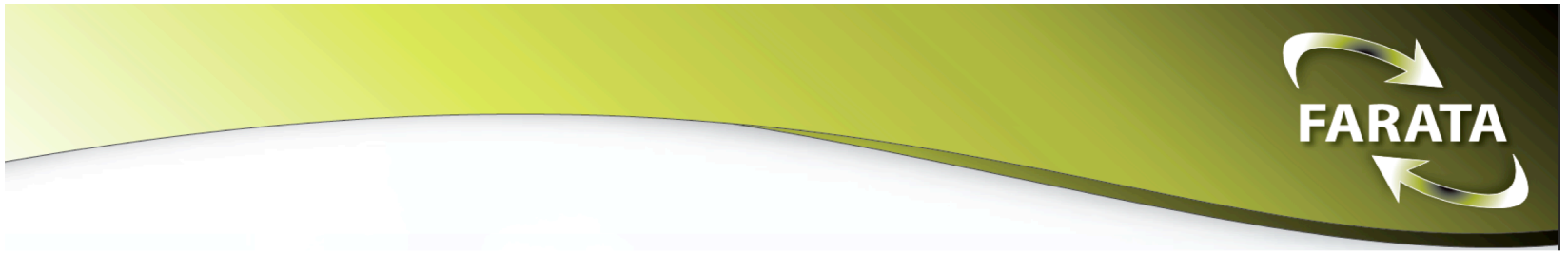


Java Programming
Unit 15

HTTP Sessions and cookies
Java Server Pages

Synchronous and Asynchronous Servlets

Java Servlets run in a servlet container.

Prior to the spec Servlet 3.0 the container would spawns a new thread for every client's request.

JSR 315 added support of asynchronous servlets:

```
@WebServlet(urlPatterns={"/bids"}, asyncSupported=true)
```

In this lesson we are considering only synchronous servlets.

Web Browser -> Servlet Workflow

The web browser can use an HTML form, a link, or another program that can send request to the server like GET, POST, et al.

```
<form action=”loginServlet" method=”post">
   Username: <input type="text" name="user">
   Password: <input type="text" name=”pwd">
        <input type="submit" value="Submit">
</form>
```

Web Browser -> Servlet Workflow

- The servlet container will create **only one** instance of the servlet and will invoke the its method `init()`.

- The container calls the method `service()` of the servlet's superclass, which redirects the request to `doGet()`, `doPost()`, or similar doXXX(), passing the arguments `HttpServletRequest` and `HTTPServletResponse`.

HTTP GET and POST requests

- HTTP specification defines several methods for data exchange: `GET`, `POST`, `PUT`, `DELETE`, and more.

- If the HTTP request is made with `GET`, the browser appends parameters, if any, at the end of the URL: http://www.mybooks.com?booktitle=Apollo

- The method `POST` is typically used for creating content on the server. With `POST`, parameters are not appended to the URL.

Data Exchange: The Servlet Side

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

    // Getting data from the browser
    String title = request.getParameter("booktitle");

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

    // Sending HTML content to the browser
    out.println("<html><body>");
    out.println("<h2>the book "+title+" costs only $65");
    out.println("<p>Please enter your credit card number");
    out.println("</body></html>");
 }
```

Walkthrough 1

- Change the code of the method `doGet()` of the servlet FindBooks from the project Lesson27 to look as on the previous slide.

- Start GlassFish and run the servlet and observe the output (note the URL: http://localhost:8080/lesson27/book).

- In the Web browser enter the following URL: http://localhost:8080/lesson27/book?booktitle=Apollo

- Observe the output – the servlet responded with "the price" of Apollo book.

Session Tracking

- A session is a logical task, which the user is trying to complete by visiting a website, e.g. buying a book in several steps.

- **HTTP is stateless protocol**, but you can implement session tracking programmatically.

- Session data on the client side can be stored using *cookies* or URL rewriting.

- Session data on the server-side is stored using session tracking API that implements the interface `javax.servlet.http.HTTPSession`.

Cookies

- ***Cookies*** are small pieces of data that Web server can send to the Web browser to be stored on the disk.

- When the Web browser connects to a URL, it tries to find locally stored cookies to send them to the server.

- The user may disable cookies by selecting the Web browser settings.

- When the session is created, a special cookie, `JSESSIONID`, is sent to the client.

Using javax.servlet.http.Cookie

```
// Sending a cookie to the client
Cookie myCookie = new Cookie("bookName",
                             "Java Programming 24-hour trainer");

// Set the lifetime of the cookie for 24 hours
myCookie.setMaxAge(60*60*24);
response.addCookie(myCookie);
```

```
//Retrieving client's cookies from HttpServletRequest:

Cookie[] cookies = request.getCookies();

for (i=0; i < cookies.length; i++){
  Cookie currentCookie = cookie[i];
  String name = currentCookie.getName();
  String value = currentCookie.getValue();
}
```

URL Re-writing

If a client disables cookies in Web browser, a servlet can use *URL rewriting* for session tracking. In this case the session ID and other required session data are attached to the URL string.

HTML 5 supports Web Storage (a.k.a. local storage) that allows to store key value pairs on the user's disk drive, but as opposed to cookies, these data always stay on the client side.

Server-Side HttpSession

Keep the data that belong to a user's session (e.g. shopping cart) inside the `javax.servlet.http.HttpSession` object in the servlet container.

```
HttpSession session = request.getSession(true);

// This sample uses ArrayList object here to store selected books.
// Try to get the shopping cart that might have been
// created during previous calls to this servlet.

ArrayList myShoppingCart= (ArrayList) session.getAttribute("shoppingCart");

if (myShoppingCart == null){
  // This is the first call – instantiate the shopping cart
   myShoppingCart = new ArrayList();
}

// create an instance of a book object
Book selectedBook = new Book();

selectedBook.title=request.getParameter("booktitle");
selectedBook.price= Double.parseDouble(request.getParameter("price"));

// Add the book to our shopping cart
myShoppingCart.add (selectedBook);

// Put the shopping cart back into the session object
session.setAttribute("shoppingCart", myShoppingCart);
```

```
class Book  {
   String title;
   double price;
}
```

When the book order is placed, close the session:

```
session.invalidate();
```

If the session has not been closed explicitly, the application server will do it automatically after a specified period of time (timeout).

Walkthrough 2 (start)

Deploy a servlet by packaging all its files into one .war file:

- Create a .war file from the Lesson27 project:
  - right-click on the project
  - select Export | Web |WAR file
  - select any directory as a destination

- Find the file lesson27.war and unzip it into any directory (you may need to change .war into .zip first).

- Examine the content of this directory

- In Eclipse undeploy Lesson27 from GlassFish (right-click on server, Add and Remove, Remove)

- Start GlassFish. Confirm that Lesson 27 is not deployed. Entering http://localhost:8080/lesson27/book should return the error 404.

Walkthrough 2 (cont.)

- Move the file lesson27.war into the directory *glassfish4/glassfish/domains/domain1/autodeploy*

- Open the *server.log* file in the directory */glassfish4/glassfish/domains/domain1/logs*

- Locate the record about lesson27.war loading or depolyment

Note: *Applications deployed from Eclipse are located in the directory eclipseApps under your GlassFish domain, e.g. glassfish4/glassfish/domains/domain1/eclipseApps. If your app is in active development, remove the war file from autodeploy directory to run the current version of your code, and not the one from the war file.*

Walkthrough 2 (end)

- See if the servlet FindBooks is running by entering: http://localhost:8080/lesson27/book

- Open GlassFish admin server at http://localhost:4848 and check if Lesson27 is deployed (see the Applications node).

- Disable the lesson27 application using the admin panel. Select the Applications node on the left, check lesson27, and press Disable.

- Entering http://localhost:8080/lesson27/book returns 404 again.

JavaServer Pages (JSP)

Servlets vs JSP

Servlet's code includes generation of the HTML.

Any UI change requires Java code change, recompilation, and redeployment:

```
out.println("<html><body>Hello World </body></html>");
```

JSP allow to separate work on UI and programming.

Matilda, who knows only HTML can make changes to the UI without the need to recompile Java code.

Peter can write Java without worrying about HTML.

Each JSP turns into a servlet

- When a Servlet container loads the JSP for the 1st time it automatically generates the servlet and deploys it.

- During code generation, JSP tags like `<%=2+2%>` turn into Java code, e.g. `out.println(2+2);`

- To deploy JSP either copy them into the document root directory of your servlet container or package it inside the WAR file.

Walkthrough 3 (Start)

1. In Eclipse create a new Dynamic Web Project named lesson28.

2. Create new index.jsp file using the menu File | New | Other | Web | JSP File. On the Select JSP Template window select New JSP File (html).

3. Locate index.jsp in the WebContent folder.

2. Modify the <body> part of index.jsp to look like this:

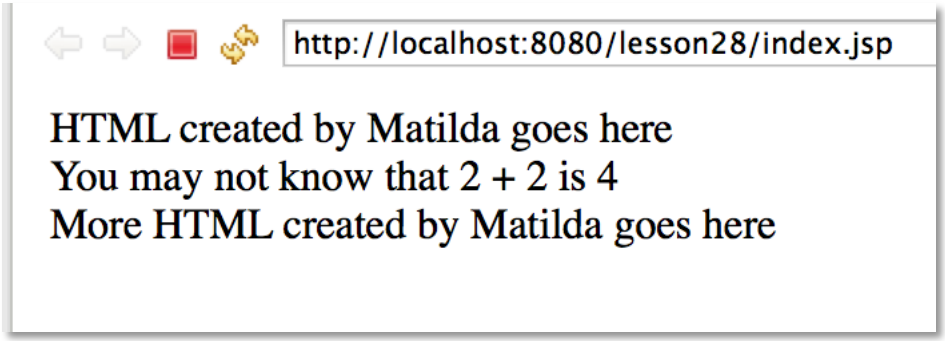
```
<body>
   HTML created by Matilda goes here
   <br>
   You may not know that 2 + 2 is  <%= 2 + 2%>
   <br>
   More   HTML created by Matilda goes here
 </body>
```

Walkthrough 3 (End)

3. In Eclipse, deploy lesson28 to GlassFish: right-click on the server and use the option Add and Remove.

4. Start the server in Eclipse if it's not running.

5. Run index.jsp in Eclipse. Observe the output. Note the URL: http://localhost:8080/Lesson28/index.jsp



Implicit JSP Objects

The following variables are pre-defined in JSP:

request has the same use as `HttpServletRequest`

response has the same use as `HttpServletResponse`

out represents the output write stream `JspWriter`. This variable points at the same object as `HttpServletResponse.getWriter()` in servlets.

```
<html>
  <body>
    <% out.println(CurrencyConverter.getDollarRate()); %>
  </body>
</html>
```

continued…

Implicit JSP Objects (cont.)

`session` represents an instance of the user's `HTTPSession` object.

`exception` represents an instance of the `Throwable` object and contains error information. This variable is available only from the JSP error page.

`pageContext` represents the JSP context and is used with Tag Libraries.

`config` provides initialization information used by the JSP container. Its use is similar to that of the container's class `ServletConfig`, that provides servlet initialization parameters, which might be specified in web.xml or `@WebServlet`'s `@WebInitParam`.

Selected JSP Directives

Directives instruct the JSP container about the rules to apply to JSP:

```
<%@ page import="java.io.*" %>

<%@ jsp:include page="calcBankRates.jsp" %>

<%@ include file="bankRates.txt" %>

<%@ taglib uri="my_taglib.tld" prefix="test" %>
```

Declarations

Declarations are used to declare variables before they are used:

```
<%! double salary; %>
```

The variable `salary` is visible only inside this page.

Similarly, you can declare Java methods in the JSP page:

```
<%! private  void myMethod(){
      // some code goes here
 }%>
```

Expressions

Expressions start with `<%=` and can contain any Java expression, which will be evaluated. The result will be displayed in the HTML page, replacing the tag itself, like this:

```
<%= salary*1.2 %>
```

Comments

Comments that start with <%-- and end with --%> are visible in the JSP source code, but will not be included in the resulting HTML page.

<%-- Some comments --%>

To keep the comments in the resulting web page, use regular HTML comments notation:

<!-- Some comments -->

Standard Actions

`jsp:include` adds the content during run time:

```
<jsp:include page "header.jsp" />
```

`jsp:forward` allows to redirect the program flow from the current JSP to another one preserving the request and response objects:

```
<jsp:forward page = "someOther.jsp" />
```

`response.sendRedirect(someURL)` also redirects the flow, but creates new request and response objects. It sends an additional client request to the server.

Error Pages

Error pages feature allows user-friendly way to display exceptions.

```
<html>
 Some code to calculate tax and other HTML stuff goes here
      . . .
   <%@ page errorPage="taxErrors.jsp" %>

</html>
```

← TaxCalc.jsp

```
<%@ page isErrorPage="true" %>
<html>
 <body>
  Unfortunately there was a problem during your tax
calculations. If the problem persists,
  please us at (212) 555-2222 and provide them with the
following information:
 <br>
 <%=exception.toString()>
 </body>
</html>
```

← TaxErrors.jsp

JavaBeans

JavaBeans specification defines a bean as a Java class that implements the `Serializable` interface and that has a public no-argument constructor, private fields, and public setter and getter methods.

Beans that are controlled by a container are called *managed beans*.

In JSP they are used to avoid mixing Java code and HTML

```
<jsp:useBean  id="Student" class="com.harward.Student" />

<jsp:getProperty name="Student" property="LastName" />

<jsp:setProperty name="Student" property="LastName"
value="Smith"/>
```

```
import java.io.Serializable;

class Student implements Serializable{
        private String lastName;
        private String firstName;
        private boolean undergraduate;

        Student(){
           // constructor's code goes here
        }

        public String getLastName(){
             return lastName;
        }
        public String getFirstName(){
             return firstName;
        }
...
}
```

Two ways of working with tag Libraries

1. Create custom tags and compile them into *tag libraries* – the .tld files (see http://bit.ly/KbDZXB)

2. Split your Web page into areas. Each is scripted and saved in a text file with name extension .tag (see http://tek.io/ILDaSN)

Say you have a file *billingForm.tag*, which is a JSP scriptlet. Save it the directory WEB-INF/tags and use it in your JSP:

```
<%@ taglib prefix="shop" tagdir="/WEB-INF/tags" %>
```

Add the billing form to your JSP:

```
<shop:billingForm />
```

Homework

Study all the materials from Lessons 28 from the textbook. Do the assignment from the Try It section of the lesson 28.

Optional homework:

Read the chapter on Spring framework in the text book.

Study the Intro to Spring MVC tutorial:
http://goo.gl/siDwQK

Additional Reading

1. Asynchronous servlets: http://www.javacodegeeks.com/2013/08/async-servlet-feature-of-servlet-3.html

2. Learn about Servlet Filters: http://www.oracle.com/technetwork/java/filters-137243.html

3. JSP Tutorial: http://docs.oracle.com/javaee/5/tutorial/doc/bnagx.html

4. Code sample of using `HTTPSession` object: http://bit.ly/dP8gpf