SENG2050 - Lab 03

March 13, 2021

Note: You may need to set your CLASSPATH and JAVA_HOME on the lab computer each time you log in.
Note: Create a webapps/lab03/ directory to complete these tasks. These tasks build on last week's, you way want to copy last week's lab content into this week's lab folder.
Note: Remember to **NEVER** copy text straight out of this lab sheet.

In this tutorial you will be introduced to Java Server Pages. This is by familiarising yourself with the basic control constructs used. More advanced usages of JSP pages, in particular, using JSPs in conjunction with Servlets and Java Beans, will be covered in the following weeks. In this laboratory we will learn how to:

- Create Java Server Pages; and
- Pass values directly to Java Server Pages.

A Java Server Page (JSP) should be considered to be a 'Servlet in disguse'. It is written as a 'HTML-like' file, that embeds Java code. The jsp file itself is not a web page. But it is used to generate a dynamic web page, controlled by the embeded Java code. This code is typically used to control the generation of content. But can also be used to do anything a normal Java servlet can do. However, the real power of a JSP is for generating web pages - not writing Java code.

JSP Tips

- JSPs should be saved in:

  `apache-tomcat/webapps/[WEBAPP_NAME]/`

  This is in the same location as static content (HTML, CSS, JavaScript, etc.).
- For this lab they will go in:

  `apache-tomcat/webapps/lab03`
- JSPs are accessed 'directly'. E.g.

  `http://localhost:8080/lab03/[DOCUMENT_NAME].jsp`
- JSP's are automatically compiled and changes are automatically updated
- If you have a problem with the content not updating, delete everything under Tomcat's **work/** directory, this is where Tomcat caches compiled JSPs.

JSP Tags

- Scriptlets -

  ```
  <% java code %>
  ```

  - Standard Java code goes between these tags.
  - i.e.

    ```
    <%
    List myList = someList();
    for(int i=0; i<myList.size(); i++)
    out.println(myList.get(i).toString());
    >
    ```

  - **Note:** you have access to the **JspWriter out** object by default (along with the **HttpServletRequest request** and **HttpServletResponse response** objects).
  - You can also mix and match HTML template and Java code

    ```
    <ul>
    <% for (int i = 0; i < 10; i++) { %>
    <li><%= i %></li>
    <% } %>
    </ul>
    ```

- Expressions -

  ```
  <%= java expression %>
  ```

  - Java code within these tags is evaluated and the result is inserted into the servlet's output.
  - i.e. $< \% = 3 + 3\% >$ and $< \% = "6"\% >$ will both output 6

- Declarations -

  ```
  <%! Java declaration or method %>
  ```

  - A declaration is a block of Java code that is used to define class-wide variables and methods in the generated Servlet class.

- Page directives -

  ```
  <%@page %>
  ```

  - Page directives.
  - Used to provide instructions to the container in relation to the current JSP page.
  - Useful for importing other classes and libraries.
  - These generally go up the top of the JSP.
  - i.e.

    ```
    <%@page import="java.io.IOException"%>
    <%@page contentType="text/html" pageEncoding="UTF-8"%>
    ```

Task 1 - Accessing Request Inputs

JSPs support accessing request inputs throught the HttpServletRequest. This is exposed as the implicit object **request**, on which you can call the familar **getParameter()** method. Retreiving parameters must occur in scriptlets. You can later print these values out using expression tags. To demonstrate the retreival of request inputs, we will create a simple 'echo' application.

1. Start by creating two .jsp files: form.jsp, and echo.jsp.
2. In form.jsp, create a HTML form that submits a *GET* request to echo.jsp. The form will have the following inputs:
   - Name
   - Age
   - Address
   - Occupation
3. In echo.jsp, write a JSP scriptlet that retrieves the four request parameters. E.g.

```
<%
String name = request.getParameter("name");
String age = ...
...
>
```

4. Print the four request parameters out into a HTML table using JSP expression tags. E.g.

```
...
<tr>
<td>Name</td>
<td><%= name %></td>
</tr>
...
```

5. After the HTML table, present the following message:
   "Thank you for using our form, (name). In 10 years, you will be (age + 10) years old!"
   Note: Make sure you replace (name), and (age+10) with the correct values.
6. Please make sure both JSP pages generate valid HTML, and you handle the case age is not passed as an integer.

Using Control Statements

In Java, we have 2 basic types of control statements:

- Conditional - If, Switch, Ternary (expression).
- Repetition - For, For (iteration), While, Do-While.

All of these statements can be embedded in JSP scriptlets or expressions. The conditional statements allow us to do things like control sections of a page to be displayed (e.g. show different sections based on inputs). The repetition statements allow us to repeat sections of a page (e.g. print out a list of items).

Task 2 - Conditional Statements

To demonstate conditional statements, we will create a simple 'age check' application. This will ask the user their age, and hide the displayed content based on their age.

1. Create a pair of .jsp files: gatekeeper.jsp and content.jsp.
2. In gatekeeper.jsp, create a HTML form that submits a *GET* request to content.jsp with the following inputs:
   - Name
   - Age
3. In content.jsp, write a JSP scriptlet that retrieves these parameters.
4. In content.jsp, use JSP scriptlets to display a message to the user:
   - *"You're too young to visit this site."*, if the user is under 18.
   - *"Go home grampa!"*, if the user is over 67.
   - *"Welcome to our site!"*, for all other users.

   Note: You can spread an if statement accross multiple scriptlets. Just make sure all curly braces match up!
5. Please make sure both JSP pages generate valid HTML, and you handle the case age is not passed as an integer.

Task 3 - Repetition Statements

To demonstrate repetition statement, we will create a 'content repeater' application. This will display a textual string in a HTML table, repeated accross r rows, and c columns.

1. Create a pair of .jsp files: input.jsp and repeater.jsp.
2. In input.jsp, create a HTML form that submits a *GET* request to repeater.jsp with the following inputs:
   - Rows
   - Columns
   - Content
3. In repeater.jsp, write a JSP scriptlet that retrieves these parameters. Note: rows and columns should be ints.

4. In repeater.jsp, write code to generate a HTML table of r rows by c columns, with each cell containing the passed content. *Hint: This can be achived with a pair of embedded for loops!*
5. Please make sure both JSP pages generate valid HTML, and you handle the case rows or columns is not passed as an integer.

Task 4 - JSP Member Declarations

As JSPs are compiled into Java classes, it standands to reason JSPs shoudl also support the declaration of members (variables and methods) for use within the page. JSPs expose this functionality through the declaration tag.

In practice, there are very few use cases for declaring a member variable (e.g. creating a page counter). Furthermore, member variables need to take into account thread safety as they can potentially be accessed many times in parallel. Likewise, there are very few use cases if declaring methods, except for utility functions used on one page only. Strictly for demonstration purposes, we can use a JSP declaration tag to declare a member and utility method.

4A - A thread-safe page counter

We can create a thread-safe counter with an AtomicInteger. First, we need to import the class.

```
<%@page import="java.util.concurrent.atomic.AtomicInteger"%>
```

This class wraps an int, enforcing thread safety on all accesses. We can declare an atomic integer that starts counting from zero as followed:

```
<%! AtomicInteger counter = new AtomicInteger(); %>
```

We can then increment this integer using a variety of methods. See `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html` for more info. For example:

```
counter.incrementAndGet() or counter.getAndIncrement()
```

Use these code fragments to write a new .jsp file, counter.jsp, that displays a view counter.

4B - A date formatter

Create a new .jsp file called 'date.jsp'. Within it, declarate a method with the following prototype:

```
<%!
String getCurrentDateWithFormat(String dateFormat) {
...
}
%>
```

In this method you will:

- Get the current date and time (as a java.util.Date, try importing this class to avoid writing it out fully-qualified).

- Format the date and time as per the provided parameter (see Java SimpleDateFormat).
- Return the formatted date as a String.

Example usage:

```
The current data and time is: <%= getCurrentDateWithFormat("dd-MM-yy hh-mm-ss a") %>
```

Task 5 - Storing Parameters in Beans

So far, in each task we had to manually retrieve request parameters. This isn't too bad, but when we have non-string parameters, it means we have to parse them into the corresponding primitive type. Thankfully, converting request parameters into primitive types is a native feature of Java beans. Modify the code from the previous exercises to use a JavaBean.

To do this you will need to create and compile a JavaBean class and store it in a Java package in the directory:

```
webapps/lab03/WEB-INF/classes/[PACKAGE_NAME]
```

Remember, when a Java class is in a package, the first line of the .java file must contain a package declaration. E.g.

```
package lab3;
```

To include the bean in your JSP use:

```
<jsp:useBean id="[INSTANCE_NAME]" class="[PACKAGE_NAME].[CLASS_NAME]" scope="request" />
```

The JSP should set all the bean's values using tag(s), i.e.

```
<jsp:setProperty ... />
```

You can access the bean's methods in your Java code by using the reference created by the jsp:useBean statement (the value of the **id** attribute), or by using:

```
<jsp:getProperty ... />
```

See the lecture slides for more details on the use of JSP bean tags.

JSP Best Practices

As a rule of thumb, a JSP page should not contain any Java code. While this is a well-supported feature, it is bad practice to mix lots of Java and HTML in the same JSP file. At a bare minimum you should not place request processing and application logic in the JSPs. But in an ideal scenario you should not have any Java code whatsoever in your JSPs! While this may seem difficult at this point in the course, JSPs contain many more advanced features that make this possible, that we will later see in this course.

For now, try and minimise the amount of Java code in your JSPs. Limit yourself to only conditional or repetition statements. Revise you lab work, and see if you can remove all JSP scriptlets, expressions and declarations. Try replacing them with beans.