eCommerce Systems / Professor H Roumani

PROJECT B

Executive Summary

You will create a server-side application that provides 7 web services. The services were designed to expose various aspects of web development, such as form handling, TCP connections, database lookups, HTTP connections, json/xml to html transformation, and open authentication. We will use an agile methodology; adhere to best-practice design patterns such as MVC; and leverage the Tomcat server to host our logic.

The First Release

You will be guided toward the 0.1 Release through step-by-step instructions. Follow them carefully and don't proceed to the next instruction until you complete the one before it. If you get stuck in one step, post a question on the forum.

Note that these instructions assume that you are working within the Virtual Box on your own machine or on a loaner laptop that you borrowed from the Prism monitor. If you want to work on a lab workstation, you first need to set it up using the instructions in this document (this needs to be done just once).

The Project

1. Launch Eclipse.
2. Select the pre-prepared workspace `ws_4413`. You know you are in the correct workspace if you see a `Server` project already created.
3. From the File menu, create a new `Dynamic Web Project`. If you don't see this option in the menubar then select `New` then `Project...` then look for "Dynamic Web Project" in the selection wizard.
4. Name the project, e.g. "ProjB" and accept all defaults.

The View

1. Right-click the "WebContent" folder within your project tree (in the Project Explorer pane) and select `New` then `HTML File`.
2. Name the file `Time.html`; click Next; select html 5; and then Finish.
3. Edit the created file so that its content matches that of this file. To do that, you can click the link and then view the source of the displayed document or save it to a file and then open the file using a text editor.

The Model

1. Right-click the "src" folder within your project tree (in the Project Explorer pane) and select `New` then `Class`.
2. Enter `model` for your class package (at the top) and `Brain` for its name. Accept all defaults and click Finish.
3. Edit the created class so that its content matches that of this file. The constants defined in the class will be used in later releases.
4. Complete the implementation of the `doTime()` method so that it would return the current time in the usual date/time format.

The Controller

1. Right-click the "src" folder within your project tree (in the Project Explorer pane) and select `New` then `Servlet`.
2. Enter `ctrl` for your servlet's Java package and `Time` for your servlet's Class name and accept all defaults.
3. The default servlet template that appears in the editor pane contains essentially three key features: an annotation at the top and two methods: doGet and doPost.
4. Edit the annotation so it becomes `@WebServlet("/Time.do")` and remove (just comment out) the provided statement in doGet which writes to the response.
5. Notice that doPost calls doGet so we need only implement the body of doGet. Copy and paste the following code fragment to the body of doGet:

```
if (request.getParameter("calc") == null)
{
   this.getServletContext().getRequestDispatcher("/Time.html").forward(request, response);
}
else
{
   Brain model = new Brain();
   try
   {
      String time = model.doTime();
      response.setContentType("text/html");
      Writer out = response.getWriter();
      String html = "<html><body>";
      html += "<p><a href='Dash.do'>Back to Dashboard</a></p>";
      html += "<p>Server Time: " + time + "</p>";
      html += "</body></html>";
      out.write(html);
   }
   catch (Exception e)
   {
      response.setContentType("text/html");
      Writer out = response.getWriter();
      String html = "<html><body>";
      html += "<p><a href=' Time.do'>Back to Dashboard</a></p>";
      html += "<p>Error " + e.getMessage() + "</p>";
      out.write(html);
   }
}
```

Essentially, the code serves (through the request dispatcher) an html page that contains a form. The client is supposed to fill the form and submit it back to this same servlet. The servlet distinguishes the first visit from the second through the presence or absence of a `calc` parameter. For the second visit, the servlet asks the model to process the request and it then serves the result.

Deploying the First Release

Right-click the `doTime` servlet in the project explorer and select "Run As" and then "Run on Server". You will see the Tomcat server startup log in the console, possibly with some irrelevant exceptions about locales, but after a few seconds, you will see that your project has been deployed, and a browser will launch with the URL:

```
http://localhost:4413/<project name>/doTime
```

Take some time now to understand what you have built and think it through. This is a good opportunity to see how the pieces fit together given that we have only one page in our webapp and that its functionality is pretty simple. Feel free to modify any of the MVC components and observe the ensuing cause-effect behaviour.

Reference Implementation

In order to clarify the requirement and provide a feel for the sought functionality, here is a link to the complete project in production:

http://red.cse.yorku.ca:4413/44F18_ProjB/Dash.do

Your completed project should have the same behaviour as this one. Take some time to explore this webapp.

The Services

The table below shows the services that your webapp will provide and stipulates a naming convention for the servlet class, the URL mapping annotation, the HTML page file, and the method in the Brain model (the return of this method is always a String):

SERVICE	SERVLET (C)	URL @	HTML (V)	METHOD (M)
S0: Dashboard	Dash	Dash.do	Dash	none

S1: Simple Computation	Time	Time.do	Time	doTime()
S2: Computation	Prime	Prime.do	Prime	doPrime(String)
S3: TCP	Tcp	Tcp.do	Tcp	doTcp(String)
S4: JDBC	Db	Db.do	Db	doDb(String)
S5: HTTP	Http	Http.do	Http	doHttp(String,String)
S6: XML Tabulation	Roster	Roster.do	Roster	doRoster(String)
S7: Open Authentication	OAuth	OAuth.do	OAuth	none

Implementation Notes

- The Dash service is merely a dashboard or a main menu for the 7 services. Each of them must provide a link back to it. You therefore should edit the link in the Time servlet that you created for the first release so it points to `Dash.do` rather than `Time.do`.
- All services have similar structures for their servlets and HTML pages. Hence, to implement a new service, simply copy-and-paste the servlet/html of the previously implemented service and then edit as needed. For the Brain model, you will need to add a new method per service.
- All the model's methods can throw exceptions. These are caught in the calling servlet and a proper response is served.
- For S2, use the same BigInteger approach as you did in Project-A.
- For S3, the idea is ***not*** to generate the prime yourself but to rely instead on the services of a TCP server elsewhere. That server is available at the hostname and port provided in the constants of the Brain class. To make a connection to that server, you create a **client** socket and then use its input and output streams to talk to that server. The server protocol is that same as in Project-A, so something like this is needed in the body of the `public String doTcp` method:

```
Socket client = new Socket(TCP_SERVER, TCP_PORT);
issue: prime digits on the socket's output stream
read: the returned prime from the socket's input stream
issue: bye on the socket's output stream
return the obtained prime to the calling servlet.
```

- `For S4, the code>doDb(String itemNo)` method needs to connect to a database (whose URL is provided in the Brain's constants) in order to find out the name and price of a given inventory item number. Here is the body of that method:

```
Class.forName("org.apache.derby.jdbc.ClientDriver").newInstance();
Connection con = DriverManager.getConnection(DB_URL);
Statement s = con.createStatement();
s.executeUpdate("set schema roumani");
String query = //SQL query to obtain the NAME and PRICE of an item whose number is itemNo in a table ITEM
ResultSet r = s.executeQuery(query);
String result = "";
if (r.next())
{
      result = "$" + r.getDouble("PRICE") + " - " + r.getString("NAME");
}
else
{
      throw new Exception(itemNo + " not found!");
}
r.close(); s.close(); con.close();
return result;
```

To experiment with this service, here are some items:

Number	Name	Price
0905A112	T6 Sirloin Meat	$6.01
2002H063	Semi-Cheddar Cheese by JC	$4.26
0905A456	Crown Fillet Meat by TH	$3.87
1409S381	Brownie Ice Cream with Praline by MG	$7.39

- For S5, you need to determine the capital of a given country or the population of the capital of a given country. You do that by consulting a web service at this URL:

  https://www.eecs.yorku.ca/~roumani/servers/4413/f18/World.cgi

  This web service expects you to indicate the name of the country and whether you want the capital or the population by providing request parameters after the URL. You prefix these parameters by '?' and delimit them by '&'. The first parameter is 'country' (whose value is a country name) and the second is 'query' (whose value is either 'capital' or 'pop'). For example, to get the capital of Canada you would write the above URL as:

  ```
  .../World.cgi?country=canada&query=capital
  ```

  To implement the HTTP lookup in code, your doHttp(String,String) method must make an HTTP connection to the server indicated by the constant in Brain. To that end, use the `URL` class in `java.net`; establish a connection; and then read the return JSON response.

- For S6, you are obtaining the student roster of a course (similar but not exactly the same as Project-A) from an HTTP server. Again, you should start by doing so manually in a browser by visiting its URL:

  https://www.eecs.yorku.ca/~roumani/servers/4413/f18/Roster.cgi

  and providing the course number (such as EECS4413) through a request parameter named 'course'. Observe that the return is XML so your method needs to capture it and then transform it to an HTML table.

- For S7, the servlet redirects the user (via `response.sendRedirect`) to the following server:

  https://www.eecs.yorku.ca/~roumani/servers/auth/oauth.cgi

  and provide a parameter named 'back' containing the URL of the OAuth servlet. This way, the user's browser is redirected to the above server where the user is prompted to login. After wards, the above server will redirect the user back to us (to OAuth) and supply the user login name and full name as parameters named 'user' and 'name'.

The Development Process

Here is a suggested development schedule:

1. Complete Release **0.1** as detailed above.
2. Incorporate a dashboard into Release **0.1**.
3. Release **0.2** should provide S2 (Prime) and S3 (TCP).
4. Release **0.3** should provide S4 (database) and S5 (HTTP).
5. Release **0.4** should provide S6 (XML-to-HTML).
6. Release **0.5** should provide S7 (OAUTH).

Persist Your Work

Follow these steps to persist / backup your work:

- Right-click your project (in the Project Explorer) and select Export.
- Select **WAR** File.

- Provide a destination for the file (e.g. ProjB.war on the Desktop).
- Check the box to *Export the source files* (extremely important)

Now copy the created war file to your Google Drive, DropBox, or some other cloud service, or copy it to a USB drive. If you later need to restore this backup into a fresh workspace, do this:

- Launch Eclipse in a new workspace.
- Right-click anywhere in the Project Explorer and select Import.
- Select WAR file.
- Point to your war file and click Finish.

Try the above procedure end-to-end (by using two machines; by switching to a different workspace on the same machine; or by deleting your workspace after the zip file has been created). Make sure you are comfortable backing up and restoring your course project. Do not wait until the day of the test to learn how to do this on a loaner laptop!

Do not delay the backup until the work is done! Do it often (at least after every release). If you are familiar with `github` then use it instead of the above (more on that in lecture or on the forum).

Finally note that the above WAR file approach is the best way to copy and/or rename your webapp.

Reflections

- Scalability-wise, instantiating the model with every request is not a good practice. Turn your model into a singleton so it is only instantiated once.
- Security-wise, critique the open auth method used. In particular, is it vulnerable to a point that someone may authenticate in the eyes of the webapp without actually authenticating at all!

• EECS4413 • Professor Roumani • Don't miss the forest for the trees •