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

Objectives

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

- Tools as before and GlassFish 4.x (replacing Tomcat)
- JavaScript core language (no APIs) and AngularJS (very basic Jasmine for testing)
- Java EE RESTful Web Services (JAX-RS aka Jersey, runtime bundled with GlassFish)
- Optional: Cache control, conditional GET's and PUT's

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

Final date : See course page.

1 Design

The figure below shows the structure of the application during execution (arrows are associations).

- HTML/CSS is not shown (Bootstrap is used). The visual design should be very similar to the previous workshop.
- Client-side downloaded js-files (Bootstrap and AngularJS files are not shown). The structure of the client- side MVC is given by AngularJS. The services.js file will host AJAX calls to the server-side resource. This service is called from the controller which contains "key/mouse-listeners" and intermediate model-data (automatically rendered in the view by Angular data bindings). The application starts in app.js (similar to main(...)). All data is sent as JSON.
- On server side incoming calls are handled by ProductCatalogueResource that consumes JSON. The output data is also returned in the form of JSON.
- ProductWrapper is a wrapper class around the Product class from the model. It is needed because the Product in the model is immutable (JAXB/JAX-RS also requires a non-private default constructor).

- ApplicationConfig is a purely technical class used by JAX-RS. Generated by NetBeans.
- SingletonShop is a wrapper around the shop model to get a single shop object.
- Filter is used later on the server-side for authentication/authorization purposes. See Authentication below.
- Filter on client-side (Angular filter). Not used for now. Can be used during rendering to filter output.

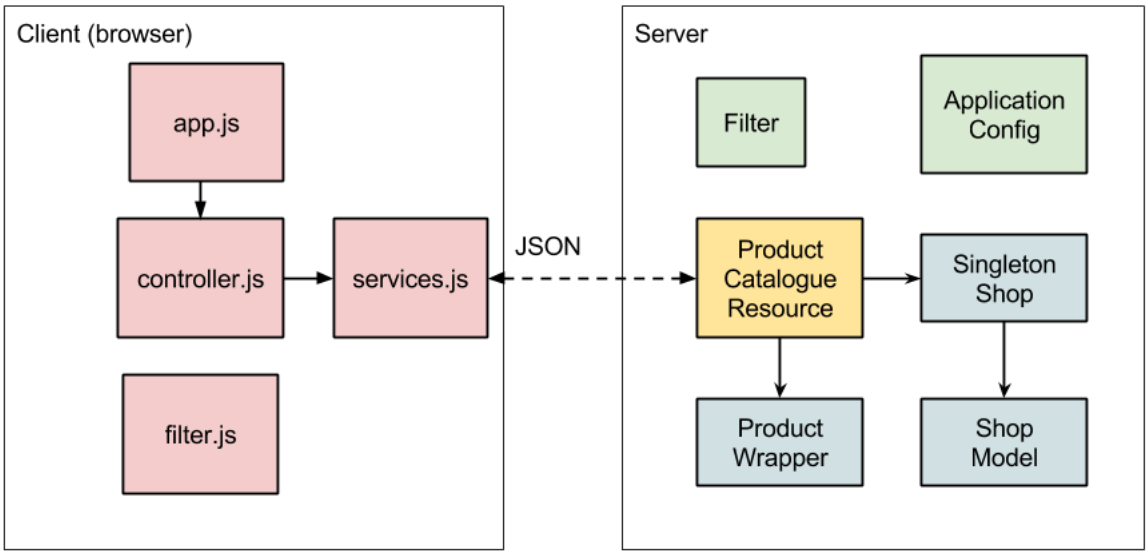


Figure 1: Runtime structure

2 GlassFish

Server will run on GlassFish. We'll handle GlassFish from inside NetBeans.

1. Go to Services tab > Servers > Mark GlassFish > Properties. Inspect (HttpMonitor!)
2. Services tab > Servers > Mark GlassFish > Start
3. Mark GlassFish > View Domain Admin Console. A Web page should show up, note port. This is the administration tool for GlassFish. Peek around!
4. Close the admin console and stop the server.

3 Implementation

3.1 Web service back-end

The workshop is contained in a single Maven project for both front- and back-end. We'll start with the back-end (server-side). The projects final development structure is in the Appendix.

1. Download the code skeleton from the course page and open it in NetBeans. Inspect. Make sure the dependency on the shop model is present and working.
2. It's should be possible to run the code. Try (it will not work)!
3. The only back-end class that you need to complete (and modify) is the ProductCatalogueResource (PCR).
4. The PCR is a JAX-RS root resource class responsible for exposing the ProductCatalogue to clients. It should have the same methods as the IEntityContainer (in shop) but with different input and output types. The return type should be "Response" for all methods. PCR converts to the correct type before and after the call to ProductCatalogue. The methods must of course be mapped to URLs/HTTP-methods and have the correct produces/consumes annotations.
5. Implement each method in PCR and use cURL (or similar) to issue HTTP requests to the service. See Test Packages > cURL_README.txt.

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

Tip You must use the GenericEntity class as a wrapper before adding Lists (or other generics) to Response.

Tip You can't return primitive types. Use JsonObject as a wrapper.

3.2 Web service front-end

Now we'll develop a "single page" AngularJS/JavaScript client for the service.

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

1. The files to work with are in Web Pages/shop/app/js and Web Pages/shop/app/partials/products.
2. Start with the services.js, an AngularJS service module. Add a factory that will create the service object (named ProductResourceProxy or similar). The service object should have the same methods as PCR. It will act as a proxy for the back-end resource class. The service only responsibility is to issue AJAX requests on

the PCR. Let all methods return promises (calling methods on $http will yield a promise).

3. (Optional) Create some Jasmin tests to test the service (or do it in some other way, debugger...). Use the ... /shop/test folder.

4. (Once again, check samle code!) The goal is to be able to list the products. Continue in parallel with the $routeProvider (in app.js), products.html and controllers.js. Start out with a very simple page and controller. Let the controller do some basic call on the service. . Call chain: URL -> routeProvider -> control -> service -> back to control setting return value as a $scope variable - > page, using ng-directives (and {{ ... }}) to loop over result in variable.

5. Add list navigation.

6. Master-Detail: Complete product-detail.html (for edit and delete) and product-new.html. Create controls for each.

4 Cache control, conditional GET and updates

(Optional)

1. Refactor ProductsCatalogueResource and rename it to ProductsCatalogueResource-Cond. Change @Path to /cond.

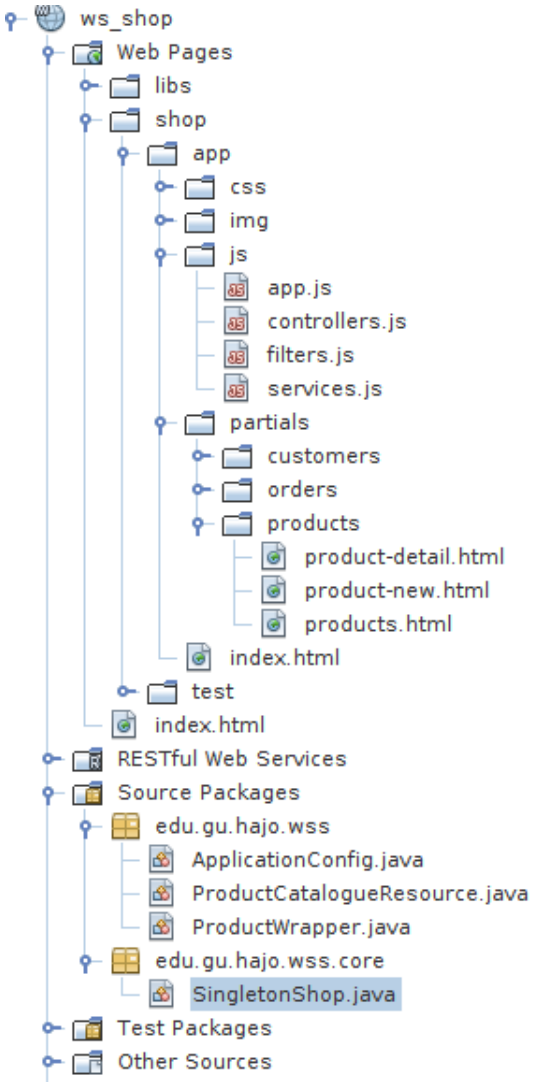
2. Modify the new resource. Add conditional gets and updates for relevant methods.

3. Check with cURL (new tests need to be created). See code samples.

5 Authentication and Authorization

(Optional) Add some authorization using some OAuth provider. Refer to the code samples on how to use Twitter. A hybrid approach like storing the authentication data inside the session is acceptable. Please note, however, that using the session is not considered to be a “clean” RESTful solution.

Appendix


ws_shop
Web Pages
libs
shop
app
css
img
js
app.js
controllers.js
filters.js
services.js
partials
customers
orders
products
product-detail.html
product-new.html
products.html
index.html
test
index.html
RESTful Web Services
Source Packages
edu.gu.hajo.wss
ApplicationConfig.java
ProductCatalogueResource.java
ProductWrapper.java
edu.gu.hajo.wss.core
SingletonShop.java
Test Packages
Other Sources