Web-based information systems 1

User authentication, sessions and JSP goodies

Michiel Noback (NOMI)
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Introduction

- The topics of this (long) presentation are
  - user authentication: check the identity of the person making a request
  - sessions: keeping track of a user's identity and/or request history
    - These are two very related subjects as you will see
  - JSP goodness: EL, JSTL, scriptlets

User authentication

- There are several reasons why you would like to know who it is that requests a certain page:
  - you want only authorized persons to have access
  - you want to track somebody's history on your site (e.g. for a shopping card)
- To be able to do this, you will need **session** information

http has no memory

- A server handling your request will have no memory of your visit after it is processed
- To be able to "keep in touch" you will need to have a tracking system: we call this a **session**
- The most well-known session implementation system is by means of **cookies**

Cookies

- “In computing, a cookie (also tracking cookie, browser cookie, and HTTP cookie) is a small piece of text stored on a user's computer by a web browser. A cookie consists of one or more name-value pairs containing bits of information such as … the identifier for a server-based session… .” (wikipedia.org)

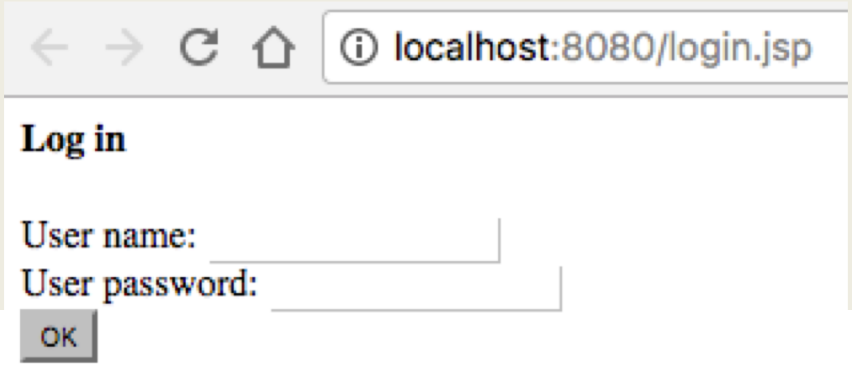
Cookies and sessions

- In java, sessions are usually maintained by means of a JSessionID stored as a cookie
- You will probably never create a cookie directly
- You simply use this statement:

  HttpSession session = request.getSession();
- Let's look at some sample code to create a login system using sessions

A login form: login.jsp

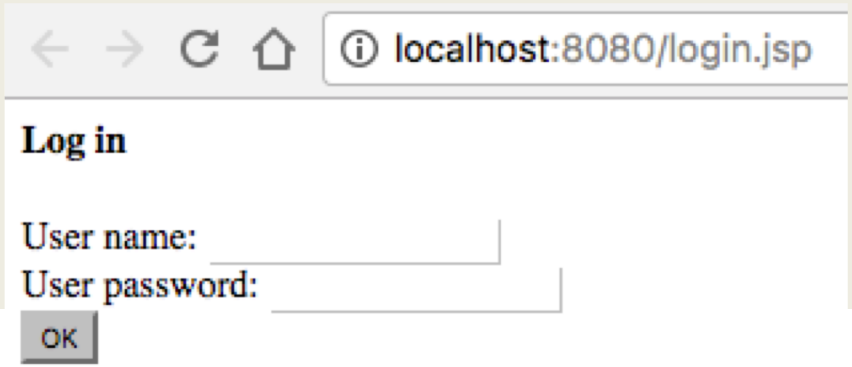
```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
  <title>Log in</title>
</head>
<body>
<h4>Log in</h4>
<h4>${requestScope.errorMessage}</h4>
<form action="login.do" method="POST">
  <label for="#username_field"> User name: </label>
  <input id="username_field" type="text" name="username" required/> <br/>
  <label for="#password_field"> User password: </label>
  <input id="password_field" type="password" name="password" required/><br/>
  <label class="login_field"> </label>
  <input type="submit" value="OK"/>
</form>
</body>
</html>
```



Oh no a big no-no!

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
  <title>Log in</title>
</head>
<body>
<h4>Log in</h4>
<h4>${requestScope.errorMe
<form action="login.do" met
  <label for="#username_fie
  <input id="username_field                                    ed/> <br/>
  <label for="#password_field"> User password: </label>
  <input id="password_field" type="password" name="password" required/><br/>
  <label class="login_field"> </label>
  <input type="submit" value="OK"/>
</form>
</body>
</html>
```

Actually, when you see a http:// protocol instead of an https:// protocol, you do NOT have safety on you connection, even though you cannot see the password in the form! But that issue is not within scope of this course.



Adding a deployment descriptor

- Go to Project Structure (Cmd + ; on Mac)
  - Facets -> Web -> Web Gradle (select your project)
  - Under pane Deployment Descriptors, click "+"
  - Select "web.xml"
  - In the path, put this: /src/main/webapp/WEB-INF/web.xml
  - You'll get something like this

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
      version="3.1">

</web-app>
```

Adding an entry to web.xml

- In web.xml, put the following snippet

```
<context-param>
   <param-name>admin_email</param-name>
   <param-value>admin@example.com</param-value>
</context-param>
```

Accessing an entry of web.xml

- In your servlets, use the web app variable like this

```
getServletContext().getInitParameter("admin_email")
```

- or in your jsp's like this

```
${initParam.admin_email}
```

Adding a session time-out

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
      version="3.1">

  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
</web-app>
```

The session-timeout entry makes your session short-lived

Creating a login servlet

- Create a servlet: new → servlet → LoginServlet.java → add urlPatterns = **"/login.do"** to the @WebServlet annotation
- Code is on the next page

Creating a session

```
@WebServlet(name = "LoginServlet", urlPatterns = "/login.do")
public class LoginServlet extends HttpServlet {
   protected void doPost(HttpServletRequest request, HttpServletResponse response) ... {
      String userName = request.getParameter("username");
      String password = request.getParameter("password");
      String viewJsp = "index.jsp";
      HttpSession session = request.getSession();
      //TODO remove at deployment!
      session.setMaxInactiveInterval(5);
      if (session.isNew() || session.getAttribute("userName") == null) {
         if(userName.equals("Henk") && password.equals("henk")) {
            session.setAttribute("userName", userName);
         } else {
            request.setAttribute("errorMessage", "Sorry, your credentials are not OK.
Please try again");
            viewJsp = "login.jsp";
         }
      }
      RequestDispatcher view = request.getRequestDispatcher(viewJsp);
      view.forward(request, response);
   }
```

make JSTL functionality available

- Open build.gradle and add the following two lines to the dependencies section (if not already present):

```
dependencies {
    providedCompile 'javax.servlet:javax.servlet-api:3.1.0'
    compile group: 'jstl', name: 'jstl', version: '1.2'
    compile group: 'taglibs', name: 'standard', version: '1.1.2'
    testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

Refactoring index.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <head>
    <title>My Hello World jsp</title>
  </head>
  <body>
    <c:choose>
      <c:when test="${sessionScope.userName != null}">
        <h1>Welcome, ${sessionScope.userName}</h1>
      </c:when>
      <c:otherwise>
        <h2>please log in first</h2>
        <c:redirect url="/login.do"></c:redirect>
      </c:otherwise>
    </c:choose>
    </body>
</html>
```

This is the whole page; we will revisit individual parts of it in the following slides

Refactoring to login using jsp tags

- Create a page directive that specifies the JSP tag library (jstl.jar)

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

Using </c:*tag* /> will instruct the use of the JSTL library tags

Refactoring to login using jsp tags

- Using JSTL tags <c:choose>, <c:when> and <c:otherwise> , create some login logic for processing the login status

<c:choose> is the parent for an if/else type switch

<c:when> is like an if/else if test. It should contain a test expression

```
<c:choose>
  <c:when test="${sessionScope.userName != null}">
    <h1>Welcome, ${sessionScope.userName}</h1>
  </c:when>
  <c:otherwise>
    <h2>please log in first</h2>
    <c:redirect url="/login.do"></c:redirect>
  </c:otherwise>
</c:choose>
```

<c:otherwise> is like an else (default) option.

Testing the pudding

- Start the server and run
  - login.jsp
  - login.do
- Enter username and password
  - False
  - Correct
- What is the result, what doe you see and what does the location bar say?

Review

- What aspects have we seen in this presentation?
  - web.xml deployment descriptor
  - session object usage and scopes
  - Expression Language (EL) usage ${}
  - jsp tag library (jstl.jar) <c:choose>
- We will explore these aspects individually in the next series of slides

web.xml deployment descriptor

- The web.xml file holds information that is relevant to your entire web application.
  - It tells Tomcat where to find classes and how they can be referenced in URLs
  - It holds information that is relevant to the whole application, such as the baseURL.
- Other examples of application-specific information that you may want to put into the web.xml file are:
  - admin email and contact information
  - database username and password (for now)
- To access web.xml init params from a servlet, use this code:

```
String param = servletConfig
      .getServletContext()
      .getInitParameter(“param_name");
```

Scopes

- As in all programming environments, scopes are really important in web applications.
- We have seen the three main ones:
- ***Application scope***. Specified in web.xml and accessed through
  - `${initParam.<attrib_name>}` in the jsp files
  - `getServletContext().getInitParameter("<attrib_name>")` in the servlet code
- ***Session scope***. When a session object is available, accessed through
  - `${sessionScope.<attrib_name>}` in the jsp files
  - `getSession.getAttribute("<attrib_name>")` or `getSession.setAttribute("<attrib_name>", <attrib> )`
- ***Request scope.*** Each http request generates its own HttpRequest object. It can be accessed through
  - `${requestScope.<attrib_name>}` in the jsp files
  - `request.getParameter("<attrib_name>")` or `request.setParameter("<attrib_name>", <attrib>)`

Expression Language

- Expression Language (EL) is a very simple form of syntax to get data from the different scope objects application, session and request.
- EL expressions all look like this: **`${}`**
- They usually have this form: **`${foo.bar}`** or **`${foo.bar...}`**
- The left part of the expression is either an EL implicit object or an attribute
- The main Implicit objects are:
  - pageScope
  - requestScope
  - sessionScope
  - applicationScope
- An attribute can be the name of an attribute stored in any of the four available scopes

Expression Language

- With this EL expression: **`${foo.bar}`**, foo is either a Map or a bean and bar is either a Map key or a bean property.
- For example, setting an object of class Person as a request attribute and using it in a JSP:

Person code:

```
class Person{
  private String name;
  public String getName(){
    return name
  }
  public void setName(String name){
    this.name = name;
  }
}
```

Servlet code:

```
Person p = new Person();
p.setName("Fred");
request.setAttribute("user",p);
```

JSP code:

```
<h1>Welcome user ${user.name}</h1>
```

Java beans

- Java beans are a special kind of class. They are coded according to a strict set of rules. That's why you can use them so conveniently in EL
- Java bean rules are:
  - They provide a no-arg constructor (e.g. `public Person(){}`)
  - They provide public standard-named getters and setters for class attributes; e.g. for String property `name` you must provide the methods
    `public String getName(){return "Fred"}` and
    `public void setName(String name){}`
  - Note that the class does not necessary have to have the actual property (private String name); only the getters and setters are required!

JSP standard actions: includes

- There are two ways to include files in JSPs: as a **directive** or as a **standard action**.
- As a **directive**, the include is done as a simple copy-and-paste
- As a **standard action**, the include is evaluated as a regular jsp and the response is inserted at the position of the include

JSP with an include **directive**:

```
<%@ include file="Header.jsp" %>
```

Here, the source is inserted

JSP with an include **standard action**:

```
<jsp:include page="Header.jsp" />
```

Here, the response is inserted

JSP standard actions: includes

- As a **standard action**, the include is processed like a regular jsp and the response is inserted

JSP to be included ("chapterHeader.jsp"):

```
<h2>this is my dynamic chapter header with chapter name
${param.chapterName}</h2>
```

Here, the source is inserted

JSP using the include with a **standard action**:

```
<html><body>
<h1>my story</h1>

<jsp:include page="Header.jsp" >
   <jsp:param name="chapterName" value="The end" />
</jsp:include>

</body></html>
```

JSP standard tag library: JSTL

- With JSTL tags, you can do a variety of actions without resorting to scripting
- In order to be able to use them, you have to put the jar in the lib directory
- Import them using the directive

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

- We will look at a few of them:
  - <c:foreach> to iterate over a arrays or collections
  - <c:if> a simple decision tool
  - <c:choose> a more complex decision tool
  - <c:set> to set a bean's property

JSTL tags

- With JSTL tags, you can do a variety of actions without resorting to scripting
- In order to be able to use them, you have to put the jar in the lib directory
- Import them using the directive

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

- We will look at a few of them:
  - <c:foreach> to iterate over a arrays or collections
  - <c:if> a simple decision tool
  - <c:choose> a more complex decision tool
  - <c:set> to set a bean's property

JSTL <c:if>

- The JSTL tag <c:if> can be used to conditionally display some piece of information.
- You generally use them in this form:

```
<c:if test='${param.p == "someValue"}'>
    Generate this template text if p equals someValue
</c:if>

<c:if test='${param.p}'>
    Generate this template text if p equals "true"
</c:if>
```

JSTL <c:choose>

- The JSTL tag <c:choose> can be used to conditionally choose between several options; it is an extension of the if form.
- You generally use them in this form:

```
<c:choose>
    <c:when test='${param.p == "0"}'>
        Generate this template text if p equals 0
    </c:when>
    <c:when test='${param.p == "1"}'>
        Generate this template text if p equals 1
    </c:when>
    <c:otherwise>
       Generate this template text if p equals
        anything else
    </c:otherwise>
</c:choose>
```

JSTL <c:foreach>

- The JSTL tag <c:foreach> can be used to iterate over an array or Collection.
- You generally use them to create tables:

```
<body>
   <table>
     <c:forEach var="person" items="${people.people}">
       <tr>
         <td>${person.name}</td>
         <td>${person.age}</td>
         <td>${person.height}</td>
       </tr>
     </c:forEach>
   </table>
 </body>
```

JSTL <c:set>

- The JSTL tag <c:set> can be used to set a property on a bean object
- General form:

```
<c:set var="str" value="..." />
```

- Examples:

```
<%-- Save data in scoped variables --%>
<c:set var="myname" value="Fred" scope="page" />
<c:set var="mycompany_name" value="ACME" scope="request" />

<%-- Show the saved values --%>
<c:out value='${pageScope.myname}' />
<c:out value='${requestScope.mycompany_name}' />
```

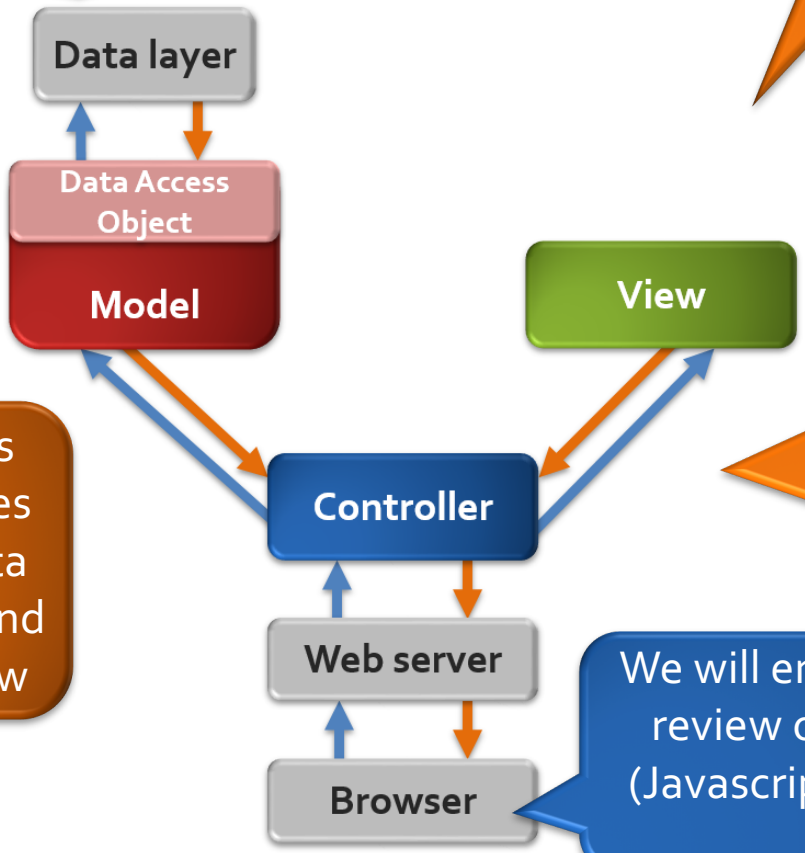
JSP tutorials

- A great resource on JSP tags and other issues is http://www.tutorialspoint.com/jsp/

Putting it all together

In another session, we will check out how the model can communicate with a data source that may change in the future

The view (JSP) takes the prepared data (beans, lists and maps preferably) and displays them without using code



The model processes the request and makes sure the returned data are readily available and accessible for the view

Now the Servlet is responsible for hadling requests and invoking correct parts of the model

We will end this course with a short review of client-side technology (Javascript, Ajax, css) for your web application