Brewery workshop @Ordina

Thank you for participating in the hands-on lab. Are you ready to brew beer with Java EE 7 and Java 8? The hands-on lab uses a set a business cases that introduces you to some of the Java EE and Java 8 features.

Prerequisites

The following software needs to be installed in order to get started:

JDK 1.8

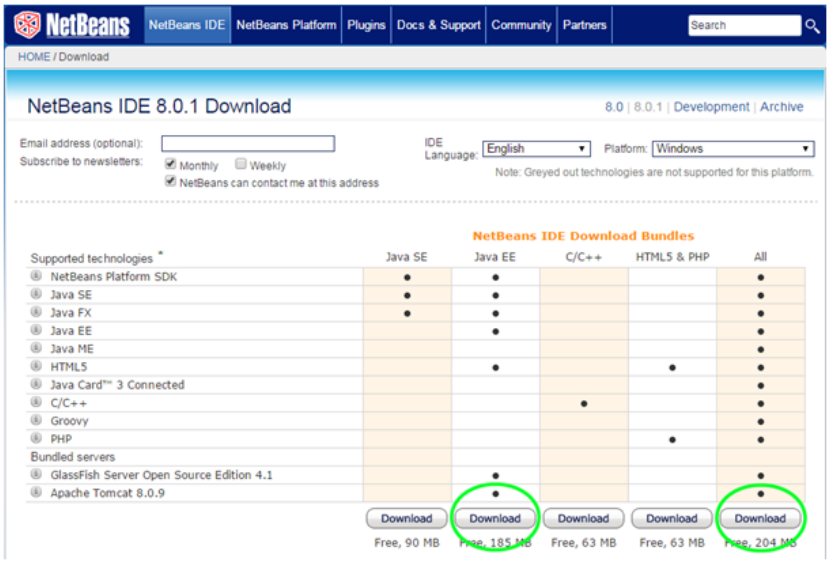
Download and install JDK 8 from
http://www.oracle.com/technetwork/java/javase/downloads/index.html

Maven 3.x

Download and install Maven from
http://maven.apache.org/download.cgi

NetBeans 8.0.1

Select the “Java EE” or “All” download from https://netbeans.org/downloads/
NetBeans contains Glassfish. When installing NetBeans, choose to install the pre-bundled Glassfish that comes with NetBeans.



Using your own tools

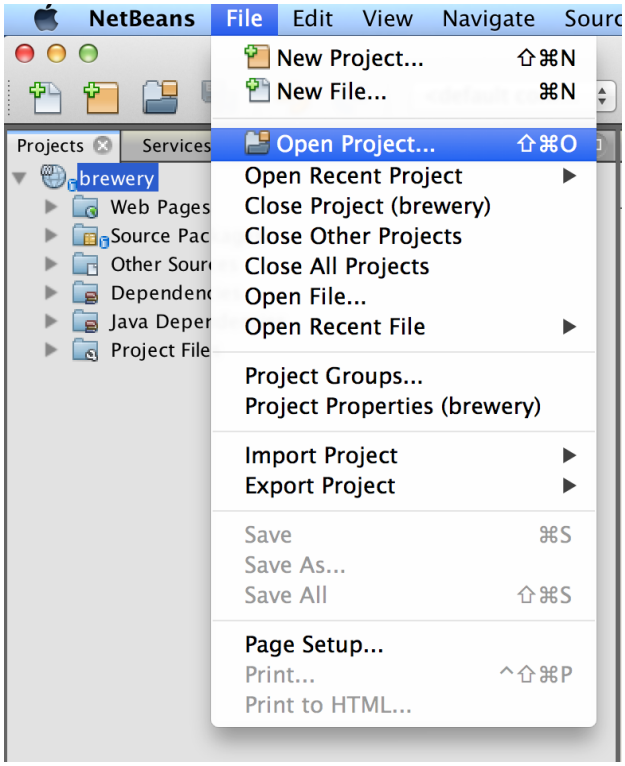
Of course you are free to bring your own tools (IntelliJ, Eclipse, etc.). However, this does imply you are willing to take the risk of environment specific problems. The workshop is built and tested in NetBeans, and should work out of the box from there.

Source code
You can download the source code at:
https://github.com/EelcoMeuter/BreweryWorkshop

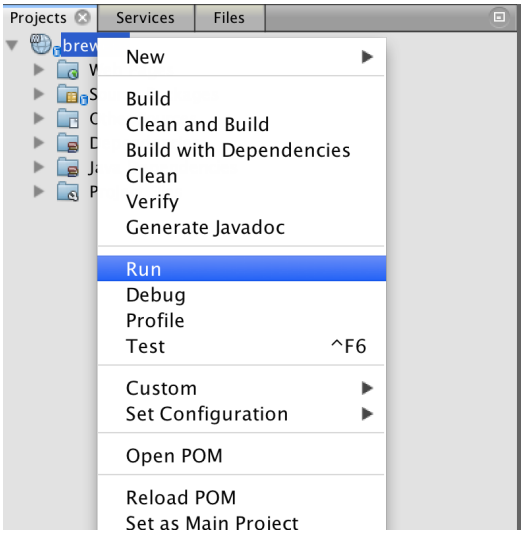
Browser
We use Chrome as our default browser. The application is not tested on all browsers as the application is only there for demonstration purposes. We kindly point you to the following shortcut keys if you are unfamiliar with Chrome:
https://developer.chrome.com/devtools/docs/shortcuts

Running the project

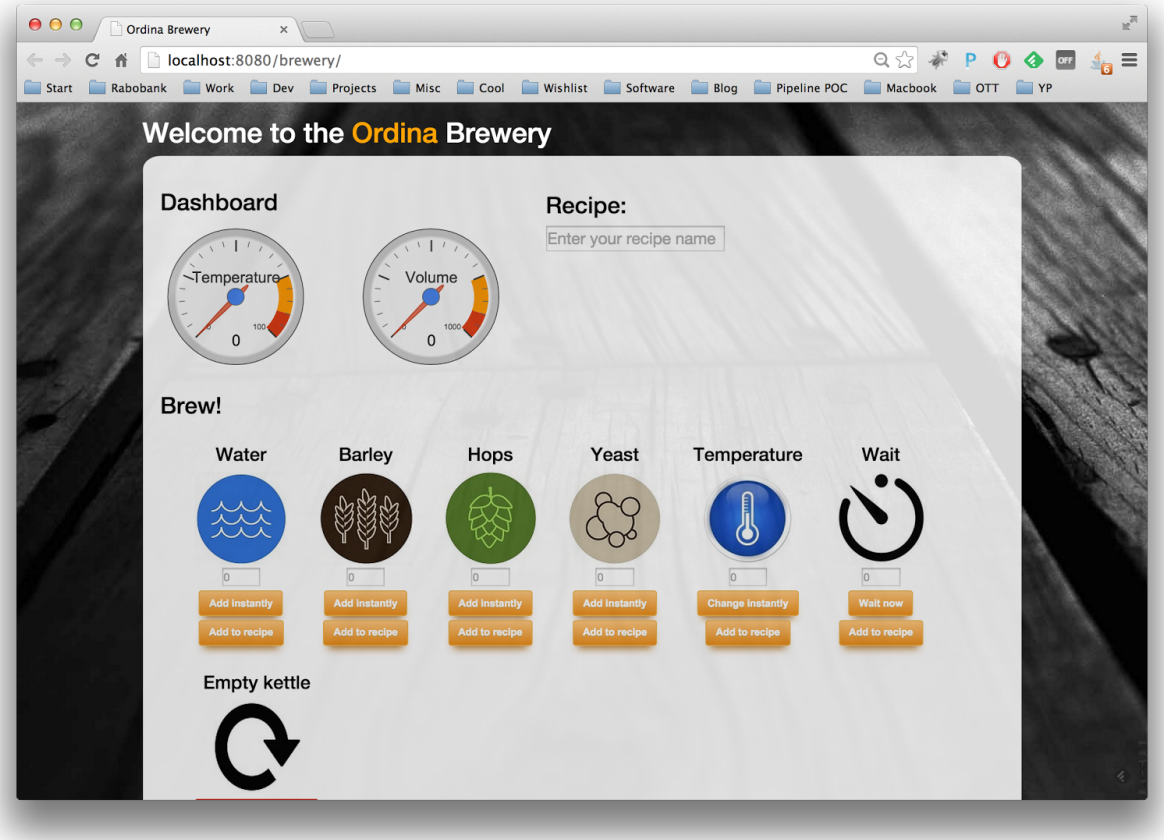
Step 1: Import the downloaded source code into NetBeans.



Step 2: Run the project

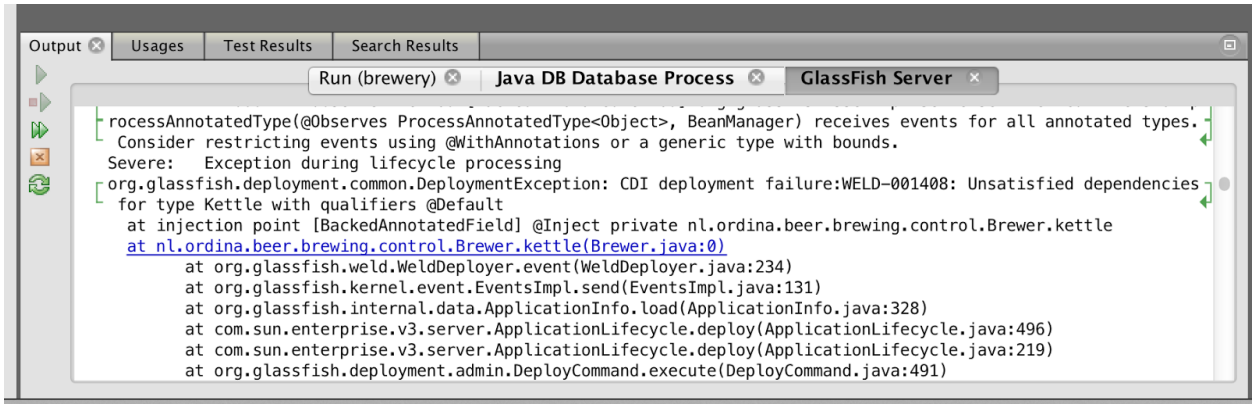


If you are setup correctly your browser should automagically open and show the following:

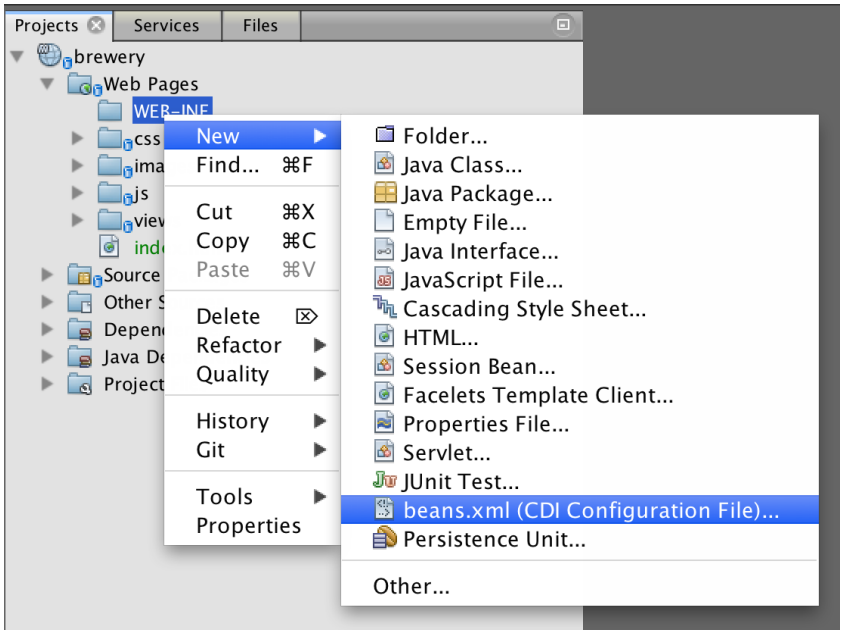


Step 3: Fixing CDI

The app is up and running, but you will probably have noticed the nice exception stack trace in the output of your server.



We can fix this by adding an empty `beans.xml` file in the WEB-INF folder. Fortunately NetBeans is smart enough to help us with this task.



We’re almost there. By default NetBeans will create a `beans.xml` file that will only look for annotated beans. Just change ‘`annotated`’ to ‘`all`’ and the exceptions should disappear.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns
       xmlns:xsi="http://www.w3.org/2001/
       xsi:schemaLocation="http://xmlns.j
       bean-discovery-mode="annotated">
</beans>
```

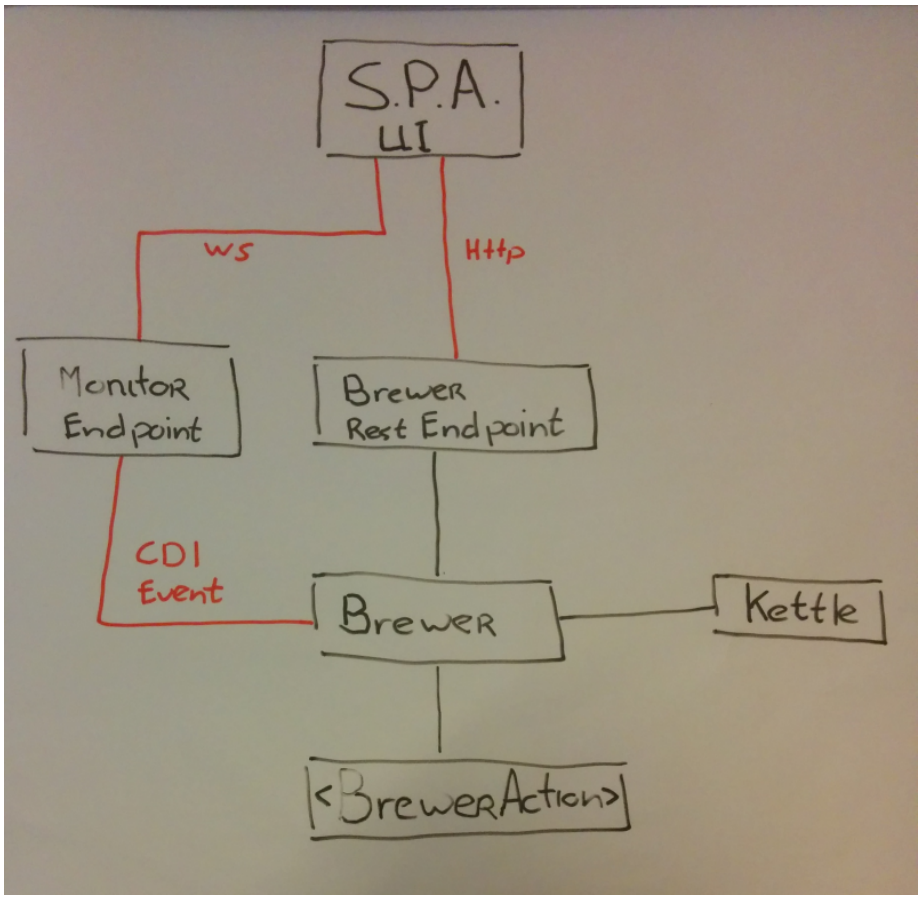
Would you like to know more? See http://docs.oracle.com/javaee/7/tutorial/doc/cdi-adv001.htm.

Context

He is a wise man who invented beer. - Plato

We are going to build a beer brewery monitoring application. This application allows you to brew beer with a few simple commands. You can add ingredients, adjust the temperature, wait and reset the process. You can also define your own recipe to automatically brew beer.

The application high level design



The User Interface is a Single Page Application (SPA) written in AngularJS. This application sends commands to the Brewer REST endpoints.

These commands are passed as `BrewerActions` to the Brewer. The Brewer places these actions on an internal queue and starts processing the different actions from the queue.

The brewers fires an event as soon as a `BrewerAction` has been completed. The `EventMonitor` observers these events and notifies the SPA via a web socket.

Business cases

Goal: Making the app RESTful

We have to tell our JAXRS implementation that our application uses resources. We've already added a file called `RestApplication` where you can add your code. Hint: you need more than just an annotation!

Goal: Add ingredient to Kettle

If you followed the prerequisites, you have the Angular application up and running in your browser.

Please open the developer tools in your browser and open the tab `Console`. Now try to add an ingredient e.g. Water by pressing the **Add Instantly** button.
The console will produce a 404 error code. You can use the error message to discover the required path. Note that this path is relative to the path defined in `RestApplication`.

Let's start implementing this functionality. We have to add the configuration and implementation of the `BrewerResource`. You need to configure this endpoint so that the 404 code disappears.

The next step is to process the actual request. Please have a look at the JavaDoc of `BrewerResource` method, which you just configured and implement the described logic.

The `EventMonitor` sends feedback about the brewing process to all registered clients. You need to implement and configure this class. The comments in the Javadoc help you to implement the functionality.

The `EventMonitor` communicates over web sockets. You need to implement a custom encoder to unmarshal a java object to a JSON object. This is an implementation of the interface `javax.websocket.Encoder.Text<T>`. You need to implement this interface in the inner class `AddIngredient.Encoder` and configure a reference to this class in the `EventMonitor` configuration. Just ignore the `init` and `destroy` methods.

Please use the following snippet to implement the encode method:

```
@Override
public String encode(final AddIngredient ai) throws EncodeException {
    return Json.createObjectBuilder().add("event", "ingredient added")
          .add("ingredient", Json.createObjectBuilder()
          .add("name", ai.ingredient.getName())
          .add("volume", Json.createObjectBuilder()
                .add("value", ai.ingredient.getVolume().getValue())
                .add("unit", ai.ingredient.getVolume().getUnit().name())
                .build())
          .build())
   .build().toString();
}
```

If you implemented this correctly, you will see some feedback in console of the browser.

Goal: Get and Empty the kettle

Now try to empty the kettle by pressing the **Empty Instantly** button.
The console will produce a 404 error code. The procedure of implementing this method is similar to the previous tasks.

So we need to reconfigure the `BrewerResource` and `EventMonitor`. Now you only need to implement the required logic in the `BrewerResource` and you are good to go.

Goal: Manage the temperature

The procedure of implementing this method is similar to the previous tasks and needs little further explanation.

You notice the requirement of a `DurationJsonAdapter`, which is an implementation of a XmlAdapter. As you have seen in the AddIngredient task, JSON objects are 'automagically' marshalled to POJO's. This is done with JAXB. `Duration` is part of the JDK, but not supported by default by JAXB. Therefore you need to implement an XMLAdapter that will provide the logic to (un)marshal a `Duration` object.

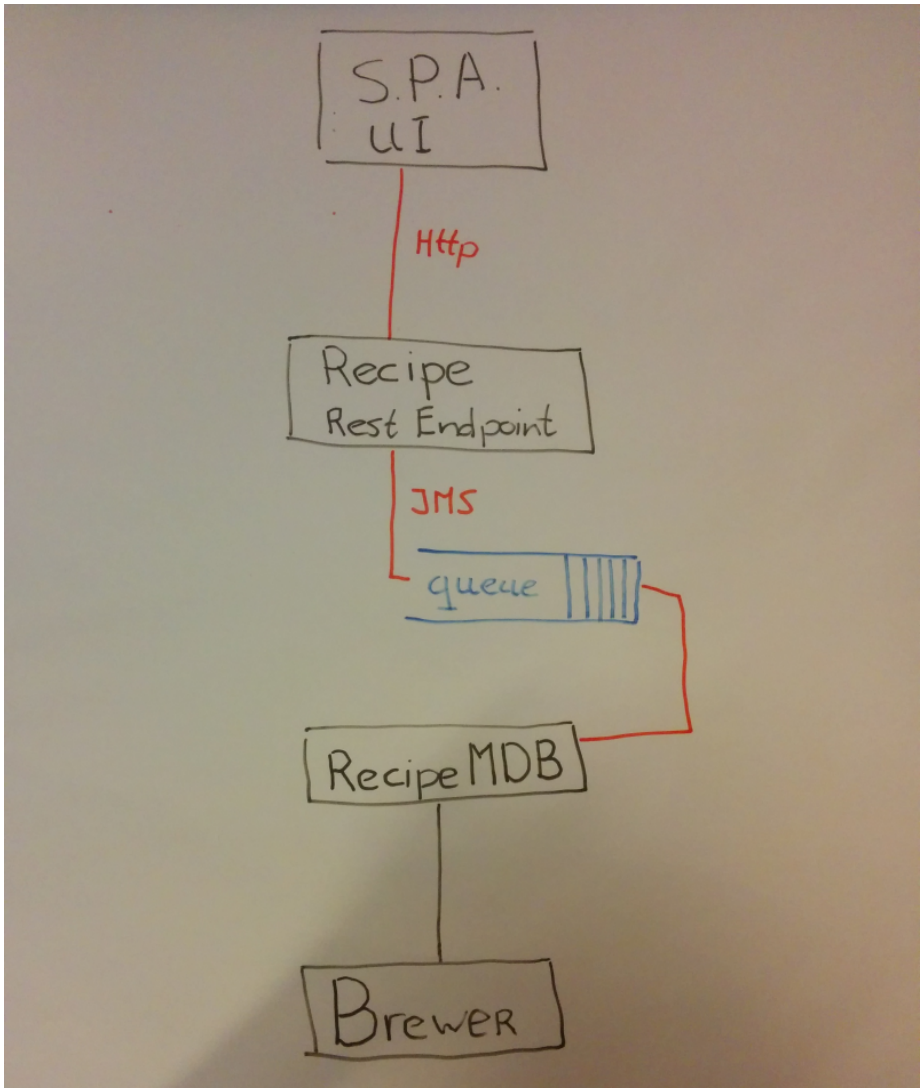
Congratulations!

Got everything working up to here? Nice! You have completed the first part. Now it's time to let the application work for you and start brewing recipes.

Goal: Process beer brew recipes

We are now able to manually brew beer. The next step is to implement some new functionality to let the application brew a recipe for you. There is another button in the UI that allows you to add an action to a recipe.

This functionality is designed as follows:



A Recipe JSON object is sent to a `RecipeResource`, which places the action on a JMS queue. A message driven bean listens to the queue and passes the actions to the `Brewer`.

We follow the same procedure as we did earlier. First open the `RecipeResource` and configure the endpoint and dependencies. Next you will need to implement the business logic. You need to try to create a `JMSContext` and produce a message on the `JMSQueue`. The message is the recipe itself.

When this is done, we build an endpoint that listens to messages on the queue. This is the `RecipeMessageDrivenBean`. In JavaEE 7 it is possible to do the entire queue configuration with annotations only!

We are reusing the `Brewer` object, so we need to inject this class together with a helper class to unmarshal the JSON message to a POJO and we are good to go.

Refactoring and extending the code

During this hands-on lab we implemented some functionality and made use of some neat new Java API's. You are now asked to refactor the code. What can be done more elegantly or in such a way that it improves the implementation and the understanding of the functionality.

For instance, the custom encoders implement some redundant code which is also not necessary in the web socket configuration. Try to refactor this code to remove all redundant code and make your code leaner. Maybe you found some other examples.

Just play with the code and improve it to your liking and discuss your finding with the other participants and trainers.

Don't know where to start? Here are some ideas:

- We don't do any validation. What happens if some smart user starts fiddling with the input? JSR 303 could help here.
- Did you spot the possible stack overflow?
- The REST endpoints are now blocking. Can you make them asynchronous?
- For the rest, use you imagination :-)

Final thoughts

We would like to know your view on this Hands-on lab. Please tell us what you liked or disliked, so we can improve. Thank you in advance.

Remko de Jong
Koen Olijve
Martijn Blankestijn
Pim Verkerk
Talip Ozkeles
Philippe Tjon-a-hen