile()` and `writeToFile()` sections of code are synchronized to prevent multiple writings from running into each other.

```
   public static synchronized void writeToFile(String filename){
         try{
           PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(
           for (Enumeration enum = users.elements() ; enum.hasMoreElements() ; 
                out.println(((BKUser)enum.nextElement()).getDBString());
           }
           out.close();
         }
         catch(Exception e){
           e.printStackTrace();
         }
    }
```

Get information from BKUser

Here we retrieve user information from the database, and add a user to the database if needed.

```
    public static BKUser getUser(String username, String password){
        BKUser user=null;
        for (Enumeration enum = users.elements() ;
                                    enum.hasMoreElements() ; ) {
```

```
         user=(BKUser)enum.nextElement();

         if(user.name.equals(username)
           ampamp(password)){
           break;
         }
         else{
           user=null;
         }
      }
    return user;
  }

   public static void addUser(String username, String password){
      users.add(new BKUser(username, password, null));
   }
 }
```

Here's the code

Here's the *UserDB.java code* .

You can also view the code listing in the *Appendix* of this tutorial.

If you have the ability to execute your servlet on a server, compile the code and put the class file in the servlet directory. Run the servlet using a URL with the following syntax. Remember that the class file name is case sensitive.

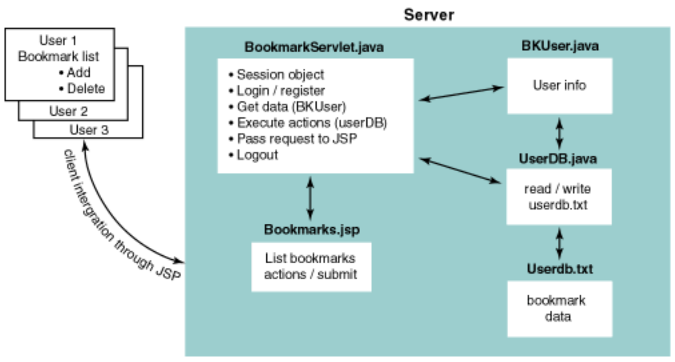
```
http://host/path/UserDB
```

Section 8. Enabling user interaction: Bookmarks.jsp

Showing it all to the user

The JSP file gets the information from the session object and presents it to the user. BookmarkServlet, through use of the BKUser and UserDB objects, has kept track of individual user's information; now it is the role of the JSP file to display it to the user. The JSP file also accepts and passes along the user requests for the actions such as add and delete.

We repeat the servlet and JSP interaction diagram here - - not to be redundant, but to reinforce the point.



Set the stage

The first lines of the JSP file import the necessary Java classes using JSP directives. The directive using the session attribute instructs the page to keep track of the user session. (A value of false would disable this.) A JSP declaration is also used for the BKUser object. Finally, we declare a constant for the URL to make the file easier to maintain.

The final statement in this section gets the user information for this session.

```
 <%@ page import="java.util.*" %>
 <%@ page import="java.net.*" %>
 <%@ page import="BKUser" %>
 <%@ page session="true" %>
 <%! private BKUser user; %>
 <%! private static final String servleturl =
                   "http://localhost:8080/examples/servlet/BookmarkServlet";
>
 <%

 user = (BKUser)session.getValue("user");
```

Determine user and session tracking information

This portion of the code inserts the username in the output to offer a personalized welcome. The other interesting code (as oppposed to the fairly uninteresting HTML tags) encodes the logout URL with the session tracking information.

```
 <html>
 <head>
 <title>Bookmarks</title>
 <body>
 <h1 align="center">Welcome <%= user.name %></h1>
 <table width="100%">
 <tr>
    <td><b><a href="<%= response.encodeURL(servleturl+"?func=logout")
>">Logout</a></b></td>
```

Display the bookmarks for user action

Next the code prints the bookmarks as rows in a table, then provides the user interface for deleting and adding bookmarks. The information passed to the server in the responses is encoded. This is necessary because the information could have spaces or other characters that could be misread by the server.

The rest of the code handles the submit action and closes the HTML tags to end the page.

```
    <td><b>Bookmark Name</b></td>
    <td><b>Bookmark URL</b></td>
 </tr>
 <tr>
    <td colspan="3"><hr></td>
 </tr>

 for (Enumeration enum = user.bookmarks.keys() ; enum.hasMoreElements() ; ) {

      String bkname=(String)enum.nextElement();
      String bkhref=(String)user.bookmarks.get(bkname);

      String deleteurl = response.encodeURL(servleturl +
&name="
                             + URLEncoder.encode(bkname));
 %>
 <tr>
    <td><a href="<%= deleteurl %>">Delete</a></td>
    <td><a href="<%= bkhref %>" target="_blank"><%= bkname
/td>
    <td><a href="<%= bkhref %>" target="_blank"><%= bkhref
/td>
 </tr>

 <%
 }
 %>
 </table>
 <hr>

 <form action="<%= response.encodeURL(servleturl) %>" method="post">
 <input type="hidden" name="func" value="add">
 Add Bookmark Name <input type="text" size="30" name="name">
 &nbsp;&nbsp;Bookmark URL <input type="text" size="30" name="href">
 <input type="submit" name="addsubmit" value="Add">
 </form>
```

```
</body>
</html>
```

Here's the code

Here's the *Bookmarks.jsp code* .

You can also view the code listing in the *Appendix* of this tutorial.

Section 9. Summary

Wrap up

The objective of this tutorial was to explain session tracking and how it can be used in servlets. We threw in a few bonus concepts, including JSP technology, which handled the user interaction, and a brief discussion of how you would use JDBC to access the data if it was in a database (instead of a flat file, such as we used here). You should be able to use the code in this tutorial to experiment further by registering new users and creating new bookmark lists for them. You can also alter the HTML page that the user sees by changing the JSP file.

Resources

The Java Servlet API is available on servers through a variety of industry partners and server vendors. The java.sun.com site lists *servers and tools* that support the API.

The *Java servlet package documentation (Version 2.2)* is the definitive source of package, interface, and class definitions.

To learn more about JSP technology, take the *Introduction to JSP technology tutorial* on developerWorks.

To learn more about JDBC technology, take the *JDBC tutorial* on developerWorks.

Developers are finding interesting ways to use servlets with XML. Read more in *Servlets and XML: Made for each other* on developerWorks.

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

For questions about the content of this tutorial, contact the author, Jeanne Murray, at *jeannem@us.ibm.com* . If you have problems running the servlets, a direct contact with Barry Busler, at *barrybusler@us.ibm.com* , will get you a faster answer.

Section 10. Appendix: Source code listings

BookmarkServlet.java

Here is the source for BookmarkServlet.java.

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

public class BookmarkServlet extends HttpServlet{

  //define constants for easy update of the servlet
  //if the locations of the resources are changed

  private static final String pathtohtml="/examples/servlets/tut/";
  private static final String pathtojsp="/Web-inf/classes/";

/**
 *Servlet begins with the init() method.  This method starts
 *off by calling the super.init(ServletConfig) method in order
 *to register the ServletConfig for later use.  This is
 *almost always done at the beginning of the init() method.
 *Next we initialize the users database.  This is where database
 *connections would be opened and initialized if we were using
 *JDBC.  In our case, we will be reading the stored user
 *information from a file (userdb.txt).  This will be
 *accomplished using a custom Java class (UserDB)
 *with synchronized methods.  If we were to use JDBC,
 *we would want to use connection pooling.
 */

public void init(ServletConfig cfig) throws ServletException{
  super.init(cfig);
  UserDB.initDB();
}

/**
 * Next we will have the doGet() method.  This method will
 * do nothing but pass GET requests to the doPost() method below.
 */

public void doGet(HttpServletRequest request,
                  HttpServletResponse response){
  this.doPost(request, response);
}


/**
 * Now we have the doPost() method.  This will be the most complex.
 * The doPost() method first sets up the session object
 * which will be used for any of the actions that will be performed.
 */
public void doPost(HttpServletRequest request,
                   HttpServletResponse response){


  // Get the session associated with this client.  If there is not one
```

```
    // available the getSession will create a new one.
    // We are telling it to do this with the "true" parameter.

    HttpSession session = request.getSession(true);

    // Determine what action is to be performed.
    String action=request.getParameter("func");

    try{
    // if we have gotten invalid input redirect to an error page
    // and exit the servlet
      if(action==null){
        response.sendRedirect(pathtohtml+"error.html");
        return;
      }

      /*
       * In this set of if else statements, we are determining what action
       * to perform based on input to the servlet.  The getParameter()
       * method of the request object is used to determine the input values.
       * In each if statement we check to see if  the session is a new
       * session (user's first request to server) using the isNew() method
       * of the session object.  We then perform
       * or a forward.
       */


      if (action.equals("login")){
        // If the user is logging in, we go get the information
        // (username, password, bookmarks) for the user id and password.

        BKUser user= UserDB.getUser(request.getParameter("username"),
                                    request.getParameter("password"));

        // If the user/pass is correct store the users information in
        // the session object using the setAttribute method. Then pass
        // off the request to a JSP that provides the user interface.
        if(user!=null){
          session.setAttribute("user", user);
          passItOn(request, response, pathtojsp+"Bookmarks.jsp");
        }

        // user/pass is incorrect; send the user to the login error page
        else{
          response.sendRedirect(pathtohtml+"error.html");
        }
      }

      else if (action.equals("add")){
        // by checking to see if the session is a new one,
        // we are checking to see if the user somehow
        // got to the add function without logging in first.
        // If so, we send them to the login page.
        if(session.isNew()){
          response.sendRedirect(pathtohtml+"login.html");
        }
        else{
          // Here we get the user information from the session object,
          // add the bookmark, put the  information back
          // in the session object, and pass request to the JSP.
          BKUser user = (BKUser)session.getAttribute("user");
          user.addBookmark(request.getParameter("name"),
                           request.getParameter("href"));
          session.setAttribute("user", user);
          passItOn(request, response, pathtojsp+"Bookmarks.jsp");
        }
      }

      else if (action.equals("delete")){
        // by checking to see if the session is a new one,
        // we are checking to see if the user somehow
        // got to the delete function without loging in first.
        // If so, we send them to the login page.
        if(session.isNew()){
          response.sendRedirect(pathtohtml+"login.html");
        }
        else{
          // Here we get the user information from the session object,
          // delete the bookmark, put the information back
          // in the session object, and pass request to the JSP.
          BKUser user = (BKUser)session.getAttribute("user");
          user.delBookmark(request.getParameter("name"));
          session.setAttribute("user", user);
          passItOn(request, response, pathtojsp+"Bookmarks.jsp");
        }
      }

      else if (action.equals("register")){
        UserDB.addUser(request.getParameter("username"),
                       request.getParameter("password"));

        BKUser user= UserDB.getUser(request.getParameter("username"),
                                    request.getParameter("password"));

        // After inserting the user into the database,
        // we store the user information in the session object
        // using the setAttribute method. We then pass off
        // the request to a JSP that provides the user interface.
        if(user!=null){
          session.setAttribute("user", user);
          passItOn(request, response, pathtojsp+"Bookmarks.jsp");
          UserDB.updateDB();
        }
        else{
          response.sendRedirect(pathtohtml+"error.html");
        }
      }

      else if (action.equals("logout")){
        UserDB.updateDB();
        // end the user session
        session.invalidate();
        response.sendRedirect(pathtohtml+"logout.html");
      }
    }
    catch(Exception e){
      e.printStackTrace();
    }
  }
```

```
/**
 *  We use this method to forward requests sent to the servlet
 *  to other resources. Forwarding allows us to separate the
 *  presentation from the application logic. When using "forward",
 *  the servlet cannot send any output before forwarding.  If
 *  the servlet needs to send some output of it's own, the
 *  "include" method must be used.
 */
private void passItOn(HttpServletRequest request,
                      HttpServletResponse response, String nextpage)
   throws ServletException, IOException{
  RequestDispatcher disp = getServletContext().getRequestDispatcher(nextpage);
  disp.forward(request, response);
}

/**
 *  The destroy method is called when the servlet is unloaded
 *  by the server. We close the database in this method.
 */

public void destroy(){
  UserDB.closeDB();
}

} // class BookmarkServlet
```

BKUser.java

Here is the source for `BKUser.java`.

```
import java.io.*;
import java.util.*;
import java.net.*;

public class BKUser{
  public String name;
  protected String password;
  public Hashtable bookmarks;

  public BKUser(String name,
                String password,
                String dbstring){
    this.name=name;
    this.password=password;
    this.bookmarks=new Hashtable();
    if(dbstring!=null){
      this.readDBString(dbstring);
    }
  }

  public void addBookmark(String name, String href){
    bookmarks.put(name, href);
  }

  public void delBookmark(String name){
    bookmarks.remove(name);
```

```
   }

   // convert user information to format for data file
   public String getDBString(){
     String dbstring="";
     boolean firsttime=true;
     dbstring += URLEncoder.encode(this.name) + " " +
                 URLEncoder.encode(this.password) + " ";
     for (Enumeration enum = bookmarks.keys() ;
                             enum.hasMoreElements() ; ) {
       String bkname=(String)enum.nextElement();
       String bkhref=(String)bookmarks.get(bkname);
       if(firsttime){
         dbstring+=URLEncoder.encode(bkname)+"!"+
                   URLEncoder.encode(bkhref);
         firsttime=false;
       }
       else{
         dbstring+="|"+URLEncoder.encode(bkname)+"!"+
                       URLEncoder.encode(bkhref);
       }
     }
     return dbstring;
   }

   // extract user information from data file format
   public void readDBString(String dbstring){
     StringTokenizer sptok = new StringTokenizer(dbstring);
     this.name=URLDecoder.decode(sptok.nextToken());
     this.password=URLDecoder.decode(sptok.nextToken());
     StringTokenizer pptok = new StringTokenizer(sptok.nextToken(),"|");
     while (pptok.hasMoreTokens()) {
       StringTokenizer exctok = new StringTokenizer(pptok.nextToken(),"!");
       bookmarks.put(URLDecoder.decode(exctok.nextToken()),
                     URLDecoder.decode(exctok.nextToken()));
     }
   }
 } //class BKUser
```

UserDB.java

Here is the source for `UserDB.java`.

```
 import java.io.*;
 import java.util.*;

 public final class UserDB{
   static final String dbfile="userdb.txt";
   static Vector users;
 // the objects stored in the users Vector will be of the type BKUser

   public static void initDB(){
     users = new Vector();
     readFromFile(dbfile);
   }
```

```
  public static void closeDB(){
    writeToFile(dbfile);
    users=null;
  }

  public static void updateDB(){
    writeToFile(dbfile);
  }
  /**
   *   The following function will open the database file and scan
   *   into the Vector users all of the user information and bookmarks.
   *   The file will be of the format
       "username password mark1!url1|mark2!url2|...markn!urln"
   *   with each user information on a separate line.
   */

  public static synchronized void readFromFile(String filename){
      try{
        BufferedReader in = new BufferedReader(new FileReader(dbfile));
        String line;
        while((line=in.readLine())!=null){
          users.add(new BKUser(null, null, line.trim()));
        }
        in.close();
      }
      catch(Exception e){
        e.printStackTrace();
      }
  }

  /**
   *   The following function will do the opposite of the previous function
   *   All of the user information and bookmarks from the Vector users
   *   will be written to the file using the format
       "username password mark1!url1|mark2!url2|...markn!urln"
   *   with each user information on a separate line.
   *   These sections of code are synchronized to prevent multiple
       writings from running into each other.
   */
public static synchronized void writeToFile(String filename){
     try{
        PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(dbf
        for (Enumeration enum = users.elements() ;
                                enum.hasMoreElements() ; ) {
          out.println(((BKUser)enum.nextElement()).getDBString());
        }
        out.close();
      }
      catch(Exception e){
        e.printStackTrace();
     }
  }

  // retrieve user information from the database
  public static BKUser getUser(String username,
                             String password){
    BKUser user=null;
    for (Enumeration enum = users.elements() ;
                            enum.hasMoreElements() ; ) {
```

```
        user=(BKUser)enum.nextElement();

        if(user.name.equals(username)
           &&user.password.equals(password)){
          break;
        }

        else{
          user=null;
        }
      }
      return user;
    }

    // add a user to the database
    public static void addUser(String username, String password){
      users.add(new BKUser(username, password, null));
    }
  } // class UserDB
```

Bookmarks.jsp

Here is the source for `Bookmarks.jsp`.

```
 <%--
   These steps are carried out in the first lines of the JSP:

   JSP directives are used to import the classes needed by the
   Java code in the page.

   The directive using the session attribute instructs the page to
   keep track of the user session.  A value of "false" disables this.

   A JSP declaration is used for a BKUser object.

   Next a JSP declaration is used for a constant for easy
   modification of the page if the servlet is moved.
 --%>
 <%@ page import="java.util.*" %>
 <%@ page import="java.net.*" %>
 <%@ page import="BKUser" %>
 <%@ page session="true" %>
 <%! private BKUser user; %>
 <%! private static final String servleturl =
               "http://localhost:8080/examples/servlet/BookmarkServlet";
>
 <%
 // get the user information for this session
 user = (BKUser)session.getValue("user");
 %>
 <html>
 <head>
 <title>
 Bookmarks
 </title>
 <body>

 <%-- Using JSP expression to insert username in the output --%>
 <h1 align="center">Welcome <%= user.name %></h1>
 <table width="100%">
 <tr>

 <%-- Encode the logout URL with the session tracking information
>
   <td>
   <b><a href="<%=response.encodeURL(servleturl+"?func=logout")%>
                  ">Logout</a></b></td>
   <td><b>Bookmark Name</b></td>
   <td><b>Bookmark URL</b></td>
 </tr>
 <tr>
   <td colspan="3"><hr></td>
 </tr>

 <%
 // print bookmark rows of bookmark table
 for (Enumeration enum = user.bookmarks.keys() ;
                        enum.hasMoreElements() ; ) {

   String bkname=(String)enum.nextElement();
   String bkhref=(String)user.bookmarks.get(bkname);

   //Encode URL for session tracking with response.encodeURL()
   //URL encode the information passed to the server with URLEncoder.encode().
   //This is needed since the parameter may have spaces or other
   //characters that could be misread by the server.
   String deleteurl = response.encodeURL(servleturl +
name="
                           + URLEncoder.encode(bkname));
 %>
 <tr>
   <td><a href="<%= deleteurl %>">Delete</a></td>
   <td><a href="<%= bkhref %>" target="_blank"><%= bkname
>
   <td><a href="<%= bkhref %>" target="_blank"><%= bkhref
>
 </tr>

 <%
 }
 %>
 </table>
 <hr>

 <%-- Form for adding bookmarks --%>
 <form action="<%= response.encodeURL(servleturl) %>" method="post">
 <input type="hidden" name="func" value="add">
 Add Bookmark Name <input type="text" size="30" name="name">
 &nbsp;&nbsp;Bookmark URL <input type="text" size="30" name="href">
 <input type="submit" name="addsubmit" value="Add">
 </form>

 </body>
 </html>
```

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.