Workshop 1: A Basic Request Based Approach, the Java Servlet API

Objectives

The goal for this workshop is to expose a given OO-model on the web (a webshop). We'll create a basic request based application to interact with a part of the model (the ProductCatalogue). You need the following tools and skills;

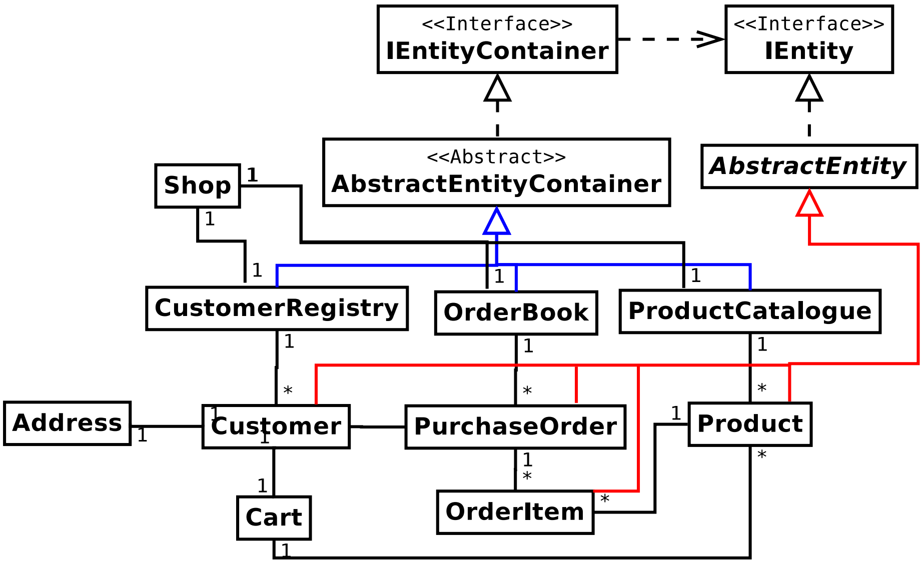
- Environment: NetBeans IDE (including Maven, Tomcat Server).
- Basic XML, HTML and CSS.
- JEE Web applications and the Servlet API (Servlets, JSP's, JSTL, ...).
- The FrontController JEE design pattern.
- The Post-Get-Request (PRG) pattern.

PLEASE: INSPECT CODE SAMPLES FROM THE LECTURES (ON COURSE PAGE)! EVERYTHING YOU NEED SHOULD BE THERE. WILL HOPEFULLY SAVE YOU A LOT OF TIME!

Final date : See Course Page

1 The shop model

We will use a basic model of a web shop during most of the workshops. Below is an UML class diagram of the model;



1. Download the model from course page > Workshops (it's a Maven Java (standalone) Application). The projects final structure is pictured in the Appendix, have a look (a lot is copy and paste and some is optional, don't panic...). Blue files are market as modified by Git using NetBeans Git plugin.

   Note In the skeleton code files are missing, you create and put in correct locations (as in Appendix).

2. Unzip and open project in NetBeans, inspect using Projects and Files-views (tabs). Possible NetBeans prompt for a "master password". If so choose any. In particular;

   a) Open the pom.xml. Note section <dependencies>!

      Note There's some code completion and also XML validation (right click). The pom must be valid! Make the pom invalid (comment out) and save! Project trashed (see icon Projects tab)! Make pom valid and save. Project restored!

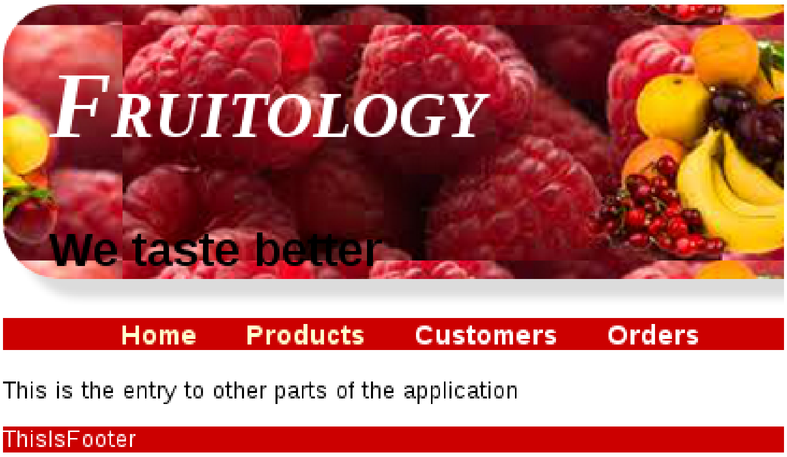
3. Switch of testing for ordinary builds: Tools > Options > Java > Maven, check "Skip Tests for any...."

4. Build project. Maven will sometimes download a lot. Inspect your local Maven repository (in folder ~/.m2). Somewhere in the repo you should find shop-1.0-SNAPSHOT.jar (the final delivery, i.e. the application). Maven will pick dependencies from the .m2-folder, if not found Maven will "automagically" download dependencies over the net and put in .m2 (if possible).

5. There's a test class with a few test's (JUnit). Run tests.

2 A request based front-end to the Shop model

The task for you is to implement the Products part of the application. Other parts are working (simulated). The main page of the application;



This is the entry to other parts of the application

ThisIsFooter

2.1 Preliminaries

1. Inspect Services-tab > Servers in NetBeans. You should find an Apache Tomcat server. Right click icon > Properties, inspect.

   Note If "Enable Http Monitor" is checked it's possible to inspect incoming HTTP requests in NetBeans Window > Debugging > HTTP Server Monitor. Useful in between.

2. Start Tomcat, right click > Start. Use a browser to visit http://localhost:8084 (default for Tomcat admin pages). Stop Tomcat.

   Note There are always two administrative applications running in Tomcat, shown as / and /manager. Don't touch!

3. There's an application skeleton on the course page. Download and open in NetBeans (there are compilation errors and possible warning "Project Problems". Ignore problems for now).

4. Fix the compilation errors by adding dependencies in the pom (this assumes you have the shop model installed in m2).

```
<dependency>
    <groupId>edu.chl.hajo</groupId>
    <artifactId>shop</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
```

```
</dependency>
```

5. Build the project: Mark project > right click, Build (possible a lot of Maven downloads).

   Note We'll use Java 1.7 (Java 7). Check project. Mark project, right click > Properties > Sources and Compile (both should say Java 1.7).

6. Run the project: Mark project > Properties > Run > Select Tomcat. Mark project > Run (products pages of application doesn't work).

7. Inspect project thoroughly! Compare code and run time behaviour.

8. Compare browser address field with content in file Web Pages/META-INF/context.xml. Change path in context.xml and run again. As expected?

   Tip Under some circumstances it's desirably to speed up the deployment of the project. Especially when testing small changes to server side code. It's possible to use "deploy on save", NetBeans will compile and deploy the application on every save (at severe exceptions possible have to Build/Run again). Use if you like...

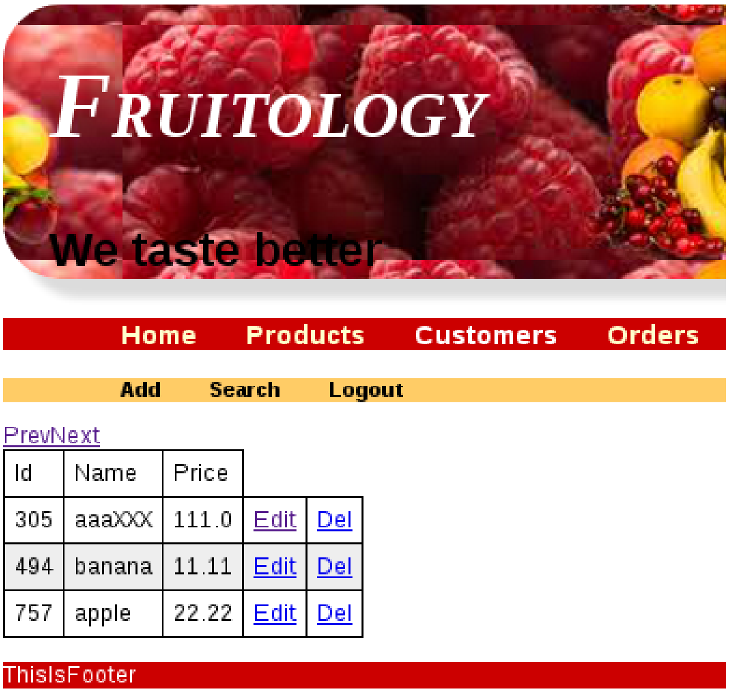
   In an existing application. Mark application, right click > Select Properties > Run > check Deploy on Save.

   Select Properties > Build > Compile > Compile On Save: For both application and test...

   Tip Indexing of local Maven repo is very time consuming. Select Tools > Options >Maven > Index Update Frequency: Never

2.2 Read the model

The first goal is to display a web page (the Products page, products.jspx) with a table of products. This page should be reachable from the Products-link in the main menu.



Id	Name	Price		
305	aaaXXX	111.0	Edit	Del
494	banana	11.11	Edit	Del
757	apple	22.22	Edit	Del

1. We need a reference to the model (Shop.INSTANCE). We'll store it in the ServletContext-object (will be accessible as long as application runs).

    a) Create the ContextListener (see appendix, a class implementing ServletContext Listener, New > Web > Web Application Listener, use defaults except name).

    b) In contextInitialized-method add (will be run at application start up);

    ```
    Logger.getAnonymousLogger().log(Level.INFO,
       "Putting Shop in application scope");
    sce.getServletContext().setAttribute(Keys.SHOP.toString(), Shop.INSTANCE);
    ```

    c) Add the SessionListener. Let i be a HTTP Session Listener. Add log messages to show when session is created/destroyed (and the session id).

    d) Run and watch Tomcat output window.

    Waring Check you web.xml for strange additions. Sometimes you miss a checkbox and then NetBeans possible add things.

2. Create the ProductServlet. Set urlPatterns = {"/products/*"}.

3. Create products.jspx, use index.jspx as a template. Use the JSTL's <c:forEach> and the HTML <table> tags to dynamically create a table. It should be possible to get the table data using something similar to:

    ```
    <c:forEach var="i" items="${requestScope.PRODUCT_LIST}">
    ```

```
    ...
    </c:forEach>
```

4. Continue until a table get displayed (see slides and code samples).

2.3 Table Navigation

1. Add the Prev and Next links.
2. Let ProductServlet handle calls and send new data to be displayed by products.jspx. Use the ContainerNavigator as a helper (store it into the HttpSession-object, Servlets should be stateless!)

2.4 Write the Model

1. Add the sub menu as a separate jspx to be included in all “products related” pages.
2. Add links with actions and/or views to the sub menu. All requests are handled by ProductServlet.
3. The Add menu selection will show the addProduct.jspx. Make it work!
4. Add the edit-links and delete links to the table (if not already there).
5. Let the edit link show an editProduct.jspx page and the del an delProduct.jspx page.
6. Make edit and del work (let Servlet use request data (id) to look up the selected product and put it into the HttpRequest-object and finally forward request to page).

3 The Post Redirect Get (PRG) Pattern

For now it’s possible to resend post requests when adding products i.e more products added by mistake.

1. Try to achieve this.
2. Make it impossible to resend post’s (the PRG-pattern). Add a redirect to products.jspx in the ProducServlet directly after adding the new product to the model (response.sendRedirect()).
3. Consider the same problem for edit and delete?

4 Authentication

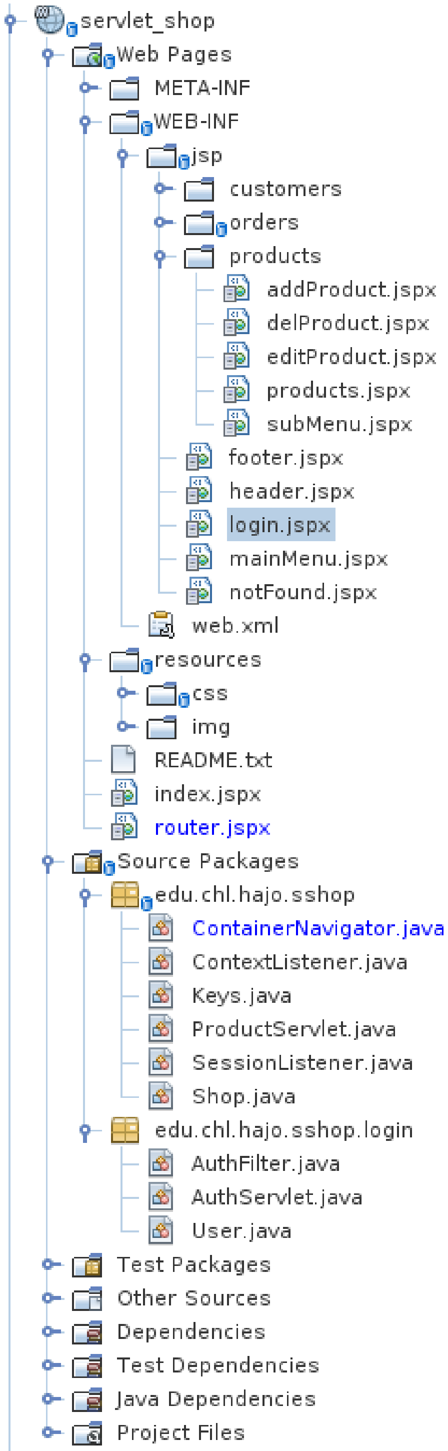
(Optional) The products pages are hidden but nothing prevents the user from entering some URI in the browser to access the pages (via ProductServlet). As a means of authentication we'll use a Filter. The general idea is;

- All request to URI "/products/*" will hit the filter.
- Filter checks if there's a user-object in the HttpSession-object.
- If so the request is passed through.
- Else there's a forward to some login page. Login is handled by an AuthServlet. If login succeeds the Servlet puts the user-object into the HttpSession. Else an error message is displayed in the login page.
- At logout the Servlet invalidates the session.

1. Create the AuthFilter and the AuthServlet and use the login.jspx to implement the authentication.

APPENDIX

Final Project content (some details missing).



- servlet_shop
  - Web Pages
    - META-INF
    - WEB-INF
      - jsp
        - customers
        - orders
        - products
          - addProduct.jspx
          - delProduct.jspx
          - editProduct.jspx
          - products.jspx
          - subMenu.jspx
        - footer.jspx
        - header.jspx
        - login.jspx
        - mainMenu.jspx
        - notFound.jspx
      - web.xml
    - resources
      - css
      - img
    - README.txt
    - index.jspx
    - router.jspx
  - Source Packages
    - edu.chl.hajo.sshop
      - ContainerNavigator.java
      - ContextListener.java
      - Keys.java
      - ProductServlet.java
      - SessionListener.java
      - Shop.java
    - edu.chl.hajo.sshop.login
      - AuthFilter.java
      - AuthServlet.java
      - User.java
  - Test Packages
  - Other Sources
  - Dependencies
  - Test Dependencies
  - Java Dependencies
  - Project Files