Servlets

Servlets are Java's solution for server-side programming. A servlet is a Java object which is designed to create a new webpage in response to an HTTP request. Here is a very simple servlet:

```
@WebServlet("/BasicServlet")
public class BasicServlet extends HttpServlet {

  protected void doGet(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {

     response.setContentType("text/html; charset=UTF-8");
     PrintWriter out = response.getWriter();

     out.println("<!DOCTYPE html>");
     out.println("<head>");
     out.println("<meta charset=\"UTF-8\" />");
     out.println("<title>A Basic Servlet Example</title>");
     out.println("</head>");
     out.println("<body>");
     out.println("<h1>A Basic Servlet</h1>");
     out.println("</body>");
     out.println("</html>");
  }

}
```

A servlet contains methods which are called directly in response to the various types of HTTP requests. In the example above, our servlet is only designed to respond to GET requests. If we wanted to respond to POST requests, we would write a doPost method. Similar methods exist for doHead, doTrace, and each of the other HTTP 1.1 request types.

In this case, our BasicServlet's doGet method will get called whenever anyone sends a GET request to any of the URLs mapped to the servlet. The most straightforward way to respond is to generate a new webpage directly. We do this using the HttpServletResponse passed in as a parameter. First, we set the content type to "text/html" – don't forget this step, it's very important, and it must be done before writing the actual response body. While "text/html" is the most common type generated, you can return any MIME type acceptable to the web browser. For example, our servlet could generate standard text, XML, or even generate a GIF or JPEG image file in response to an HTTP request. Next we get a hold of PrintWriter from the response, and we print out all the lines we want to appear in the HTML file.

As our BasicServlet only has a doGet method. That means that it will ignore any HTTP POST requests. Standard links are sent as GET requests, and any forms which do not explicitly specify the POST method will also use GET. However, if we are not sure whether or not we will receive POST requests, we can simply write a doPost which calls the doGet method like this:

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
   doGet(request,response);
}
```

Getting Inputs

When a user enters information into a form and then submits it, their form information is either encoded into the URL if the form uses a GET request or sent as part of the body if the form uses a POST request. Regardless of the method used, we can get the information back out from the HttpServletRequest object which our doGet or doPost method receives as a parameter.

Call getParameter on the HttpServletRequest object to retrieve the value of a specific parameter. The value returned will always be a string. Here is an example where we retrieve the age sent to us from an HTML form:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {

   response.setContentType("text/html; charset=UTF-8");
   PrintWriter out = response.getWriter();

   …
   int age = Integer.parseInt(request.getParameter("age"));

   if (age >= 18) {
      out.println("Can Vote!");
   } else {
      out.println("Not Eligible");
   }
   …
}
```

Methods are also available to get a listing of all form inputs sent, and to handle the case where multiple form inputs with the same name are received.

In addition the HttpServletRequest provides methods to access a variety of other information including information on the URL, the hostname of the sending computer, access to cookie information, and access to HTTP headers in the request.

Servlet's URL

The URL a user enters into the web browser, or the URL used within an … link determines which webpage a server returns. For HTML files, the URL determines the directory and file name of the HTML file. In the case of a Servlet, the Servlet's @WebServlet contains the URLs which will be mapped to the Servlet. For our BasicServlet example we have:

```
@WebServlet("/BasicServlet")
public class BasicServlet extends HttpServlet {
      …
```

```
}
```

This means that URLs of the form:

```
http://hostname/projectname/BasicServlet
```

will be mapped to this Servlet. For example, if I am running tests using my computer as localhost and the name of my project is Examples, then the URL should be:[1]

```
http://localhost:8080/Examples/BasicServlet
```

Using a Servlet to Forward to Another Webpage

Servlets don't necessarily have to generate a new webpage using the PrintWriter. One alternative that can be very useful in certain situations is to use the servlet code to switch to different static HTML webpages. As we'll see in the next lecture, this gives even more flexibility when combined with JSPs (JavaServer Pages).

Let's take a look at an example. Suppose that we have a webpage that asks the user what their age is. Based on their age, we want to display one of two pages – either canvote.html or noteligible.html. Here's what our doGet or doPut might look like:

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

   int age = Integer.parseInt(request.getParameter("age"));

   if (age >= 18) {
      RequestDispatcher dispatch = request.getRequestDispatcher("canvote.html");
      dispatch.forward(request, response);
   } else {
      RequestDispatcher dispatch =
               request.getRequestDispatcher("noteligible.html");
      dispatch.forward(request, response);
   }

}
```

We retrieve a RequestDispatcher, passing in the new URL that we want to use as a parameter. We then call forward, passing the current request and response parameters. Depending on the age input, the user will either see the canvote.html file or the noteligible.html file.

[1] The 8080 indicates the port number. 8080 is a common port number used for test webservers. Port 80 is used for regular webservers and 443 is used for https secure web transactions.

Sharing Information

As you start using servlets to create more complex websites you'll discover a need to share information between servlets. In a traditional program, it's easy to share information. But if your program consists of a number of different servlets, where is the information stored?

There are in fact a number of different places where information can be stored. These include:

ServletContext—In spite of its name, the ServletContext actually is shared by all servlets on your website. It therefore is the best place for information that you want to pass between servlets. You can get a hold of the ServletContext by calling getServletContext, which is a method of the Servlet class.

A database connection is an example of an object you might want to store in the ServletContext. The database connection could get initialized once, and then shared by different Servlets as needed.

HttpSession—Each individual user accessing the webserver has their own HttpSession. Call getSession on the HttpServletRequest to retrieve the session.

Items in a shopping cart are an example of data you would want to store in the HttpSession, because this information is per user.

HttpServletRequest—As we've already seen each HTTP request has their own request object. We previously saw how a servlet can forward to an HTML file. Servlets can also forward to other servlets or to JSPs. In these cases, passing information on to the new JSP or servlet may be useful. The HttpServletRequest can be used to pass information in these cases.

The ServletContext, HttpSession, and HttpServletRequest can all act as essentially maps, where you pass in a name and an object and then later retrieve the object by passing in the name. For example we could ask the user to enter their name on one webpage, and then store it in the session like this:

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

   String userName = request.getParameter("name");

   HttpSession session = request.getSession();
   session.setAttribute("name",userName);
   …
}
```

In another servlet we could retrieve the name like this:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

   HttpSession session = request.getSession();
   String username = (String) session.getAttribute("name");

   response.setContentType("text/html; charset=UTF-8");
   PrintWriter out = response.getWriter();

   …
   out.println("<h1>Hello " + username + "</h1>");
   …
}
```

There is one more place you can use to store information:

ServletConfig—Each servlet has its own ServletConfig. This object is used to set initial values on the servlet. The ServletConfig values in the servlet config are set in advance by the webserver, and cannot be changed while the program is running. The values come from a special configuration file called web.xml.

The name of the database to use is an example of something you might want to access through the ServletConfig. Suppose you wanted to be able to quickly switch your web server from accessing a testing database to using a production database. If your program looks up the name of the database in the ServletConfig, a simple change to the web.xml file would automatically change your server from one database to the other.

The ServletConfig only allows you to retrieve string values. You can retrieve a specific value by calling getInitParameter along with the name of the parameter you are asking for. You can retrieve the names of all the parameters available by calling getInitParameterNames.

Listeners

While the ServletContext and HttpSession can be used to exchange information between servlets, one issue you will run into is who sets an initial value for that information. One possible solution is to have each servlet check and see if the information has been initialized. If it hasn't the servlet can go ahead and initialize the information itself. A better solution is to take advantage of a listener.

In Java Enterprise Edition, we can create a listener class. This class will listen for the creation of ServletContexts, HttpSessions, or ServletRequests. It can also listen for changes to attributes on these objects. Make sure you tell Eclipse specifically that you want to create a Listener ("New > Other …" and then "Web > Listener"). Eclipse needs to do some setup to make sure that your listener gets called automatically by the system when the events you are interested in occur.

Here we define a listener which will run code when an HttpSession is created:

```
public class CityListener
      implements HttpSessionListener, HttpSessionAttributeListener {
```

```
    public void sessionCreated(HttpSessionEvent event) {
        HttpSession session = event.getSession();
        session.setAttribute("city", "Palo Alto");
    }
}
```

As you can see the sessionCreated code will set an initial value for the "city" attribute. This code will run before any of our servlets get a chance to interact with the HttpSession object. Therefore they can all assume that the "city" attribute has a reasonable value set, instead of the null which will be returned if we do not initialize the attribute.

Warning: Some students have not been able to get listeners working. Unfortunately we have not been able to track down the source of the problem. This error is rare, and may be tied to specific machine configurations. If you run into this error, use the technique mentioned at the start of this section (have each servlet check to see if the information is initialized and if it isn't initialize it) otherwise use listeners.