MicroProfile Workshop

Introduction

What is MicroProfile? In 2016, a group of vendors and individuals started an initiative to optimize Enterprise Java for a micro-services architecture.

It builds on top of a few Java EE specifications, CDI and JAX-RS mainly, and defines some specifications towards resilience, distributed, reactive, and serviceability. At the end of 2020, the MicroProfile Working Group was formed that defines the processes and governance structures so that MicroProfile now delivers actual specifications.

Requirements

In order to follow this workshop, you need the following software installed on your laptop.

- JDK 8 or JDK 11
- Maven
- An IDE of your choice like IntelliJ, Netbeans, or Eclipse.
- A program which can expand a ZIP file.
- A WebBrowser
- A Rest client like PostMan or CURL.

MicroProfile Starter

MicroProfile Starter is an online tool to generate a project that incorporates the MicroProfile frameworks.

- It is not geared towards a runtime of a specific vendor. This will strengthen the image that MicroProfile is a collaborative initiative. It also highlights the standardization factor, that a MicroProfile-based application can be used on any of the implementations.
- It also generates examples for the specifications, getting you a quick start in the process of learning the features of each of these specifications.

We will use the starter project to generate a project which we will expand with some more examples of the MicroProfile Specification.

- Go to https://start.microprofile.io/
- Select the MicroProfile Version 4.0.
- Select your favorite runtime.
- Click 'Select All' specification
- Click on the Download Button

- Expand the downloaded zip file
- Open the console and set the current directory to the `service-a` directory of the expanded ZIP file (contains the pom.xml file)
- Run the `mvn package` command.
- Look at the `readme.md` file (located in the same directory as the pom.xml file) for the next steps to get the demo application up and running (they are slightly different depending on the implementation you choose)
- Run the application through `java -jar target/demo-microbundle.jar` or similar.
- Get the URL for the main test page and use it in your browser like http://localhost:8080/demo/index.xhtml
- Click on the `Hello JAX-RS endpoint` link.

MicroProfile implementations

There are 2 types of MicroProfile Implementations.

The first type is those that are based on Jakarta EE implementations. Since MicroProfile is using some of the Jakarta EE implementations, JAX-RS, CDI, and JSON-P, any implementation compatible with Jakarta EE can be used as a basis for a MicroProfile implementation. So besides the MicroProfile frameworks, you can also use all the Jakarta EE features. Examples are Payara Server, OpenLiberty, WildFly, and Apache TomEE.

The second type of implementation is based on JAX-RS implementations like Eclipse Jersey. These implementations assemble around the JAX-RS implementation and all other required frameworks. Examples are KumuluzEE and Helidon.

JAX-RS

One of the core concepts within MicroProfile is JAX-RS, used for creating web services according to the Representational State Transfer (REST) architectural pattern.

If you know JAX-RS already, you can skip this section and go to the next topic. The rest of this chapter is only a short introduction to JAX-RS. Much more can be found on the internet, and it will all apply to MicroProfile as JAX-RS is an integral part of MicroProfile.

Configuration

There are many aspects of JAX-RS that you can configure, but most of the time, it is enough to just define the URL part for the front controller. Let us have a look at this configuration.

- Open the generated `service-a` project with the IDE of your choice (to make it easier)
- Go to the Java class `<artifact>RestApplication`, for example `com.example.demo.DemoRestApplication`

The class should look like this for MicroProfile implementations (Java EE based ones)

```
@ApplicationPath("/data")
public class DemoRestApplication extends Application {

}
```

Within these Jakarta EE runtimes, the application is scanned for JAX-RS resources and so there is no need to define the individual JAX-RS resources. The only thing you need to define is the part of the URL on which the JAX-RS resources are exposed, `data` in this case using the `@ApplicationPath` annotation.

On the purely JAX-RS implementations, you can see the following class definition.

```
@ApplicationPath("/data")
public class DemoRestApplication extends Application {

    @Override
    public Set<Class<?>> getClasses() {

        Set<Class<?>> classes = new HashSet<>();

        // microprofile jwt auth filters
        classes.add(JWTAuthorizationFilter.class);
        classes.add(JWTRolesAllowedDynamicFeature.class);

        // resources
        classes.add(HelloController.class);
        classes.add(ConfigTestController.class);
        classes.add(ResilienceController.class);
        classes.add(MetricController.class);
        classes.add(ProtectedController.class);
        return classes;
    }

}
```

Here we need to define all JAX-RS resource so that they are found by the implementation and functionality can be used.

JAX-RS resources

After the configuration, we can start creating the REST endpoints. We do this by annotating a POJO class with the `javax.ws.rs.Path` annotation. By doing this, it marks the class as a REST endpoint and defines the URL for the endpoint.

- Open the generated `service-a` project with the IDE of your choice (to make it easier)
- Go to the Java class `HelloController`, for example `com.example.demo.HelloController`

```
@Path("/hello")
@Singleton
public class HelloController {

    @GET
    public String sayHello() {
        return "Hello World";
    }
}
```

Besides the path annotation, which makes the endpoint available at the URL `<host>/<root>/data/hello`, you see also the `@Singleton` annotation. This `javax.inject.Singleton` CDI annotation defines that all requests are handled by a single instance. By default, when no annotation is given, a new instance is created for each new request. But since REST calls are meant to be stateless, there is no need to have a new instance of the class for each request. We will see later on that for most MicroProfile features a so-called CDI Scope defining annotation is required in order to work properly. After all, MicroProfile is built on top of CDI and is using CDI features extensively.

On the method, we see the `javax.ws.rs.GET` annotation which makes that any GET operation on the URL defined by the `@Path` will be handled by this method. In this case, it just returns the (HTML) text `Hello World`.

Dynamic Resources

In this section, we will add new functionality to our demo project which was generated by the MicroProfile Starter application. We will make the above Hello-world style application a bit more interactive and will pass it the name of the person we want to greet.

So when we use the URL `<host>/<root>/data/greet/Rudy`, we want to have the response `Hello Rudy`.

- Open the generated `service-a` project with the IDE of your choice.
- Create a new Java class called `GreetingResource`.
- Add the `@javax.ws.rs.Path("greet")` and the `@javax.inject.Singleton` at the class definition.
- Create the method which will implement the 'business logic'. In this case a simple greeter method.

```
public String sayHi(String name) {
    return "Hello " + name;
}
```

- Let us convert this method to a REST endpoint by adding the `@javax.ws.rs.GET` and `@javax.ws.rs.Path("{name}")` to the method. The `@Path` here defines the placeholder within the URL to retrieve our parameter. When we put something within curly brackets in the `@Path` it is not considered a fixed value but denotes a variable, a placeholder which will be determined for each request.

- Map the placeholder of the URL to the method parameter by annotating the method parameter with `@javax.ws.rs.PathParam("name")`. The value of the placeholder defined within the `@Path` and value used in `@PathParam` must match of course.

The complete Java code looks like this:

```
@Path("/greet")
@Singleton
public class GreetingResource {

    @GET
    @Path("{name}")
    public String sayHi(@PathParam("name") String name) {
        return "Hello "+name;
    }
}
```

Remark: For the JAX-RS based implementations, we need to add this `GreetingResource` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

You can now run the updated application and see if the URL works as intended.

JSON support

The support for consuming and producing JSON is a required feature for all implementations of MicroProfile. This is already by default when the implementation is based on Jakarta EE. As a developer, you don't need to configure anything, just indicating that a certain endpoint uses the JSON (de)serialization.

In this example, we will create an endpoint that accepts some JSON payload through a POST and returns some JSON data as a result.

- Open the generated `service-a` project with the IDE of your choice.
- Create a new Java class called `InputData`. It will hold the data received through the endpoint. It is just a POJO with 2 properties, `name` and `age`.

```
public class InputData {

    private String name;
    private int age;
    // getters and setter omitted
```

- Create a new Java class called `OutputData`. It will hold the data sent by the endpoint. It is just a POJO with 2 properties, `name` and `year`.

```
public class OutputData {

    private String name;
    private int year;
    // getters and setter omitted
```

- Create a new Java class called `PersonResource`.
- Add the `@javax.ws.rs.Path("person")` and the `@javax.inject.Singleton` at the class definition.
- Create the method which will implement the 'business logic'. Here it will calculate the year of birth based on the age.

```
public OutputData calculateYear(InputData inputData) {
    OutputData result = new OutputData();
    result.setName(inputData.getName());
    LocalDateTime year = LocalDateTime.now().minusYears(inputData.getAge());
    result.setYear(year.getYear());

    return result;
}
```

- Convert this method to a REST endpoint by adding the following annotations

```
@javax.ws.rs.POST
@javax.ws.rs.Consumes(javax.ws.rs.core.MediaType.APPLICATION_JSON)
@javax.ws.rs.Produces(javax.ws.rs.core.MediaType.APPLICATION_JSON)
```

- We now define the Http Method as POST, and specify that the body and result need to be converted to/from JSON.

The complete Java code for `PersonResource` looks like this:

```
@Path("/person")
@Singleton
public class PersonResource {

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public OutputData calculateYear(InputData inputData) {
        OutputData result = new OutputData();
        result.setName(inputData.getName());
        LocalDateTime year = LocalDateTime.now().minusYears(inputData.getAge());
        result.setYear(year.getYear());

        return result;
    }
}
```

Remark: For the JAX-RS based implementations, we need to add this `PersonResource` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

You can test this endpoint with the following CURL command. use the corresponding functionality of a REST client tool like PostMan.

```
curl  -H "Content-Type: application/json" --data '{"name":"Rudy","age":48}' -X POST
http://localhost:8080/data/person
```

Validating input

In the above example when we calculate the year, we do not perform any check on the input data. In this example, we will fix this as data validation is an important part of your application.

With JAX-RS, the HTTP status of the response tells us something about the outcome of the request. Until now, we have just returned some value (a String, a POJO transformed to JSON, etc …) from the method which was interpreted by the system as a success and thus status 200 is used. If we want to define the HTTP Status our-self, we have to return an instance of `javax.ws.rs.core.Response`. In this example, we will validate if the `age` value is a valid value and return an HTTP Error Status in case it is not.

- Open the generated `service-a` project with the IDE of your choice.
- Copy the code within the `PersonResource` to a new Java Class `ValidatingPersonResource`.
- Change the `@Path` value to `person2` since we need a different URL than the original endpoint.
- Change the return value of the method from `OutputData` to `javax.ws.rs.core.Response`
- Add the following check at the beginning of the method.

```
if (inputData.getAge() < 1) {
    return Response.status(PRECONDITION_FAILED).build();
}
```

- The 'Precondition Failed' corresponds with HTTP Status 412 and is used to indicate that the requirements for the input data aren't met. In this case, an age value of 0 or lower is not valid.
- Change the return for the method to type Response as follow:

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

- We now specifically define the HTTP status of the response (ok which means 200) and define the payload of the response with the `entity()` method.

The method should look now like this

```
public Response calculateYear(InputData inputData) {
    if (inputData.getAge() < 1) {
        return Response.status(PRECONDITION_FAILED).build();
    }
    OutputData result = new OutputData();
    result.setName(inputData.getName());
    LocalDateTime year = LocalDateTime.now().minusYears(inputData.getAge());
    result.setYear(year.getYear());

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

Remark: For the JAX-RS based implementations, we need to add this `ValidatingPersonResource` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

After deploying the updated version of our application we can test it out with a similar CURL command. Do not forget to change the URL and to play with the value of the age within the JSON data we send.

```
curl  -H "Content-Type: application/json" --data '{"name":"Rudy","age":48}' -X POST
http://localhost:8080/data/person2
curl  -H "Content-Type: application/json" --data '{"name":"Rudy","age":-1}' -X POST
http://localhost:8080/data/person2
```

CDI

With CDI (Contexts and Dependency Injection), you can easily get hold of a required dependency for your code like a service or utility class.

If you know CDI already, you can skip this section and go to the next topic. The rest of this chapter is only a short introduction. Much more can be found on the internet and it will all apply to MicroProfile as CDI is an integral part of MicroProfile.

When relying on CDI, there is no need to instantiate the object, called a CDI bean, yourself. This is done by the container but it does much more than just performing the creation. It also makes sure that you receive a copy of the object suited for your situation based on the scope you specified. It wraps the object in a proxy and the functionality which is specified by the container and/or the developer (applying interceptors and alike) and allows for initialization of the object.

Scopes

Within CDI, you can assign your component a specific scope. Classic scopes are `RequestScoped`, `SessionScoped` and `ApplicationScoped`. This scope determines if a new instance needs to be created or an existing one can be reused. When you define the Class as `RequestScoped` then you receive a different instance for each request. On the other hand, there is only one instance created and used in case you use the `ApplicationScoped`. When there are no users and request specific data, this one is, of course, a memory-efficient way.

Within MicroProfile based applications, the `RequestScoped` and `ApplicationScoped` are the 2 most important scopes. The first one can be used when you store some request-specific information. The latter will be used for service or utility like stateless classes.

Let us update the Dynamic Resources example of the JAX-RS section. In case you skipped, have a look, at Dynamic Resources

- Open the generated `service-a` project with the IDE of your choice.
- Create a new Java class called `GreetingHelper`.
- Define the CDI scope as `javax.enterprise.context.ApplicationScoped` since is it a stateless helper method.
- Define the 'business' method, here it returns a greeting for a name.

```
  public String defineGreeting(String name) {
      return "Hello " + name;
  }
```

- 'Inject' the `GreetingHelper` within the `GreetingController` we have created in the JAX-RS example.

```
@Inject
private GreetingHelper greetingHelper;
```

- The container will provide a suitable object for this dependency to our controller. In this example, it is just the instance of the *GreetingHelper* the container has prepared for us.
- Use the *GreetingHelper* within the `sayHi` method. It is a good practice to keep the 'business logic' separated (the greeting logic) from the code responsible for the interfacing with the client (the

JAX-RS code in our case) to increase the reusability later on if needed.

beans.xml

The `beans.xml` file can be used to configure the CDI functionality within your application. You can use it to activate alternative bean definitions and interceptors. They are not covered in this tutorial, so have a look at some internet resource for examples on this.

The `beans.xml` can also be used to indicate what Java classes must be turned into a CDI bean. You can have the option `ALL` that creates a CDI bean definition for all Java classes. That is the case for the generated demo code although this is not the best practice. It is required for some examples on certain runtimes to work correctly.

The preferred mode for the *discovery mode* is the value `ANNOTATED`. It gives you as developer the full control of what is handled by the CDI engine and is optimal from a performance perspective. It only creates CDI bean metadata for those Java classes that have a CDI scope annotation (the correct definition is any java class that has a bean defining annotation) and thus less memory is used to store this information.

- Open the generated `service-a` project with the IDE of your choice.
- Change the ``bean-discovery-mode`property value within the `beans.xml`` file.

```
beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="annotated">
```

Initialization

When a CDI bean is created by the container, it gets initialized with all other dependencies. But as a developer, you have the chance to perform some specific initialization too. There is also the option to execute a method when the CDI bean is removed from the container and made available for Garbage Collection but this is less used.

The method annotated with the `javax.annotation.PostConstruct` annotation gets executed after the bean has received all the other dependencies but just before it is placed into service within the container. The developer can use it for any programmatic preparation of the CDI bean and can use at that point already all the *injected* references.

As an example, we will continue on the previous example and will initialize the *Hello* greeting value in a variable.

- Open the generated `service-a` project with the IDE of your choice.
- Create an instance variable in the class GreetingHelper to hold the greeting value

```
private String greetingMsg;
```

- Create the initialization method where we set the instance variable.

```
public void init() {
   greetingMsg = "Hello from helper %s";
}
```

- Annotate the `init()` method with `javax.annotation.PostConstruct` so that the method is executed when the CDI bean is created.
- Use the `greetingMsg` variable within the `defineGreeting()` method.

Factory method

Besides the initialization method described in the previous section, there are situations where you need to perform a more complex setup of the CDI bean. For those cases, you can use the factory method pattern to create your CDI bean.

In that scenario, you instantiate the object yourself and perform any necessary action. Other dependencies aren't injected automatically into your instance, but you can programmatically access the CDI container and requests beans from it.

A method can be marked as creating a CDI bean with the `javax.enterprise.inject.Produces` annotation. (Do not confuse this Produces annotation with the one from JAX-RS which defines the format of the response)

In this example, we will transform the `GreetingHelper` example we created above to use the Factory method pattern. But of course, it is not a very good example of this pattern as it was already working fine as it was.

- Open the project with the IDE of your choice.
- Remove the `ApplicationScoped` and `PostConstruct` annotations from the class `GreetingHelper`.
- Create a new Java class called `GreetingHelperProducer`. The name can be anything and has no specific requirements but it is always a good practice to give it a name that clearly indicates the functionality.
- Annotate the class with the `javax.enterprise.context.ApplicationScoped`.
- Create a method that returns an instance of the `GreetingHelper` object.

```
public GreetingHelper createHelper() {
    GreetingHelper result = new GreetingHelper();
    result.init();
    return result;
}
```

- Annotate this method with `javax.enterprise.inject.Produces`. The `GreetingHelper` instance in our case will have scope *ApplicationScope* since it is produced by a CDI bean of that scope. You can create CDi beans with a specific scope by using a scope on the method.
- When a bean now needs to have an instance of the `GreetingHelper` class, it will execute the `createHelper()` method. In this case, the method will only be executed once since there is only one bean of type *ApplicationScoped*.

Events

Another very nice feature of CDI is the ability to use events between a producer and consumer. In contrast to other similar systems, there is no need to register a listener to the consumer. The system is completely loosely couples without any links between the parts.

In this example, we will update the project so that a CDI event is sent out when we access the dynamic REST endpoint.

- Open the generated `service-a` project with the IDE of your choice.
- Create the class `SomeEvent which will contain the payload of our event but is also used to characterize our specific event.
- Create an instance variable `name` to hold the String value used.
- Create a constructor to set this parameter and a getter to retrieve the value.

```
private String name;

public SomeEvent(String name) {
    this.name = name;
}

public String getName() {
    return name;
}
```

- Create a CDI bean called `UsageCollector` which will capture all the CDI events which are created.
- Define the CDI bean as `@ApplicationScoped`
- Create a method which will be called as the listener method. The `@Obserserves` annotation on the parameter determines for which CDI events this method will be used as a listener.

```
public void logUsage(@Observes SomeEvent someEvent) {
    System.out.println("Endpoint called with name : " + someEvent.getName());
}
```

- Within the `GreetingResource` inject a producer which can fire events of type `SomeEvent`

```
@Inject
private Event<SomeEvent> someEvent;
```

- When the JAX-RS endpoint is called, fire the CDI event with the *name* as payload. Add the following line within the `sayHi` method.

```
    someEvent.fire(new SomeEvent(name));
```

Deploy the updated application and let us test it out. Within your browser, use the following URL `<host>/<root>/data/hello/Rudy` to access the endpoint which will also emit the CDI event. The message should appear in the log.

JSONP

JSONP stands for JSON Processing or parse, generate, transform and query JSON. Most developers use it to generate some JSON without the need to have a Java Object first. You can define the name-value pairs which are converted to the correct syntax.

Using JSONP to parse is less used as having the user data posted to your endpoint in a Java object is most of the time much more convenient (and type-safe)

Generate JSON

In this example, we will recreate the scenario which calculates the year of birth we handled when looking at JAX-RS dynamic use case.

- Open the generated `service-a` project with the IDE of your choice.
- Create a new Java class called `PersonControllerJSONP`.
- Add the `@javax.ws.rs.Path("personJSONP")` and the `@javax.inject.Singleton` at the class definition.
- Create the method which will implement the 'business logic'. Here it will calculate the year of birth based on age.

```
public String calculateYear(InputData inputData) {
    LocalDateTime year = LocalDateTime.now().minusYears(inputData.getAge());

    JsonObject json = Json.createObjectBuilder()
        .add("name", inputData.getName())
        .add("year", year.getYear()).build();

    return json.toString();
}
```

- With the help of a builder, retrieved by `Json.createObjectBuilder()`, you can create the JSON by adding name/value pairs. More complex structures are also possible of course.

- Convert this method to a REST endpoint by adding the following annotations

```
@javax.ws.rs.POST
@javax.ws.rs.Consumes(javax.ws.rs.core.MediaType.APPLICATION_JSON)
@javax.ws.rs.Produces(javax.ws.rs.core.MediaType.APPLICATION_JSON)
```

- We now define the Http Method as POST, and specify that the body and result need to be converted to/from JSON.

Remark: For the JAX-RS based implementations, we need to add this `PersonControllerJSONP` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

You can test this endpoint with the following CURL command. use the corresponding functionality of a REST client tool like PostMan

```
curl  -H "Content-Type: application/json" --data '{"name":"Rudy","age":48}' -X POST
http://localhost:8080/data/personJSONP
```

MicroProfile Config

With the MicroProfile Config specification, your application can retrieve some configuration values from various sources. It allows you to create an immutable application and read the values which are different for each environment (like test, acceptance, and production).

The value is searched in different sources which can be configured or even created by the developer. Each source has a priority and when a configuration value is found in a source, that value is taken. This allows having a configuration system where values can easily be overridden for specific cases or environments. And of course, the conversion between String values and Java types are also covered by this specification.

Get configuration value

The most straightforward way of getting a configuration value is by using the CDI injection mechanism. The MicroProfile Config has defined the `org.eclipse.microprofile.config.inject.ConfigProperty` CDI qualifier to identify which value we want.

This is important because the CDI container would not know which String we are referring to when we ask it to inject a 'String'. There are some many String`s at runtime but probably none of them is a CDI bean. With the CDI qualifier (the *ConfigProperty* annotation), we can specifically indicate which String we want from the Configuration value.

The MicroProfile Starter created an example of such an injection of the configuration value. Let us have a look, at it.

- Open the generated `service-a` project with the IDE of your choice (to make it easier)
- Have a look at the class `ConfigTestController`
- You will see the injection of the configuration value. With the `@ConfigProperty` we indicate the key of the config value we are interested in.

```
@Inject
@ConfigProperty(name = "injected.value")
private String injectedValue;
```

- The value for this configuration parameter is defined within the `microprofile-config.properties` file which is located within the `/resources/META-INF` directory.

```
injected.value=Injected value
```

- The injected value is used in the endpoint defined by the method `getInjectedConfigValue()`.

You can see the injection of the parameter in action by using the following URL in your browser `<host>/<root>/data/config/injected`

Using variables

It is now also possible to use variables within configuration values. This allows you to avoid some duplication when you define parameters. A typical example is the hostname of an external system like as you can define with the configuration of the MicroProfile rest Client we will see later on.

Let us change the example that uses the MicroProfile Configuration value through injection.

- Open the generated `service-a` project with the IDE of your choice (to make it easier)
- Edit the configuration value within the `microprofile-config.properties` file.

```
injected.value=Injected value; ${otherkey}
```

After building the application and running it with an additional parameter `-Dotherkey=variables` on the command line, you can see that the outcome on the page in the browser for the URL `<host>/<root>/data/config/injected` contains now the resolved key we specified on the command line also.

Programmatic getting values

Another alternative for getting a configuration value is the programmatic way using the Config API. For the moment there is no real benefit unless the class can’t be turned into a CDI bean, from using the API to retrieve configuration values but in a future release, this API will be extended to contain more features. With the API you can have a look at which Configuration sources are defined and get a list of all configuration parameters which are defined. Of course, the Config API can also be used outside the CDI scope.

The MicroProfile Starter created also an example of programmatic access. Let us have a look, at it.

- Open the generated `service-a` project with the IDE of your choice (to make it easier)
- Have a look at the `getLookupConfigValue()` method in the class `ConfigTestController`.

```
Config config = ConfigProvider.getConfig();
String value = config.getValue("value", String.class);
```

- The `Config` instance is retrieved from the Configuration provider and then in the next statement the value for the parameter is retrieved.

ConfigSources

As already mentioned in the introduction, various sources are consulted to find the value of a configuration key.

In this example, we build a simple rest endpoint and play a bit more with the various configuration sources.

- Open the generated `service-a` project with the IDE of your choice.
- Add a new JAX-RS resource, called `ConfigSourcesController` and set the URL with the `@Path("/config-sources")`
- Specify also the CDI scope `@ApplicationScoped` so that we can inject the configuration value.
- Inject a configuration value, but we specify also a default value. When no value is defined within the configuration sources, we do not get an exception but this default value is taken.

```
@Inject
@ConfigProperty(name = "config.key", defaultValue = "From Code")
private String configValue;
```

- Define the GET endpoint to return the configuration value so that we can see what the application received as a value.

```
@GET
public String getInjectedConfigValue() {
    return "Config value : " + configValue;
}
```

Remark: For the JAX-RS based implementations, we need to add this `ConfigSourcesController` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

Build the application with maven (`mvn clean package`) and start it up. The exact command depends on the MicroProfile implementation you have chosen and this you can find in the *readme.md* file which is also available in the project.

We first start it up with no additional parameters

```
java -jar target/demo-microbundle.jar
```

If we now open our browser with the following URL `<host>/<root>/data/config-sources` we get the default value we have defined in the code.

But we can start the application with a System parameter defining an alternative value for the configuration value.

```
java -Dconfig.key=System -jar target/demo-microbundle.jar
```

If we refresh our browser when the application is started, we see that the endpoint returns the value we have specified on the command line.

Another source that is supported by default are the environment variables. In that case, a special rule is applied as environment variables can't contain a `.` (dot), for example. These characters are converted to an _ before the key value is looked up. In the above example, `config_key` env variable is used when the application request the value if `config.key`.

Optional configuration

Besides the fact that a configuration parameter always has a value, there are also many use cases when the value can be optional. Most of those cases are encountered when you create some kind of library or reusable code. In those situations, you can easily have the case that for some applications you have a value but for others, it should be blank.

When you inject the value as we have seen in the previous section, or when you retrieve it programmatically with `config.getValue()` you receive an exception when the parameter key isn't defined in any of the config resources.

However, you also have the possibility to request for an optional parameter value with MicroProfile Config. Besides the `defaultValue` attribute we have used in the previous section, another option is the use of `Optional` from Java.

- Open the generated `service-a` project with the IDE of your choice.
- Create a class named `OptionalValueController` and define it as a JAX-RS resource by annotating it with `@Path("/config/optional")` which also define the URL on which it will be available.
- Define the JAX-RS resource as a CDI bean by annotating it with `@ApplicationScoped`.
- Define an instance variable of type `Optional<String>`
- Add the annotations to make it a configuration parameter which will be injected.

```
@Inject
@ConfigProperty(name = "optional.value")
private Optional<String> optionalValue;
```

- Create a method which will return the instance variable value or some default value.

```
public String getInjectedConfigValue() {
    return "Optional Config value " + optionalValue.orElse("No Value Defined");
}
```

- Mark the method as linked to the GET operation by adding the `@GET` annotation.

Remark: For the JAX-RS based implementations, we need to add this `OptionalValueController` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

When you now deploy the application and request the following URL in your browser `<host>/<root>/data/config/optional` you receive the default value defined within the `orElse()` method.

In the case you define a value within the file `microprofile-config.properties`, you will get the configured value.

```
optional.value=A real value
```

Conversions

Values defined as configuration values are all a set of characters and thus a String value. MicroProfile Config allows for automatic conversion to the correct Java type.

You probably saw already an example of this when we introduced MicroProfile Config in the example generated by the MicroProfile Starter application. If you missed it, let go back to that example.

- Open the generated `service-a` project with the IDE of your choice (to make it easier)
- Have a look at the class `ConfigTestController`
- You will see the injection of the configuration value. With the `@ConfigProperty` we indicate the key of the config value we are interested in.

```
@Inject
@ConfigProperty(name = "injected.age")
private Integer age;
```

- In the above case, the value will be converted to an `Integer` and this conversion will result in an exception when this conversion fails of course.

Conversion to the basic Java types, like boolean, int, float, and Class are supported by default.

There is also the principle of the implicit converter which tries to converter the String value. These are the rules.

- The target type `T` has a `public static T of(String)` method, or
- The target type `T` has a `public static T valueOf(String)` method, or
- The target type `T` has a `public Constructor with a String parameter`, or
- The target type `T` has a `public static T parse(CharSequence)` method

You can read more about converters on this page from the MicroProfile Configuration Spec

Parameter and collection type

MicroProfile Config not only supports single values but also supports collections of a type as a parameter value. No specific configuration or programming is required to activate this support. It can be combined with the default, implicit, and custom converters.

In this example, we will see how we can retrieve a parameter that consists of a List of String values.

- Open the generated `service-a` project with the IDE of your choice.
- Create a class named `ConverterController` and define it as a JAX-RS resource by annotating it with `@Path("/config/converter")` which also defines the URL on which it will be available.
- Define the JAX-RS resource as a CDI bean by annotating it with `@ApplicationScoped`.
- Define an instance variable of type `List<String>` and define the `@ConfigProperty` to refer to a configuration key.

```
@Inject
@ConfigProperty(name = "myPets")
private List<String> pets;
```

- Create the method which will return the value of the String list. We will use the Java 8 joining method for this purpose.

```
public String getConfigValue() {
    String myPets = String.join(" - ", pets);
    return String.format("Pets : %s", myPets);
}
```

- Mark the method as linked to the GET operation by adding the `@GET` annotation.
- Define a value for the configuration within the file `microprofile-config.properties`. The `,` is used to separate the different items. When the value itself contains a `,` you need to escape it as demonstrated in the example.

```
myPets=dog,cat,dog\\,cat
```

Remark: For the JAX-RS based implementations, we need to add this `ConverterController` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

Deploy the application and open the browser and point to the URL `<host>/<root>/data/config/converter` and see how the configuration parameter value is corrected converted to a collection type.

Custom converters

When the default converters (to instances like int, boolean and float) or the implicit converters are not doing the job you want, you can always create your own converter.

In this example, we create a converter for our RGB class that holds colour values.

- Open the generated `service-a` project with the IDE of your choice.
- Create a `RGB` class that represent the class for our specific configuration parameter.
- Create the 3 instance variables to hold the colour values, a constructor to initialise the class, the getters and a `toString()` implementation so that we can easily print out the contents of the instance.

```
public class RGB {
    private int r;
    private int g;
    private int b;

    public RGB(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    // getters omitted for brevity

    @Override
    public String toString() {
        return "RGB{" +
                "r=" + r +
                ", g=" + g +
                ", b=" + b +
                '}';
    }
}
```

- Create a `RGBConverter` class which will convert the String value to an instance of the `RGB` class we have defined.

- The class needs to implement the `org.eclipse.microprofile.config.spi.Converter` interface

```
implements Converter<RGB>
```

- Create the required `convert()` method where we implement a simplified conversion (without error checking) --- public RGB convert(String s) { String[] parts = s.split(","); return new RGB(Integer.parseInt(parts[0]), Integer.parseInt(parts[1]), Integer.parseInt(parts[2])); } ---
- The converter needs to be registered through the service Loader mechanism of Java. So start by creating the file `src/main/resources/META-INF/services/org.eclipse.microprofile.config.spi.Converter`
- Add the fully qualified name of our converter into this file.

```
com.example.demo.config.RGBConverter
```

- Open the `ConverterController` we created in the previous example.
- Add an instance variable of `RGB` and define also the link with the configuration key. There is no need to specify our custom converter here since it will be selected based on the type of the instance variable.

```
@Inject
@ConfigProperty(name = "color")
private RGB rgbValue;
```

- Update the return value of the method `getConfigValue()` to include the value of the `RGB` instance.

```
return String.format("Pets : %s, Color :  %s", myPets, rgbValue);
```

- Add a configuration value for the key `color` within the `microprofile-config.properties` file.

```
color=100,200,128
```

Update the running application and have a look at the output for the endpoint `<host>/<root>/data/config/converter` in your browser.

MicroProfile Rest Client

The JAX-RS specification (Jakarta EE) has a nice client-side builder for creating and calling REST endpoints. An example of calling such an endpoint that retrieves the `Employee` data could look like this.

```
private Client client = ClientBuilder.newClient();

public Employee getJsonEmployee(int id) {
    return client
      .target(REST_URI)
      .path(String.valueOf(id))
      .request(MediaType.APPLICATION_JSON)
      .get(Employee.class);
}
```

However, this approach is still closely related to how the REST call is building up and executed. It assembles the URL, defines the expected format of the response, and then performs the GET operation.

With the MicroProfile Rest Client specification, the idea is that we work in a much more type-safe manner and that the called endpoints are represented by interfaces. There would almost be no difference between calling a method within the same JVM and calling a REST endpoint on a remote server. However, one should not forget that REST is not made for remote procedure calls but is data-oriented.

The idea of MicroProfile Rest Client is that you indicate some JAX-RS concepts on the interface and that a JAX-RS client will be generated automatically based on this information to call the endpoint. The example will make this concept much clearer.

CDI Based Type-Safe Rest Client

The starter has generated an example for the Rest Client. Until now, we only have looked at the `service-a` project that was in the download.

The downloaded zip file also contains a `service-b` directory that contains a few endpoints that are called from the `service-a` application. It demonstrates the typical usage of a microservices architecture solution.

Let us have a look at the example of the Rest Client.

- Open the generated `service-b` project with the IDE of your choice.
- Have a look at the class `ServiceController`. It defines an endpoint that takes a URL parameter and returns this to the caller.
  This doesn’t contain any MicroProfile specific code or annotations. This means you will be able to call any kind of endpoint, regardless of the technology it is created with.
- Follow the instructions in the readme document of the service-b directory and start up the runtime. Once it is up, you can test it out by calling a similar URL in the browser

```
http://localhost:8180/data/client/service/Payara
```

- Open the generated `service-a` project with the IDE of your choice.

- Look for the interface `Service`

```
@RegisterRestClient
@ApplicationScoped
public interface Service {

    @GET
    @Path("/{parameter}")
    String doSomething(@PathParam("parameter") String parameter);

}
```

- The `@RegisterRestClient` annotation instructs the runtime to create a Rest Client for an endpoint described in this interface.
- The Method signature and the annotations within the interface need to match with the endpoint of the service we like to call. In this example, it means we define de @Path annotation with the path parameter and define the Path Parameter as a method argument. We also indicate the call must be performed through a GET method.
  As you can see, the method signature and annotations on the method within the `Service` interface and the one in the `ServiceController` class are identical. In fact, you can define it as an interface and let the annotations inherit if you like that.
- The generated Rest Client is used within the class `ClientController` in the `service-a` project.

```
    @Inject
    @RestClient
    private Service service;

    @GET
    @Path("/test/{parameter}")
    public String onClientSide(@PathParam("parameter") String parameter) {
        return service.doSomething(parameter);
    }
```

- The generated Rest Client is a CDI bean and thus can be injected into another bean. We only need to add a qualifier `@RestClient` to indicate to the runtime we want to the generated code here.
- Calling the remote endpoint looks like a regular call of an in JVM call. Based on the statement `service.doSomething(parameter)` it appears a regular call but in fact, a call to the remote endpoint is generated, executed and necessary conversions to and from JSON are performed.
- The last piece of the puzzle is that we define the actual URL where the endpoint can be called. Add the configuration for the endpoint as a MicroProfile configuration value. Add a similar entry like this in the `microprofile-config.properties` file.

```
com.example.demo.client.Service/mp-
rest/url=http://localhost:8180/microprofile/data/client/service
```

- In the above config key, the first part if the fully qualified class name of the interface describing the remote service. The config value is dependent on the URL we have tested for the server side part in the first half of this example.

You can build and run the `service-a` project and test it out in the browser with the following URL `<host>/<root>/data/client/test/rudy`. The return value you will see in the browser is produced by the `service-b` code.

Using ClientBuilder

Just as you have the possibility of using CDI and a Java API within MicroProfile Config, you also have both options with MicroProfile Rest Client.

The general principles are the same. You model the endpoint you want to call in an interface, but this time you do not specify any CDI annotations, and with the `clientBuilder` you request the generated class capable of calling the endpoint.

In this example, we perform exactly the same functionality as in the previous section which was completely based on CDI. So we will reuse the server side part and the interface `Service` from the previous example.

- Open the generated `service-a` project with the IDE of your choice.
- Create a JAX-RS resource so that we can call our code. The class can be called `ClientBuilderController`.
- Annotate it with `javax.ws.rs.Path("/clientBuilder")` and `javax.enterprise.context.ApplicationScoped`.
- Inject the URI on which the service can be found. We now just need the URL on which the endpoint can be found and do not need the Rest Client config anymore which we had in the previous example.

```
@Inject
@ConfigProperty(name = "serviceURI")
private URI serviceURI;
```

- Create the method which will be exposed as the endpoint for this JAX-RS resource. It will call the remote endpoint with a generated Rest Client.

```
public String withBuilder(String parameter) {
    // Implement
}
* The Rest Client can be build using the `RestClientBuilder` and then we just call the
method as before.
```

Service serviceFromBuilder = RestClientBuilder.newBuilder() .baseUri(serviceURI) .build(Service.class); return serviceFromBuilder.doSomething(parameter);

```
* The method needs to be linked with the JAX-RS resource path. This is also described
a few times in the above examples, so here is the end result.
----)
@GET
@Path("/{parameter}")
public String withBuilder(@PathParam("parameter") String parameter) {
```

- Add the configuration for the URL as a MicroProfile configuration value.

```
serviceURI=http://localhost:8180/microprofile/data/client/service
```

- The config value is dependent on the URL we have tested for the server side part of the previous example.

Remark: For the JAX-RS based implementations, we need to add this `ClientBuilderController` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

Deploy the application and test it out in the browser with the following URL `<host>/<root>/data/client/builder/testing`.

Adding functionality

The JAX-RS client has the possibility the add custom functionality. You can define `ClientRequestFilter` s to modify the outgoing request, `ClientResponseFilter` s to handle the response, and `MessageBodyWriter` s and `MessageBodyReader` s to define how the request and response data are written and read.

This kind of functionality can also be added to the MicroProfile Rest Client using both the CDI and the Java API way.

In this example, we will create a *ClientRequestFilter* to view the final URL which is called by the Rest Client which can be handy for debugging purposes. It is built on top of the 2 previous examples, the CDI-based and the Java API-based usage of the MicroProfile Rest Client.

- Open the generated `service-a` project with the IDE of your choice.
- Create the class `DebugClientRequestFilter` which implements the `ClientRequestFilter` interface.
- Implement the `filter` method, here we just output the final URL.

```
@Override
public void filter(ClientRequestContext requestContext) throws IOException {
    System.out.println("Calling URL:" + requestContext.getUri());
}
```

- Add the `@RegisterProvider` annotation on the `Service` interface we have created for the CDI-based example earlier in this chapter. It makes sure that our *ClientRequestFilter* is registered.

```
@RegisterProvider(DebugClientRequestFilter.class)
```

No other changes are required to use this custom functionality.

In this same example, we will also update the Java API-based example.

- Open the `ClientBuilderController` class we have created earlier.
- Add the following line in the builder usage so that our `DebugClientRequestFilter` will be used.

```
    .baseUri(serviceURI)
    .register(DebugClientRequestFilter.class)  // This is the line we need to add
    .build(Service.class);
```

Deploy the application and test it out in the browser with the following urls

- using the registration on the CDI based version `<host>/<root>/data/client/test/rudy`
- using the Java API based registration `<host>/<root>/data/clientBuilder/testing`

Calling a Rest endpoint asynchronous

In this last example of the MicroProfile Rest Client, we will asynchronously call the Rest endpoint. There is no specific requirement on the 'server' side part of the example. It can just be a regular endpoint. The client side can continue with other tasks and pick up the result of the endpoint later on.

In this example, we will create a *heavy* service that takes some time to complete. Here, for simplicity's sake, it is just a wait we introduce. The client will call this endpoint and will continue with some other tasks before it 'retrieves' the value of this endpoint.

We need again two parts in this example, a remote endpoint which we call the *'heavy service'* implemented in `service-b` project, and a client within the `service-a` project, the JAX-RS endpoint we will call directly from the browser.

- Open the generated `service-b` project with the IDE of your choice.
- Create a Class called `HeavyCalculationController` which will represent the 'heavy server' endpoint.
- Annotate the class with a `javax.ws.rs.Path` annotation so that it is associated with a certain URL.

```
@Path("/client/heavy")
```

- Create the method which will perform the calculation. In our example, it is just a wait and some logging output so that we can better understand what is happening

```
    public String calculate() {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();  // FIXME Demo code
        }
        System.out.println("Returning result from Heavy service");
        return "The result after some substantial amount of calculation";
    }
```

- Add the `@javax.ws.rs.GET` annotations to the method so that it is turned into a JAX-RS endpoint.

You can see that we did not introduce any specific code on this server-side for the asynchronous handling (although that is also possible).

Remark: For the JAX-RS based implementations, we need to add this `HeavyCalculationController` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

Now we will asynchronously use this endpoint.

- Open the generated `service-a` project with the IDE of your choice.
- Create the Java interface `HeavyService` which will hold the definition of our remote endpoint.
- Indicate it is a CDI bean with a specific scope, `@ApplicationScoped` in our example, and also annotate it with `@org.eclipse.microprofile.rest.client.inject.RegisterRestClient` so that the implementation create a CDI bean out of this interface and makes it available for injection through CDI.
- The method we create for the endpoint has a specific return type, `CompletionStage` to make the usage asynchronous.

```
    @GET
    CompletionStage<String> calculate();
```

- Create a class `ClientAsyncController` which will represent the client side of this example.
- Make it also a JAX-RS resource so that we can call it from the browser. Annotate it with `javax.ws.rs.Path("/client-async")` and `javax.enterprise.context.ApplicationScoped`.
- Inject the Rest Client CDI bean which is generated by the system based on the interface definition we have created above.

```
@Inject
@RestClient
private HeavyService service;
```

- Create the method which will be exposed as the endpoint for this JAX-RS resource.

```
public String useAsync() {

}
```

- The first step is to call the remote endpoint. This returns us now a `CompletionStage` instance. This means that the method returns immediately and we can continue on the client-side and use this `CompletionStage` instance to retrieve the result of the call later on.

```
   CompletionStage<String> stage = heavyService.calculate();
```

- The 'other client-side' activity here is just writing something to the console.
- And then we can retrieve our result from the JAX-RS endpoint

```
String result;
try {
    result = stage.toCompletableFuture().get();
} catch (InterruptedException | ExecutionException e) {
    result = e.getMessage();
}
```

- Add the configuration for the endpoint as a MicroProfile configuration value. Add a similar entry like this in the `microprofile-config.properties` file.

```
com.example.demo.client.HeavyService/mp-
rest/url=http://localhost:8180/microprofile/data/client/heavy
```

- In the above config key, the first part is the fully qualified class name of the interface describing the remote service. The config value is dependent on the URL we have tested for the server-side part in the first half of this example.

Remark: For the JAX-RS based implementations, we need to add this `ClientAsyncController` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

Deploy the application and test it out in the browser with the following URL `<host>/<root>/data/client-async`.

You can even play a bit more with the example and see the interaction. The 'client' can do many things and should not be 'listening' for the response from our heavy service. So when you also add a *sleep* there, you will see the changed order of the log messages.

Fault tolerance

With a micro-services architecture, you are entering the world of distributed computing systems. And thus you will be vulnerable to the distributed computing fallacies. One of the important aspects is that you have to deal with failure. Some services can be temporarily down, there are network issues, etc … . The MicroProfile Fault Tolerance gives you strategies to handle those scenarios where execution isn't going as planned.

BulkHead

With a bulkhead approach, you can avoid that an issue in one part of your system affects your complete application. With the bulkhead, you can limit the number of concurrent requests to a resource in a declarative way.

In this example, we will create a JAX-RS endpoint that takes some time to process (so it allows us to call it multiple times in a concurrent way) and that we make sure that only a certain amount of requests can be processed at the same time. This strategy can protect your endpoint and your system from being overloaded and bring the complete application down.

- Open the generated `service-a` project with the IDE of your choice.
- Create a Class called `SlowController` which will represent the endpoint that takes some time to complete and should not be overloaded.
- Annotate the class with a `javax.ws.rs.Path` annotation so that it is associated with a certain URL.

```
@Path("/bulk/slow")
```

- Define besides the `javax.enterprise.context.ApplicationScoped` annotation also the `org.eclipse.microprofile.faulttolerance.Bulkhead` annotation to define the maximal number of concurrent executions.

```
@Bulkhead(3)
```

- Define the method which will be turned into a JAX-RS endpoint. In this case, it doesn't do any useful but we make sure that it takes more than 500 milliseconds to execute.

```
public String calculate() {
    try {
        Thread.sleep(500);
    } catch (InterruptedException e) {
        e.printStackTrace();  // FIXME Demo code
    }
    return "Slow Result";
}
```

- Annotate the method `calculate()` with the `javax.ws.rs.GET` annotation so that we can call it from an URL.

Remark: For the JAX-RS based implementations, we need to add this `SlowController` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

We can already test this out by deploying the application and pointing our browser to the URL `<host>/<root>/data/bulk/slow`.

In the second part of this example, we create a small test program where we simulate multiple concurrent calls to the Bulkhead protected endpoint.

Since this test program involves a little bit more code than the rest of the examples here, you can find the code For the CallRest class at https://github.com/rdebusscher/microprofile-tutorial/blob/c47d543006b0413ffe70b869e65a1be91c5dc166/solution/service-a/src/main/java/com/example/demo/resilient/bulkhead/BulkHeadTester.java#L34

- Create a Class called `BulkHeadTester` and copy the `CallRest` class as an inner class into it.
- Make sure that you update the URL which will be called to the correct value according to the implementation you are using.
- Create a `main()` method which is our test program.
- Define a variable that will hold the number of requests and a for loop that calls the endpoint.

```
int number = 4;

for (int i = 0; i < number; i++) {
    new Thread(new CallRest(i)).start();
}
```

- With the server and our application running, execute the test programming

Since the number of requests is higher than the value we defined in the `Bulkhead` annotation, we receive an exception for some requests and only an actual response for 3 of our requests.

The Bulkhead annotation is a CDI interceptor that keeps track of the number of executions and thus can throw an exception when we have exceeded the maximum limit. When we combine the annotation with `org.eclipse.microprofile.faulttolerance.Asynchronous`, actual threads are used to execute the method and a Thread Pool is then used to limit the concurrent executions. In that case, we have also a waiting queue to our disposal instead of just throwing an exception immediately when the maximum number is exceeded.

Try this out and compare with the solution code if you are unsure if it is correct.

Retry

A retry policy can be a good option in the case we are calling another service, possibly out of our control, which sometimes fails. With the retry we can then have another call in the hope it works this time.

So suppose we have our Service X which calls another service Y. When we receive an error from Service Y, we can propagate the error. By annotating the Service X with `@Retry` we can automatically perform a retry of Service X (and thus also try to call Service Y again)

In this example, we create a JAX-RS endpoint which alternatively fails and succeeds. But by using the annotation, we will see only the successful call because of the retry.

- Open the generated `service-a` project with the IDE of your choice.
- Create a Class called `RetryController` which will contain the JAX-RS endpoint which will fail regularly in our case.
- Annotate the class with a `javax.ws.rs.Path` annotation so that it is associated with a certain URL.

```
@Path("/retry")
```

- The `javax.enterprise.context.ApplicationScoped` annotation serves 2 important purposes now. We need it because of the CDI nature of the MicroProfile Fault Tolerance code but we will keep also a counter which tracks the number of invocations.
- Create the method which will be used as JAX-RS endpoint. The method will return a WebApplicationException when the invocation count is odd or return the invocation count.

```
private int counter = 0;

public String getNextEvenValue() {
    counter++;
    if (counter%2 == 0) {
        return String.valueOf(counter);
    } else {
        throw new WebApplicationException("Only even number allowed");
    }
}
```

- Annotate the method `getNextEvenValue()` with the `javax.ws.rs.GET` annotation so that we can call it from a URL.
- Annotate the class (or method) with `org.eclipse.microprofile.faulttolerance.Retry` so that we retry in case of an exception.

Remark: For the JAX-RS based implementations, we need to add this `RetryController` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

Deploy the application and your browser to the URL `<host>/<root>/data/retry`. You will see that you will receive the value `2` as the result. And that there is no error in the log nor in the browser for that first call to the method which resulted in the `WebApplicationException`.

When you refresh the page, you will receive `4`, then `6` and so on.

Have a look at the members of the `@Retry` annotation. The `maxRetries` limit the number of retries (so that we do no enter in an infinitive loop). We can also define a wait time between retries and define a period after which we give up retrying, even if the maximum number of retries isn't reached yet.

CircuitBreaker

The circuit breaker pattern also deals with JAX-RS endpoints that fail. But it is targeted to another type of problem and provides a different solution. With `@Retry` we want to solve an occasional issue with an endpoint. With circuit breaker, we want to mitigate an outage or problem which can occur at a certain moment in time.

After a few problematic executions, the circuit breaker goes open and no call is going through anymore. After some wait time, some requests are again allowed to the actual endpoint to see if the problem is solved (the half-open state) Based on this 'test' request, the state can become closed (normal functionality) or back to open because the problem is still there.

In this example, we create a variant on the example we used for the Retry annotation. In contrast with what I mentioned, the endpoint will fail in a predictable way so that we can better understand how the functionality works.

- Open the generated `service-a` project with the IDE of your choice.
- Create a Class called `CircuitBreakerController` which will contain the JAX-RS endpoint which will fail predictably in our case.
- Annotate the class with a `javax.ws.rs.Path` annotation so that it is associated with a certain URL.

```
@Path("/breaker")
```

- The `javax.enterprise.context.ApplicationScoped` annotation serves 2 important purposes now. We need it because of the CDI nature of the MicroProfile Fault Tolerance code but we will keep also a counter which tracks the number of invocations.
- Create the method which will be used as JAX-RS endpoint. The method will return a WebApplicationException every third invocation of the method. It also returns the current timestamp to illustrate the waiting period.

```
private int counter = 0;

public String getValue() {
    counter++;
    if (counter % 3 != 0) {
        return String.format("counter : %s - time : %s", counter, new Date());
    } else {
        throw new WebApplicationException("Every 3th request fails");
    }
}
```

- Annotate the method `getValue()` with the `javax.ws.rs.GET` annotation so that we can call it from an URL.
- Annotate the class (or method) with `org.eclipse.microprofile.faulttolerance.CircuitBreaker` to apply that pattern.

```
@CircuitBreaker(requestVolumeThreshold = 6, failureRatio = 0.3)
```

- The `requestVolumeThreshold` indicates how many requests the statistics need to be calculated. When the `failureRatio` is higher than indicated, the circuit breaker goes open (as it will be in our case with a failure ratio of 33%)

Remark: For the JAX-RS based implementations, we need to add this `CircuitBreakerController` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

Deploy the application and your browser to the URL `<host>/<root>/data/breaker`. You will see the following pattern:

Call	**Result**
1	1
2	2
3	Exception
4	4
5	5
6	Exception
7	Indication of open state

That last invocation is interesting. According to our implementation, we know it will succeed. But based on the previous 6 invocations with a failure rate of 33%, the system does not even try to execute it and just returns the Exception.

If we wait more than 5 seconds after the 6th invocation, the circuit breaker is within the half-open state and we will receive a result again (indicating it is the 7th call to the code)

Have also a look at the many members of the annotation and the specification so that you understand all the configuration options it has.

TimeOut

Another variation of a failure in a distributed system is that service execution can sometimes take too much time. Instead of that the service B just fails, it can take maybe 10 or 100 times more time to receive the response.

Our service A should be able to continue in the case when the remote service fails to respond in a timely manner, probably returning an error condition to the caller of Service A.

The MicroProfile Starter has generated an example that uses the `@TimeOut` annotation. It also combines it with the `@FallBack` annotation which will be discussed in the next section.

- Open the generated `service-a` project with the IDE of your choice.
- Have a look at the class `ResilienceController`
- You will see a method that has the `org.eclipse.microprofile.faulttolerance.Timeout` annotation and defines a time out of 500 milliseconds.

```
@Timeout(500)
```

- The method itself has a sleep statement of 700 milliseconds. So there will never be an answer within the configured time period of 500 milliseconds.

```
public String checkTimeout() {
    try {
        Thread.sleep(700L);
    } catch (InterruptedException e) {
        //
    }
    return "Never from normal processing";
}
```

- For this example, just comment out the `@Fallback` annotation.

You can see the timeout in action by using the following URL in your browser `<host>/<root>/data/resilience`

The result of the call is a TimeOutException because the method took too long to respond.

Fallback

In all the above examples we have described regarding the Fault Tolerance specification of MicroProfile, we are missing a recovery or alternative action. There are many situations that we can rely on a fallback mechanism, for example, a cache or return with the indication that no data could be retrieved.

The above annotations can all be combined with the `@Fallback` to provide a more meaningful return than throwing an exception.

The MicroProfile Starter generated an example, see also the Timeout section.

- Open the generated `service-a` project with the IDE of your choice.
- Have a look at the class `ResilienceController`
- The `checkTimeout` method has a `@Fallback` annotation which defines the method that needs to be executed in case of 'problem' (time out in this case)

```
@Fallback(fallbackMethod = "fallback")
```

- The String point to the method which should be executed

```
public String fallback() {
    return "Fallback answer due to timeout";
}
```

You can see the fallback in action (make sure you uncomment the @Fallback if you have commented it in the Timeout example) by using the following URL in your browser `<host>/<root>/data/resilience`

You can see the return value of the `fallback()` method as a result.

Instead of a String pointing to a method, you can also use a class (implementing the `FallbackHandler` interface) which is, of course, a more type-safe solution. Be aware however that it is an unmanaged instance that is executed and thus no CDI injection can be used.

Metrics

Within a distributed environment, as a micro-service architecture is, it is even more important to have some good values for metrics. These kinds of metrics can be an important instrument to determine if you need to scale the number of instances for your particular service for example. If the CPU metric indicates for instance that you have a high load in the last period, another instance could help to reduce the load on an individual instance.

Required metrics

Every compatible MicroProfile implementation is required to provide you with a set of basic metrics which are related to the JVM. The values for these metrics all come from MBean instances of the JVM.

These metrics can be retrieved from an endpoint and are provided in 2 formats. The default format is the Prometheus one which is a very popular time-series database for storing metrics.

Let us have a look at the metrics endpoint output for these required metrics.

- Deploy the MicroProfile starter generated application or the one you have created in the previous sections.
- Open the browser and request the URL `<host>/metrics/base`

The resulting page contains a list of metrics in the following format.

```
# TYPE base:cpu_available_processors gauge
# HELP base:cpu_available_processors Displays the number of processors available to
the JVM. This value may change during a particular invocation of the virtual machine.
base:cpu_available_processors 12
```

The Prometheus format is self-describing to the product and gives information about what can be expected (as a gauge, see further on) and a descriptive text of what the value means.

When you like to see the information in a JSON format, you can execute for example the following curl command

```
curl -H "Accept: application/json" http://localhost:8080/metrics/base
```

Gauge

With the Gauge metric, you can expose any kind of value into the Metrics registry. And every Metric which is registered into that registry can be retrieved from the Metric endpoint we saw in the previous section.

What you put as a value into that metric is up to you. So it is an easy way to expose a value you think is important or that you require to be able to be monitorable. Let us have a look at the MicroProfile Starter generated application as it has an example.

- Open the generated `service-a` project with the IDE of your choice
- Go to the Java class `MetricController`, a simple JAX-RS Endpoint.
- Have a look at the method `getCustomerCount()`, it returns a certain value, here the number of customers.
- It has a @Gauge annotation which indicates that this method will be called when the Metrics endpoints need the value for the metric.

```
@Gauge(name = "counter_gauge", unit = MetricUnits.NONE)
private long getCustomerCount() {
```

- The CDI scope of the JAX-RS resource is very important here and is defined as `@ApplicationScoped`. Since the Metric system will call the method when it requires the value, we need to make sure that we only have one instance which supplies this value. Otherwise, the

Metric system would not know which instance to call to retrieve the value.

The use of a JAX-RS resource here is also no coincidence. The @Gauge annotation will make sure that the metric is registered when this class is 'processed'. For a JAX-RS resource, this is at deployment time. For a regular CDI bean, this will be the time the bean will be instantiated which is not necessarily at deployment time. In that case, the gauge value might not be available from the moment the application is deployed.

Before the metric information appears on the metics page, make sure you call the endpoint `<host>/<root>/data/metric/increment`

The gauge value can be consulted by requesting the `<host>/metrics/application` URL in the browser or if you want the JSON output format you can use curl command for examples

```
curl -H "Accept: application/json" http://localhost:8080/metrics/application
```

Timed

Another type of metric is the execution time for a JAX-RS call. In this case, we do not determine the metric value our-self but it is provided by the implementation. Based on the execution time, all kinds of metrics are available. Let us have a look at the MicroProfile Starter generated application as it has an example.

- Open the generated `service-a` project with the IDE of your choice
- Go to the Java class `MetricController`, a simple JAX-RS Endpoint.
- Have a look at the method `timedRequest()` which has some random wait time before it returns a certain value.
- It has a @Timed annotation so that statistics are available about the execution time

```
@Timed(name = "timed-request")
```

When we deploy the application, we can call the `<host>/<root>/data/metric/timed` endpoint which will record then one execution timing for it. We can consult the `<host>/metrics/application` URL in the browser to have a look at the values for our metric. These values get more interesting of course when we execute our timed endpoint multiple times of course.

The statistics include

- The average number of calls
- The average number of calls during the last 5 and 15 minutes
- Some statistics like, Minimum, Maximum, Average, Standard Deviation, and Percentiles.

Metered

In the case you are only interested in the execution number, how many times is a particular endpoint is called, and not in the timings, you can use the @Metered annotation.

As an example, let us change the timed metrics from the Starter generated project.

- Open the generated `service-a` project with the IDE of your choice
- Go to the Java class `MetricController`
- Replace the `@Timed` with the `@org.eclipse.microprofile.metrics.annotation.Metered` annotation.

Deploy the application, we can call the `<host>/<root>/data/metric/timed` endpoint. Have a look again at the `<host>/metrics/application` endpoint and you see now only the rate values and no longer the statistics like percentiles.

Metric

The MicroProfile Metric specification can also be used to gather some metrics for your own application needs. We have seen already the @Gauge annotation where the developer is responsible for defining the value. But if your application requires a histogram for example, which you want to consult from the metrics page (but you can consult and retrieve the values also programmatically) then the MicroProfile Metrics can also give you an easy solution.

In this example, we create a histogram for the Random Number generator to verify its correctness.

- Open the generated `service-a` project with the IDE of your choice.
- Create a Class called `HistogramController` which will contain the JAX-RS endpoint that generates the histogram.
- Annotate the class with a `javax.ws.rs.Path` annotation so that it is associated with a certain URL.

```
@Path("/histo")
```

- The `javax.enterprise.context.ApplicationScoped` annotation is required to have a correct integration with the MicroProfile Metrics specification. Some implementations may not need this annotation in order to have a correctly working example.
- Inject an instance of a Histogram which we can manipulate our-self.

```
@Inject
@Metric(name = "Random Number Histogram")
private Histogram histo;
```

- Define a method which generates some random numbers and adds it to the histogram.

```
public String calculateRandomNumberHistogram() {
    Random rnd = new Random();
    for (int i = 0; i < 1000; i++) {
        histo.update(rnd.nextInt(100) - 50);
    }
    return "See Metric page for Random Number Histogram";
}
```

- The method generates random integers between 0 and 100 and by subtracting 50, the average and mean value should be 0.
- Add the `@javax.ws.rs.GET` annotations to the method so that it is turned in to a JAX-RS endpoint.

Remark: For the JAX-RS based implementations, we need to add this `HistogramController` class to the list of JAX-RS endpoints within the `<artifact>RestApplication` class.

When you deploy the application, execute the endpoint which generates the histogram `<host>/<root>/data/histo`, you can verify then if the mean and average are indeed O on the metrics page `<host>/metrics/application`.

HealthChecks

Within a micro-service architecture, it is important to know if your micro-service is still healthy. That is still can respond properly when requests are sent. In most cases, this means you need to have more information than just knowing it is up and running. As the developer of the micro-service, you can define some criteria which define if the service is running fine or that it should be discarded and started up again.

The MicroProfile Health Check specification defines such an API which allows you to define these criteria and combines this into a status that can be consulted through an endpoint.

Check implementation

The MicroProfile starter has an example of how you can implement the health Check API and provide a status.

- Open the generated `service-a` project with the IDE of your choice
- Go to the Java class `ServiceLiveHealthCheck`
- The class implements the `org.eclipse.microprofile.health.HealthCheck` interface and is annotated with `org.eclipse.microprofile.health.Liveness`. This is required to be picked up by the implementation and the determination of the Health status (is the application still live and can it continue to receive requests).
- The class implements the `call()` method from the interface.

```
public HealthCheckResponse call() {
```

- The method returns a HealthCheckResponse, in this demo example it just returns a UP status.

```
return
HealthCheckResponse.named(ServiceHealthCheck.class.getSimpleName()).up().build();
```

- Every Health check needs a name (here we used the class name) and a status, UP or DOWN.

Deploy the application and consult the specific endpoint for retrieving the health status of your application `<host>/health/live`. The implementation will aggregate all the checks it finds within the application and calculates an overall status.

If one of the checks returns a status DOWN, the overall status is also DOWN.

Have a look at the MicroProfile Health Check specification for more information on the wire protocol of the response of this endpoint.

OpenAPI

The OpenAPI lets you describe the REST APIs of your application. It specifies which endpoints are available, which operations you can use, what parameters it takes, and describes the response, just to name the most important parts. This description can also be used to generate a client which can call the endpoint. So the OpenAPI description of your application is a valuable tool.

The MicroProfile OpenAPI specification allows you to generate this description easily.

All implementations must generate this documentation by default, and it is accessible from the openapi endpoint. So with an application deployed, visit the URL `<host>/openapi` in your browser.

You receive now the OpenAPI specification for your application.

Customizing

The generated description can be customised in different ways. You can add annotations on the JAX-RS endpoints and supply some information or you can use the `org.eclipse.microprofile.openapi.OASFilter` to do this programmatically.

Have a look at the MicroProfile OpenAPI specification on how you can use the OASFilter. In this example, we will adjust the generated documentation using annotations as the generator created a complete example.

- Open the generated `service-a` project with the IDE of your choice.
- Create a Class called `BookingController` which will contain the JAX-RS endpoint that is customized
- The annotation `org.eclipse.microprofile.openapi.annotations.OpenAPIDefinition` can be used to set some general info at the 'top' of the OpenAPI document. Here we set some information text.

```
@OpenAPIDefinition(info = @Info(title = "Booking endpoint", version = "1.0"))
```

- For each JAX-RS endpoint, we can define all information of the expected responses and what they mean. Look at the `org.eclipse.microprofile.openapi.annotations.responses.APIResponses` on the method.
- Add a `org.eclipse.microprofile.openapi.annotations.Operation` annotation with some additional information.

```
@Operation(description = "Some more info about the endpoint")
```

Deploy the application and request the OpenAPI description again. For the /booking endpoint, you will see the inclusion of the description.

There are many more annotations available to customize the generated specification. Have a look at the `org.eclipse.microprofile.openapi.annotations` package.

OpenTracing

https://blog.payara.fish/new-feature-in-payara-server-payara-micro-5.182-microprofile-opentracing

https://github.com/OpenLiberty/guide-microprofile-opentracing

https://github.com/kumuluz/kumuluzee-samples/tree/master/kumuluzee-opentracing

JWT

Security is another important aspect of the micro-service architecture. With the help of JWT tokens (JSON Web Token) we can securely transfer some information about the user from one service to another. It also allows propagating user identification from one service to another service in a decentralised way so that the verification can’t be the bottleneck in your environment.

The JWT format is defined of course, but the specification defines the required claims and how a MicroProfile implementation should handle the authorization aspects for example.

Secure an endpoint

The MicroProfile Starter application already generated an example of how you can use the MicroProfile JWT spec to secure a JAX-RS endpoint. Let us go over the important parts of this example.

- Open the generated `service-b` project with the IDE of your choice
- Have a look at the `DemoRestApplication` where we define that some of the endpoints will be protected using a JWT token.

```
@LoginConfig(authMethod = "MP-JWT")
@DeclareRoles({"protected"})
```

- The `org.eclipse.microprofile.auth.LoginConfig` defines the login configuration and the `javax.annotation.security.DeclareRoles' annotation defines the roles we will be using within the application.
- The `ProtectedController` has a JAX-RS endpoint defined which can only be accessed by a request which contains a token having the `protected` role.

```
@GET
@RolesAllowed("protected")
public String getJWTBasedValue()
```

- That same `ProtectedController` class shows you another feature of the specification, injecting some values defined in the claims of the JWT.

```
@Inject
@Claim("custom-value")
private ClaimValue<String> custom;
```

- For the validation of the JWT token, we need the public key of the signature. The example still uses the 1.0 version of the spec where this part was not yet standardised. So the key is defined in an implementation-specific way. Have a look at the src/main/resources directory and you should see the data.

The other part of the example is in the `service-a` part of the generated example.

- Open the generated `service-a` project with the IDE of your choice
- Have a look at the `TestSecureController` java class. During the construction of the class, a private key in the PEM format is read.

```
    public void init() {
        key = readPemFile();
    }
```

- The class has an endpoint, that creates a JWT, creates a RestClient (JAX-RS version), and adds the JWT as a header.

```
        String jwt = generateJWT(key);
        WebTarget target =
ClientBuilder.newClient().target("http://localhost:8180/data/protected");
        Response response = target.request().header("authorization", "Bearer " +
jwt).buildGet().invoke();
        return String.format("Claim value within JWT of 'custom-value' : %s",
response.readEntity(String.class));
```

All JWT tokens must be passed as a authorization header to the endpoints so that the MicroProfile implementation can pick it up. * Have look at the `generateJWT` method for the various parts that are defined within the JWT.

```
        token.addAdditionalClaims("custom-value", "Jessie specific value");
        token.setGroups(Arrays.asList("user", "protected"));
```

The custom value is set as claim and read by the ProtectedController as we have seen earlier. Also, the group is specified `protected` in the example so that the authorisation allows access to the endpoint we call.

For testing it out, make sure the `service-a` and `service-b` are running and call the URL `<host>/<root>/data/secured/test`. You can also play with some 'parameters' to learn more about the specification.

- Try another 'group' value.
- Generate a new key pair and use the wrong public key in service-b.
- Try to read other claims for the JWT in the `ProtectedController`