Workshop 2: A Service Oriented Approach, RESTful Web Services with JAX-RS

Objectives

Same as in previous workshop, we'll expose the shop (ProductCatalogue) on the web. You need the following;

- Tools as before and GlassFish 3.1.2.x
- More HTML, CSS and basic JavaScript
- JQuery (DOM and AJAX API's) and JQueryUI
- Java EE RESTful Web Services (JAX-RS)
- Cache control, conditional GET's and update's

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 GlassFish

We can handle GlassFish from inside NetBeans.

1. Go to Services tab > Servers > Mar GlassFish > Start
2. Mark GlassFish > View Domain Admin Console. A Web page should show up, note port. This is the administration tool for GlassFish. Peek around!
3. Stop Server.

2 The model as a RESTful web service

We'll design a RESTful web service for the ProductCatalogue. The projects final structure is in the Appendix.

1. Download the skeleton app from course page and open in NetBeans. Inspect Test Packages/.../cURL_README.txt to get an understanding of the REST API.

2.1 Implementing the Service

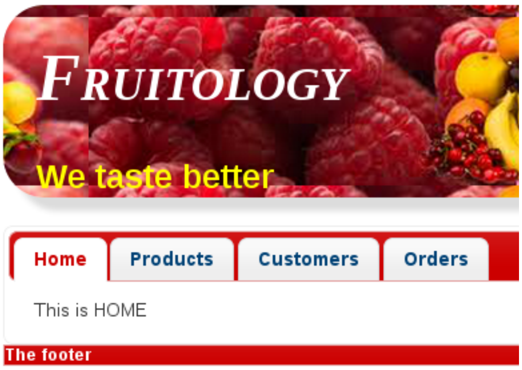
Warning There are annotations with the same names but from different packages! Must select the correct ones. For now just use from java.xml.bind or javax.ws.rs! Watch out!

We'll try to keep the model uncluttered, so no annotation in the shop model. We'll use wrapper classes.

1. Add JAXB-annotations to the ProductProxy class. Use XmlAccessType.PROPERTY and annotations on the getter-methods to access data from the wrapped Product. Let the the returned XML contain <product> element(s) (i.e. not <productProxy>-elements). There's a JUnit "pseudo"-test to dump the XML result. Use it!
2. Create a plain Java class ProductCatalogueResource as a wrapper for the ProductCatalogue in the model (i.e. all calls to ProductCatalogueResource will be forwarded to the model class).
   a) Annotate class with @Path("products").
   b) Inspect web.xml, there should be a ServletAdaptor and "Jersey" ServletContainer. If not check any code sample and compare. All URL's /rs/* should be handled by JAX-RS.
   c) Build the project. An icon, RESTful Web Services, should show up in the project (see Appendix image).
3. Implement a method getAll() in the resource class that just returns all Products. Try to run using GlassFish. Use cURL to test the method.
4. Now we'll implement and test the ProductCatalogueResource (the central piece of the application). Let ProductCatalogueResource have the same functionality as IProductCatalogue in shop model, except the method sort(). Don't implement the interface, we need different signatures.
   - All methods should return a Response-object with correct status code returned (OK, Created, etc.)
     - Must wrap Products if used as return values.
     - All methods should be able to return JSON and XML
     - It's not possible to return primitive types (a wrapper is supplied).
     - Method handling form data should use @FormParam
     - getRange-method should use @QueryParam (first (element) and nItems (number of returned items in list)
     - Generic classes (List<T> etc.) must be wrapped using the javax.ws.rs.core.GenericEntity<T class.

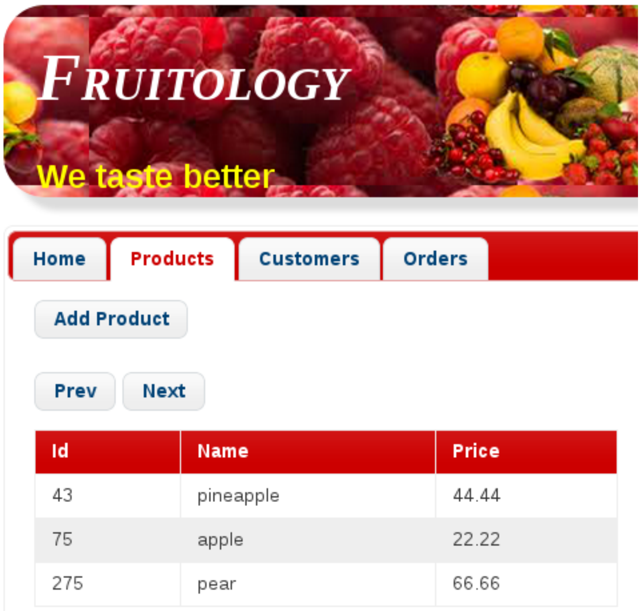
3 A client for the service

Now we'll develop a "single page" JavaScript client for the service similar to the previous workshop. A few screen shoots (uses JQueryUI for tabs and dialogs, tabs should work)...

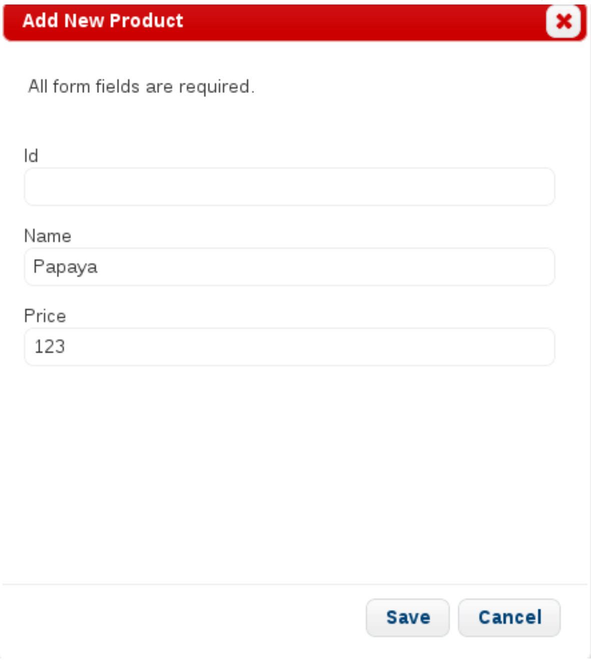


This is HOME

The footer



Id	Name	Price
43	pineapple	44.44
75	apple	22.22
275	pear	66.66

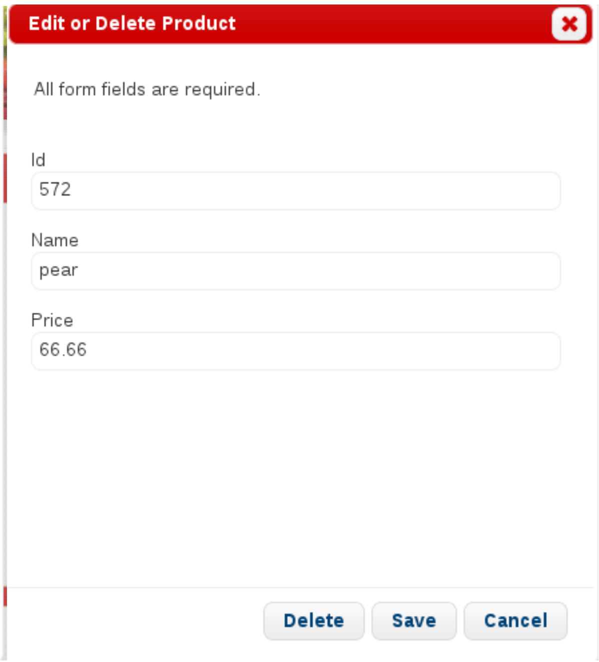


All form fields are required.

Id

Name: Papaya

Price: 123



All form fields are required.

Id: 572

Name: pear

Price: 66.66

Note The browser address field should stay the same all the time.

Tip To debug the running JavaScript use Chrome/Developer Tools or Firefox/Firebug or other.

Note We use JQuery, no low level JavaScript DOM or AJAX API!

3.1 A Proxy for the ProductCatalogue

We'll represent the server side ProductCatalogueResource as a JavaScript ProductCatalogue "class" on the client side (a remote proxy). The proxy shall have the same methods as the server side class.

1. Inspect navigator.js to see an example of the "pseudo classical" JS style.
2. Implement the proxy class using the "pseudo classical"- style in file productCatalogue.js. The single task for any methods should be to to execute an AJAX call to the REST service. All methods should return the JQuery "deferred"- object.

   Hint Many functions are one-liners.
3. There's some JS tests in /js/test. Use to test the proxy.

3.2 Event handling and DOM manipulation

The task is to forward calls from the products page via products.js to the productProxy and get the result back into the DOM (i.e. display result in products.html). We will use JQueryUI components, possible have a look at their home page. All JS code for this should be in the products.js file (downloaded dynamically when products.html is loaded).

1. Go to products.js
2. Implement the "rendering" functions (or at least some of). The table should have a listener. When clicking on a row the Edit/Delete dialog should pop up (populated with selected product). The Edit/Delete dialog shows a confirmation dialog before deleting.
3. Implement listeners for the buttons.
4. Continue until you have a fully functional JavaScript REST client.

3.3 A Java RESTclient

(Optional) It's rather easy to generate Java clients for any REST service.

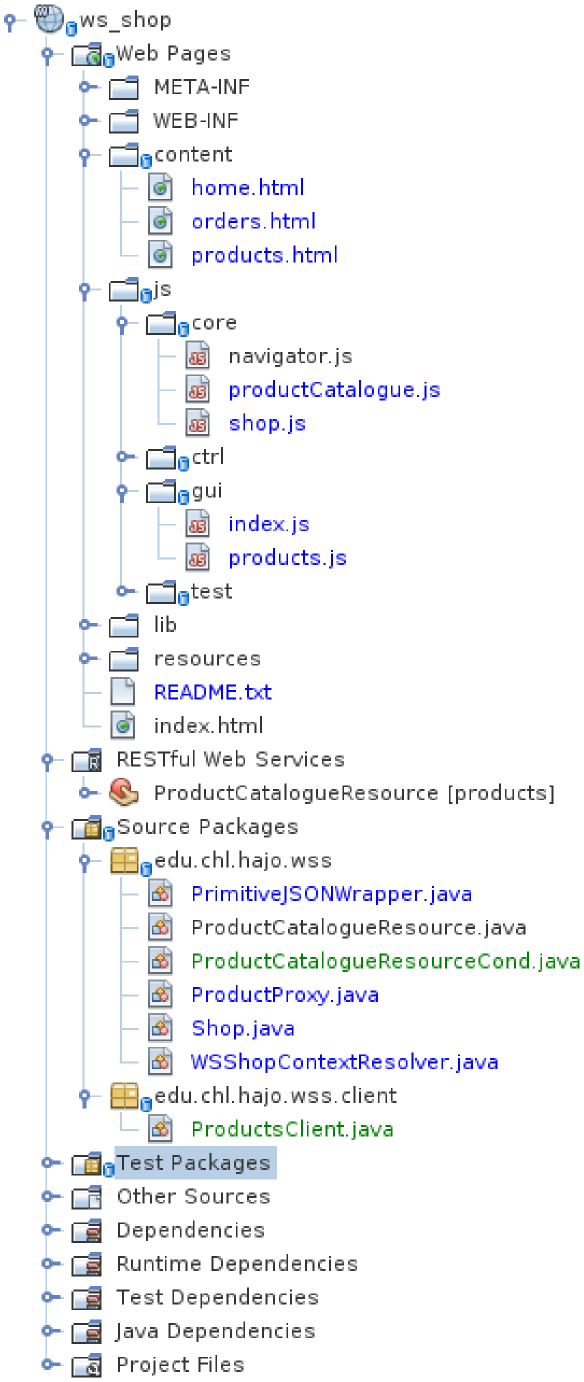
1. Create the ProductsClient for our product catalogue service. New > Web Service > RESTful Java Client > etc.
2. There's a JUnit test to try.

3.4 Cache control, conditional GET and update's

(Optional)

1. Refactor copy ProductsCatalogueResource and rename to ProductsCatalogueResourceCond. Change @Path.
2. Modify new resource. Add conditional gets and updates for relevant methods.
3. Check with cURL.

Appendix



```
ws_shop
  Web Pages
    META-INF
    WEB-INF
    content
      home.html
      orders.html
      products.html
    js
      core
        navigator.js
        productCatalogue.js
        shop.js
      ctrl
      gui
        index.js
        products.js
      test
    lib
    resources
    README.txt
    index.html
  RESTful Web Services
    ProductCatalogueResource [products]
  Source Packages
    edu.chl.hajo.wss
      PrimitiveJSONWrapper.java
      ProductCatalogueResource.java
      ProductCatalogueResourceCond.java
      ProductProxy.java
      Shop.java
      WSShopContextResolver.java
    edu.chl.hajo.wss.client
      ProductsClient.java
  Test Packages
  Other Sources
  Dependencies
  Runtime Dependencies
  Test Dependencies
  Java Dependencies
  Project Files
```