Micro Profile Hands on Lab

Dmitry Alexandrov, Ivan St. Ivanov

Version 0.9, 15 November 2016

Table of Contents

Introduction

The Java MicroProfile

Magazine Manager

To showcase the MicroProfile we'll develop together a sample app. Let's suppose that it is used by an owner of a magazine to manage the various parts of their business. It consists of four microservices, each of which can be scaled and upgraded separately.

The responsibility of the *Authors* microservice is to keep track of the authors that are writing for the magazine. An author has the following properties:

```
public class Author {
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private boolean isRegular;
    private int salary;
}
```

In the initial version the microservice should support the following operations:

- Retrieving all the authors
- Finding author by its names
- Finding author by its ID
- Adding an author

The *Content* microservice takes care of the articles submitted by the authors as well as the comments on those articles. There are two domain objects in it:

```
public class Article {
    private Long id;
    private String title;
    private String content;
    private String author;
    private List<Comment> comments = new ArrayList<>();
}

public class Comment {
    private Long id;
    private String author;
    private String content;
}
```

Its initial functionality covers things like:

- Retrieving all the articles along with their comments
- Finding an article by its ID
- Retrieving all the articles for an author
- Creating an articles
- Adding a comment to an existing article

We'll go into more details in the Persistence chapter why in `Article` we don't refer directly to the `Author` entity from the *Authors* microservice. For now let's assume that we want to explore the Database per service pattern.

The third microservice in our list is the *Advertisers*. As its name implies it contains operations around the magazine's advertisers:

```
public class Advertiser {
    private Long id;
    private String name;
    private String website;
    private String contactEmail;
    private SponsorPackage sponsorPackage;
}

public enum SponsorPackage {

    GOLD(1000), SILVER(500), BRONZE(100);

    private int price;

    SponsorPackage(int price) {
        this.price = price;
    }

    public int getPrice() {
        return price;
    }
}
```

The common functionality that we'll provide with this microservice are these:

- Get all the advertisers
- Find advertiser by name
- Find all the advertisers with a certain sponsor package
- Add an advertiser

Last but not least comes the *Subscriber* microservice. Its domain model is as simple as that:

```
public class Subscriber {
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String address;
    private LocalDate subscribedUntil;
}
```

And the operations that we are going to implement are:

- Get the list of all subscribers
- Find subscriber by ID
- Retrieve all the expiring subscriptions
- Add a subscriber

These four microservices at the end will be configured to run on the four runtimes supporting the MicroProfile (one microservice per server). Basically you will be able to deploy each separate service on every runtime. However, to keep things simple, we did the following breakdown:

- *Authors* running on Apache TomEE
- *Content* running on IBM Liberty Profile
- *Advertisers* running on WildFly Swarm
- *Subscribers* running on Payara Micro

In order to follow better the next steps in this lab, we would suggest that you pick the same setup.

Initial project

We've sketched for you an initial maven project containing the four microservices as Maven subprojects. It is located under the `lab/magman` directory of this lab. Use your favorite IDE to import that project. Make sure that it is imported as Maven project for the IDEs that it is not the default structure.

You'll notice that besides the four microservices there is a simple pom project that groups the three specs:

```
<properties>
    <cdi-version>1.2</cdi-version>
    <jaxrs-version>2.0.1</jaxrs-version>
    <jsonp-version>1.0</jsonp-version>
</properties>

<dependencies>
    <dependency>
        <groupId>javax.enterprise</groupId>
        <artifactId>cdi-api</artifactId>
        <version>${cdi-version}</version>
    </dependency>
    <dependency>
        <groupId>javax.ws.rs</groupId>
        <artifactId>javax.ws.rs-api</artifactId>
        <version>${jaxrs-version}</version>
    </dependency>
    <dependency>
        <groupId>javax.json</groupId>
        <artifactId>javax.json-api</artifactId>
        <version>${jsonp-version}</version>
    </dependency>
</dependencies>
```

By depending on that project instead of `javaee-api`, we will make sure that the microservices we develop will not leak a non-MicroProfile technology like EJB.

The we simply depend on the project by adding the following dependency in the four microservices:

```
<dependency>
    <groupId>bg.jug</groupId>
    <artifactId>microprofile-dependencies</artifactId>
    <version>${project.version}</version>
    <type>pom</type>
    <scope>provided</scope>
</dependency>
```

Thus they can use the three MicroProfile specs. The only thing you need to do is to run in the `lab/magman` directory:

```
mvn clean install
```

After that, you can go to the target directory of each microservice and you'll notice one little war each. This is the standard format to deliver web applications in Java EE. So far.

How this lab works

In the next few chapters we'll guide you through the process of building a CDI, JAX-RS, JSON-P application, packing it in fat jar instead of war and dealing with persistence. In each chapter we'll show you the different implementation aspects of two of the microservices (*Author* and *Content*), while the other two we'll leave to you (with some hints from our side).

You've may also noticed the `sources` directory in the root of this repository. You can always consult it if the hints are not helpful enough and you don't have whom to ask. Besides the solution for the current version of the lab, it contains other features that will probably enter in future extensions that we plan to provide.

Core Micro Profile technologies

In this chapter of the lab we are going to see how we can implement basic business functionality using the three MicroProfile 1.0 technologies:

- CDI
- JAX-RS
- JSON-P

We assume that you have some basic knowledge about Java EE, but if you don't, don't worry. You'll be able to follow along.

We'll start with adding a data access object with a method that returns all the available records of a given resource. Then we'll make sure that this is exposed as a RESTful webservice. Finally we'll show you how you can transform the Java representation of the domain object to JSON.

Dependency Injection and object lifecycle

The first thing we'll do is to create the data access object for the authors, make it return all the authors and inject it in `ResourceAuthor`.

Go to the *authors* project in your IDE and under the `bg.jug.magman.authors` package create the package `persistence`. Inside the newly created package add the following class:

```
@javax.enterprise.context.ApplicationScoped
public class AuthorsDAO {
}
```

The annotation (you may add its package as import instead) makes sure that the CDI container will create only one instance of the above class (we'll use the term *bean* from now on). Now let's add the retrieve method. For the time being it will return hard coded data:

```
@ApplicationScoped
public class AuthorsDAO {
    ArrayList<Author> authors = new ArrayList<>();
    Author bilboBaggins = new Author("Bilbo", "Baggins", "bilbo@shire.com", true,
1000);
    Author spiderman = new Author("Spider", "Man", "spiderman@comics.com", false, 860
);
    Author captainPower = new Author("Captain", "Power", "power@futuresoldiers.com",
true, 750);

    authors.add(bilboBaggins);
    authors.add(spiderman);
    authors.add(captainPower);

    return authors;
}
```

Finally, let's inject this bean into our resource class. If you still don't understand the role of `ResourceAuthors`, bear with us.

```
public class ResourceAuthors {

    @Inject
    private AuthorsDAO authorsDAO;

    public String getAuthors() {
        return "authors";
    }

}
```

Make sure that you import the missing types.

What the above construction does is telling the CDI container when creating the `ResourceAuthors` class, to inject that single instance of `AuthorsDAO`.

So far so good. Before we proceed to the next section, let's do together the `ArticlesDAO` (in the *Content* microservice):

```
package bg.jug.magman.content.persistence;

@ApplicationScoped
public class ArticleDAO {

    public List<Article> getAllArticles() {
        List<Article> articles = new ArrayList<>();
        Article article1 = new Article("Bulgarian JUG's 2015", "2015 is over and 2016
is a week old now. However, I can't forget the past year, which happened to be the most
```

```
active one for the Bulgarian JUG, where I happen to be one of the co-leads. And what a
year it was! We had everything: seminar talks with local and foreign speakers, hands
on labs, Adopt OpenJDK and Adopt a JSR hackathons, a code retreat and a big
international conference. In this blog post I will briefly go through all the events
that kept our community busy in 2015.", "Bilbo Baggins");
        Article article2 = new Article("Primitives in Generics, part 3", "In the
Bulgarian JUG we had an event dedicated to trying out the OpenJDK Valhalla project's
achievements in the area of using primitive parameters of generics. Our colleague and
blogger Mihail Stoynov already wrote about our workshop. I decided, though, to go in a
little bit more details and explain the various aspects of the feature.", "Spider Man
");
        Article article3 = new Article("Primitives in Generics, part 2", "Whenever the
OpenJDK developers want to experiment with a concept they first create a dedicated
OpenJDK project for that. This project usually has its own source repository, which is
a fork of the OpenJDK sources. It has its page and mailing list and its main purpose
is to experiment with ideas for implementing the new concept before creating the Java
Enhancement Proposals (JEPs), the Java Specification Requests (JSRs) and committing
source code in the real repositories. Features like lambdas, script engine support,
method handles and invokedynamic walked this way before entering the official Java
release.", "Spider Man");
        Article article4 = new Article("Primitives in Generics, part 1", "Java
generics is one of its most widely commented topics. While the discussion whether they
should be reified, i.e. the generic parameter information is not erased by javac, is
arguably the hottest topic for years now, the lack of support for primitives as
parameter types is something that at least causes some confusion. It leads to applying
unnecessary boxing when for example you want to put an int into a List (read on to
find out about the performance penalty). It also leads to adding "companion" classes
in most of the generic APIs, like IntStream and LongStream for example.", "Spider Man
");
        Article article5 = new Article("JavaOne, day 4", "The last day at JavaOne
started as usual with the community keynote. I didn't go to it, because I wanted to
have a rest after the Aerosmith and Macklemore & Ryan Lewis concert last night and
also wanted to catch up with my blogs. However, the people that I follow on twitter
were kind enough to update me with the most interesting bits of the session.
Additionally, there's already a blog from Ben Evans about it.", "Captain Power");
        Article article6 = new Article("JavaOne days two and three", "In March this
year I had great time at the JavaLand conference. Along with other great people, I met
there the freelancer and blog author Roberto Cortez. He told me that he is going to
send a few session proposals to JavaOne and asked me whether I wanted to join him for
the Java EE Batch talk. I hadn't heard much about that topic at that time, but I
agreed. Then the proposal got accepted and here I am at JavaOne now. What do you know
", "Captain Power");

        articles.add(article1);
        articles.add(article2);
        articles.add(article3);
        articles.add(article4);
        articles.add(article5);
        articles.add(article6);

        return articles;
```

```
    }
}
```

You may have noticed the last parameter in the `Article` constructor. That matches to the first and last name of a particular records from the *Authors* microservice.

Finally, let's inject the DAO in the `ContentResource`:

```
public class ContentResource {

    @Inject
    private ArticleDAO articleDAO;

    public String getArticles() {
        return "articles";
    }

}
```

Exposure to clients

Now that we have the data access logic in place, let's make sure that it is available for the clients of our resources. The way to do it in the MicroProfile is via a JAX-RS resource. In order to enable at all JAX-RS, we will need to create a special marker class.

Let's first do it for the *Authors* microservice. Create a new class called `Application` under `bg.jug.magman.authors.rest`. Make it extend `javax.ws.rs.core.Application` and annotate it with `@javax.ws.rs.ApplicationPath`:

```
package bg.jug.magman.authors.rest;

import javax.ws.rs.ApplicationPath;

@ApplicationPath("/authors")
public class Application extends javax.ws.rs.core.Application {
}
```

The value of the `@ApplicationPath` annotation (in our case `"/authors"`) implies that all the JAX-RS resources exposed by the *Authors* microservice will be located under the /authors path.

Do the same in the *Content* microservice:

```
package bg.jug.magman.content.rest;

import javax.ws.rs.ApplicationPath;

@ApplicationPath("/content")
public class Application extends javax.ws.rs.core.Application {
}
```

Now let's turn our resource classes to full-fledged JAX-RS endpoints. In order to do that they need to be annotated with `@Path`, they need to have a proper CDI scope (`@RequestScoped` is good enough) and need to have at least one method annotated with any of the JAX-RS verb annotations (`@GET`, `@POST`, etc.). Let's first change the `ResourceAuthors` class:

```
@Path("/")
@RequestScoped
public class ResourceAuthors {

    @Inject
    private AuthorsDAO authorsDAO;

    @GET
    public String getAuthors() {
        return "authors";
    }
}
```

Now let's return the results of the `authorsDAO.getAuthors()` method:

```
@GET
public String getAuthors() {
    return authorsDAO.getAuthors().toString();
}
```

Now we can build again the authors.war and deploy it on any application server.

Once you do it, try to access `http://localhost:8080/authors/authors` (assuming 8080 is the default HTTP port). The above path is formed by the war name ("authors"), the application path ("authors") and the resource path (empty string). You should be able to see our three initial authors:

```
[{id=null, firstName='Bilbo', lastName='Baggins', email='bilbo@shire.com',
isRegular=true, salary=1000}, {id=null, firstName='Spider', lastName='Man',
email='spiderman@comics.com', isRegular=false, salary=860}, {id=null,
firstName='Captain', lastName='Power', email='power@futuresoldiers.com',
isRegular=true, salary=750}]
```

The *Content* resource is not much different than the *Authors*:

```
@Path("/")
@RequestScoped
public class ContentResource {

    @Inject
    private ArticleDAO articleDAO;

    @GET
    public String getArticles() {
        return articleDAO.getAllArticles().toString();
    }

}
```

Communication format

So far the responses of our resources were in plain text format. However, most of the contemporary clients (web, mobile, etc.) are using JSON format for communicating with the backend. That is why we should not return plain string, but rather JSON formatted one. We should also tell our clients that the response that they get is in that format, so that they can treat it properly. Last, but not least, a good practice for the endpoints is to return `javax.ws.rs.core.Response` objects. Thus they can encapsulate additional meta information besides the mere payload (like the HTTP response code or some headers for example).

First things first. Let's add functionality in our Author domain object to convert itself into JSON. For that we'll use the JSON-P API - the third core technology in MicroProfile.

```
public JsonObject toJson() {
    JsonObjectBuilder result = Json.createObjectBuilder();

    result.add("lastName", lastName)
          .add("firstName", firstName)
          .add("email", email)
          .add("salary", salary)
          .add("regular", isRegular);
    if (id != null)
        result.add("id", id);

    return result.build()();
}
```

The entry point here is the `Json.createObjectBuilder()` method. Once we get hold of the `JsonObjectBuilder` instance, we can use its `add` method to add the varioud attributes of our JSON object. At the end, we retrieve the built `JsonObject` and return it.

Back in the resource class, we need to create a JSON array out of our author JSON objects. We use again the JSON-P API:

```
@Path("/")
@RequestScoped
public class ResourceAuthors {

    @Inject
    private AuthorsDAO authorsDAO;

    @GET
    public String getAuthors() {
        List<Author> authors = authorsDAO.getAuthors();
        List<JsonObject> authorJsons = authors.stream()
                                        .map(Author::toJson)
                                        .collect(Collectors.toList());
        JsonArrayBuilder arrayBuilder = Json.createArrayBuilder();
        authorJsons.forEach(arrayBuilder::add);
        JsonArray result = arrayBuilder.build();
    }

}
```

Finally, let's change the return type to `Response`, build an instance, set its HTTP code to 200 and return it:

```
@GET
public Response getAuthors() {
    List<Author> authors = authorsDAO.getAuthors();
    List<JsonObject> authorJsons = authors.stream()
                                    .map(Author::toJson)
                                    .collect(Collectors.toList());
    JsonArrayBuilder arrayBuilder = Json.createArrayBuilder();
    authorJsons.forEach(arrayBuilder::add);
    JsonArray result = arrayBuilder.build();

    return Response.ok(result).build();
}
```

Last, but not least, our method should specify that it produces JSON instead of plain text:

```
@Produces(MediaType.APPLICATION_JSON)
public Response getAuthors() {
  //
}
```

Now, if you deploy the authors application and access it, you should get a proper JSON:

```
[{"lastName":"Baggins","firstName":"Bilbo","email":"bilbo@shire.com","salary":1000,"regular":true},{"lastName":"Man","firstName":"Spider","email":"spiderman@comics.com","salary":860,"regular":false},{"lastName":"Power","firstName":"Captain","email":"power@futuresoldiers.com","salary":750,"regular":true}]
```


The implementation for the *Content* microservice is almost the same. The only difference comes from the fact that the `Article` domain object itself contains `Comment` objects:

```
public class Comment {

    // fields and methods skipped

    public JsonObject toJson() {
        JsonObjectBuilder builder = Json.createObjectBuilder();
        builder.add("id", id);
        builder.add("author", author);
        builder.add("content", content);

        return builder.build();
    }
}

public class Article {

    // fields and methods skipped

    public JsonObject toJson() {
        JsonObjectBuilder builder = Json.createObjectBuilder();
        if (id != null) {
            builder.add("id", id);
        }
        builder.add("title", title);
        builder.add("content", content);
        builder.add("author", author);
        if (comments != null)
            builder.add("comments", getCommentsArray(comments));
        return builder.build();
    }

    private JsonArray getCommentsArray(List<Comment> comments) {
        JsonArrayBuilder builder = Json.createArrayBuilder();
        comments.forEach(comment -> builder.add(comment.toJson().toString()));
        return builder.build();
    }
}
```

And then in the resource class:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response getArticles() {
    List<Article> articles = articleDAO.getAllArticles();
    List<JsonObject> articlesJson = articles.stream()
            .map(Article::toJson)
            .collect(Collectors.toList());
    JsonArrayBuilder arrayBuilder = Json.createArrayBuilder();
    articlesJson.forEach(arrayBuilder::add);
    JsonArray result = arrayBuilder.build();

    return Response.ok(result).build();
}
```

And now what?

You may go on and implement the MicroProfile trinity for the other two microservices. Some hints:

- You may consider using JAXB for some of the remaining microservices (as we did for *Advertisers* in the solution). JAXB is part of Java SE (not EE technology), which makes it available anyway
- Consider formatting the `subscribedUntil` field of the `Subscriber` when turning it to JSON. One working format is `"yyyy-MM-dd"`

Make jar, not war

So far what we showed was nothing different than a usual Java EE application with a stripped number of technologies. But the delivery model stays the same: war deployed into an application server. The current trend in microservices development is to build an executable jar that packs everything - the application code plus the application server.

The fat jar approach is not yet standardized (it's not even talked to be standardized). However, it is one of the requirements for the MicroProfile. That is why it is implemented in a quite different way between the application server vendors. In the following subsections we'll show you how you can configure the Maven build to spawn executable jar besides the deployable war.

Another thing to make sure is the path to the RESTful resources. So far it was `/<war-name>/<application-path>/<resource-path>` Which resulted for example to `/authors/authors`. We want to avoid repeating one and thing twice and make it look like the application was deployed in the application server root.

WildFly Swarm

For WildFly Swarm we need to adds its plugin and dependency to pom.xml. As we decided earlier on, we'll deliver the *Advertisers* microservice on that runtime, so open its POM file.

You should first add the WildFly Swarm Maven plugin:

```
<project>
    ....
    <build>
        <finalName>advertisers</finalName>
        <plugins>
            ....
            <plugin>
                <groupId>org.wildfly.swarm</groupId>
                <artifactId>wildfly-swarm-plugin</artifactId>
                <version>${version.wildfly-swarm}</version>
                <executions>
                    <execution>
                        <id>package</id>
                        <goals>
                            <goal>package</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
    ....
</project>
```

As the plugin and dependency share the same version, it is a good practice to put it in the POM properties section:

```
<properties>
    ....
    <version.wildfly-swarm>2016.9</version.wildfly-swarm>
</properties>
```

Finally add the WildFly Swarm dependency:

```
<dependencies>
    ....
    <dependency>
        <groupId>org.wildfly.swarm</groupId>
        <artifactId>microprofile</artifactId>
        <version>${version.wildfly-swarm}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

After building the project with Maven, you will notice `advertisers-swarm.jar` in the target directory. You can run it with

```
java -jar advertisers-swarm
```

If you've worked on this microservice so far, you should be able to access the resource at http://localhost:8080/advertisers

Apache TomEE

Add the TomEE maven plugin to the plugins section of your pom.xml:

```
<build>
    <finalName>ROOT</finalName>
    <plugins>
        ....
        <plugin>
            <groupId>org.apache.tomee.maven</groupId>
            <artifactId>tomee-maven-plugin</artifactId>
            <version>7.0.1</version>
            <executions>
                <execution>
                    <id>build-exec</id>
                    <phase>package</phase>
                    <goals>
                        <goal>exec</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <tomeeClassifier>webprofile</tomeeClassifier>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Notice the `finalName` element. We set it to `ROOT` so that the resource is exposed directly at the server root path.

After running `mvn clean install`, you'll find a ROOT-exec.jar in the target directory. You can run it like that (from the `authors` directory):

```
java -jar ROOT-exec.jar
```

And the authors resource should be available at `http://localhost:8080/authors`

IBM Liberty Profile

IBM Liberty profile has also its dedicated Maven plugin. It needs much more things to configure though:

```
<plugin>
    <groupId>net.wasdev.wlp.maven.plugins</groupId>
    <artifactId>liberty-maven-plugin</artifactId>
    <executions>
        <execution>
            <id>create-server</id>
            <phase>pre-integration-test</phase>
            <goals>
```

```
                <goal>create-server</goal>
            </goals>
        </execution>
        <execution>
            <id>start-server</id>
            <phase>pre-integration-test</phase>
            <goals>
                <goal>start-server</goal>
            </goals>
            <configuration>
                <verifyTimeout>60</verifyTimeout>
            </configuration>
        </execution>
        <execution>
            <id>deploy-app</id>
            <phase>pre-integration-test</phase>
            <goals>
                <goal>deploy</goal>
            </goals>
            <configuration>
                <appArchive>${project.build.directory}/content.war</appArchive>
            </configuration>
        </execution>
        <execution>
            <id>stop-server</id>
            <phase>pre-integration-test</phase>
            <goals>
                <goal>stop-server</goal>
            </goals>
            <configuration>
                <serverStopTimeout>60</serverStopTimeout>
            </configuration>
        </execution>
        <execution>
            <id>package-server</id>
            <phase>pre-integration-test</phase>
            <goals>
                <goal>package-server</goal>
            </goals>
            <configuration>
                <assemblyInstallDirectory>
${project.build.directory}</assemblyInstallDirectory>
                <packageFile>${project.build.directory}/content.jar</packageFile>
                <include>runnable</include>
            </configuration>
        </execution>
    </executions>
    <configuration>
        <assemblyArtifact>
            <groupId>com.ibm.websphere.appserver.runtime</groupId>
            <artifactId>wlp-microProfile1</artifactId>
```

```
            <version>16.0.0.3</version>
            <type>zip</type>
        </assemblyArtifact>
        <userDirectory>${project.build.directory}</userDirectory>
        <serverName>contentServer</serverName>
    </configuration>
</plugin>
```

If you want your resorce to be exposed directly at the root path, add this content to content/src/main/webapp/WEB-INF/ibm-web-ext.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-ext
    xmlns="http://websphere.ibm.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee
http://websphere.ibm.com/xml/ns/javaee/ibm-web-ext_1_0.xsd"
    version="1.0">

    <reload-interval value="3"/>
    <context-root uri="/"/>
    <enable-directory-browsing value="false"/>
    <enable-file-serving value="true"/>
    <enable-reloading value="true"/>
    <enable-serving-servlets-by-class-name value="false" />

</web-ext>
```

This time, after running the maven build, you'll get content.jar file in the target directoy. Run it: `java -jar content.jar`.

By default the server runs on port 9080, so you can access the articles at `http://localhost:9080/content`

Payara Micro

In Payara micro you simply run a Java main class from the `payara-microprofile` artifact:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <dependencies>
        <dependency>
            <groupId>fish.payara.extras</groupId>
            <artifactId>payara-microprofile</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <id>payara-uber-jar</id>
            <phase>package</phase>
            <goals>
                <goal>java</goal>
            </goals>
            <configuration>
                <mainClass>fish.payara.micro.PayaraMicro</mainClass>
                <arguments>
                    <argument>--port 8080</argument>
                    <argument>--noCluster</argument>
                    <argument>--autoBindHttp</argument>
                    <argument>--logo</argument>
                    <argument>--deploy</argument>
                    <argument>${basedir}/target/ROOT.war</argument>
                    <argument>--outputUberJar</argument>
                    <argument>${basedir}/target/ROOT.jar</argument>
                </arguments>
                <includeProjectDependencies>true</includeProjectDependencies>
                <includePluginDependencies>true</includePluginDependencies>
                <executableDependency>
                    <groupId>fish.payara.extras</groupId>
                    <artifactId>payara-microprofile</artifactId>
                </executableDependency>
            </configuration>
        </execution>
    </executions>
</plugin>
```

We told it to build file ROOT.jar. You also need to create a `glassfish-web.xml` file in the `subscribers/src/main/webapp/WEB-INF` directory with the following content:

```
<glassfish-web-app>
    <context-root>/</context-root>
</glassfish-web-app>
```

What about persistence?

Other interesting features