

# Micro Profile Hands on Lab

Dmitry Alexandrov, Ivan St. Ivanov

Version 0.9, 15 November 2016

# Table of Contents

Introduction .....	1
The Java MicroProfile .....	1
Magazine Manager .....	1
Initial project .....	3
How this lab works .....	5
Core Micro Profile technologies .....	6
Dependency Injection and object lifecycle .....	6
Exposure to clients .....	9
Communication format .....	9
Micro Services Enablement.....	10
Make jar, not war .....	10
WildFly Swarm .....	10
Apache TomEE.....	10
IBM Liberty Profile .....	10
Payara Micro .....	10
Running on non-default HTTP port.....	10
WildFly Swarm .....	10
Apache TomEE.....	10
IBM Liberty Profile .....	10
Payara Micro .....	10
What about persistence? .....	11
Other interesting features .....	12

# 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.

Then 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 Stoyanov 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;
```

```
}  
}
```

And let's inject it in the `ContentResource`:

```
public class ContentResource {  
  
    @Inject  
    private ArticleDAO articleDAO;  
  
    public String getArticles() {  
        return "articles";  
    }  
  
}
```

## Exposure to clients

## Communication format

# Micro Services Enablement

## Make jar, not war

WildFly Swarm

Apache TomEE

IBM Liberty Profile

Payara Micro

## Running on non-default HTTP port

WildFly Swarm

Apache TomEE

IBM Liberty Profile

Payara Micro

# What about persistence?

## Other interesting features