Quarkus with JakartaEE

Course Handout

Table of Contents

Chapter 1. Introduction to Quarkus, project setup and structure, developer experience

1.1. Introduction to Quarkus

1.1.1. What is Quarkus?

> Traditional Java stacks were engineered for monolithic applications with long startup times and large memory requirements in a world where the cloud, containers, and Kubernetes did not exist. Java frameworks needed to evolve to meet the needs of this new world.
> Quarkus was created to enable Java developers to create applications for a modern, cloud-native world. Quarkus is a Kubernetes-native Java framework tailored for GraalVM and HotSpot, crafted from best-of-breed Java libraries and standards. The goal is to make Java the leading platform in Kubernetes and serverless environments while offering developers a framework to address a wider range of distributed application architectures. - "What is Quarkus?", https://quarkus.io/about/

In other words Quarkus is a Java application server, that implements its own version of the Java EE spec, and provides the most modern and up-to-date capabilities to help Java and Java EE catch up with the latest technologies and standards. Unlike traditional Java application servers, Quarkus is lightweight, it has a small consumptional fingerprint on the system, and it starts up runs blazingly fast. These features allow for more scalable applications, less powerful hardware and serverless applications to be created and used. As a result development becomes easier, deployment becomes faster and costs become lower.

Here are some good points to use Quarkus:

- It implements most of the Jakarta EE and MicroProfile specifications - with Quarkus you have most of the Java / Jakarta EE specs provided, along wih MicroProfile, you're getting the most out of the up to date technologies to implement modern web applications.
- Fast startup time and low memory footprint - a Quarkus project includes only the stuff you need and use in your project. Implementations are also deeply optimized to give you the best performance for your application.
- Native mode - in combination with Graal VM, you can build native AOT applications with Quarkus. This allows even smaller memory footprint, even quicker startup time (under a second).
- Great developer experience - Quarkus allows you to test and develop, while your app is running, similar to Node.js. You need to change something? No need to restart the server and wait to redeploy again. Simply refresh the page, or give it another REST call and the app will redeploy automatically.

You can read more about Quarkus' performance here.

After Quarkus version 3 all the packages from Java EE are now using Jakarta EE, which means that packages starting with `javax` now start with `jakarta`.

1.1.2. What is this course about?

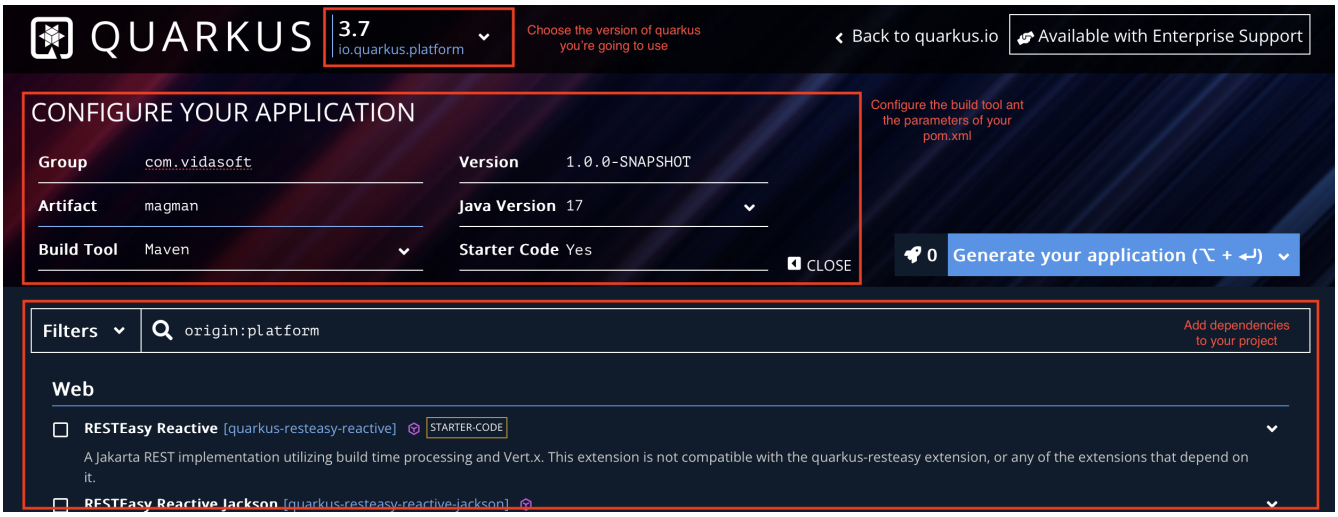
This course will walk you through the basics of Java/Jakarta EE using the Quarkus development framework. We will wall you through the most modern and up to date specs of Jakarta EE and how to develop your applications with Quarkus. At the end, you should have the common knowledge to write your own applications or be able to understand other applications and get involved with your team project easily.

Chapter 2. Creating your first application with Quarkus

Now that you know what Java/Jakarta EE and Quarkus is, it is time to create our own Quarkus application. This section will give you a quick start, showing you the ways you can create and run your a Quarkus project. Next lessons will aim to extend that application, with intent to upgrade your knowledge and understandings. At the end you will have a functioning example of all important Java EE specs implemented into this project.

2.1. The code.quarkus.io approach

Starting on a new project? It can always get a bit frustrating, having to configure your environment and deal with all the specific configurations and XMLs within your project. Don't worry though, Quarkus has you covered. All you need to do is go to https://code.quarkus.io. There you will find simple and easy project generator, that will help you to set the name of the project and your package domain (group and artifact id)



The interface is quite easy to use and understand. You can choose the version of Quarkus, the group and artifact id of your application, the version of the JDK of your application, as well as the version of your application.

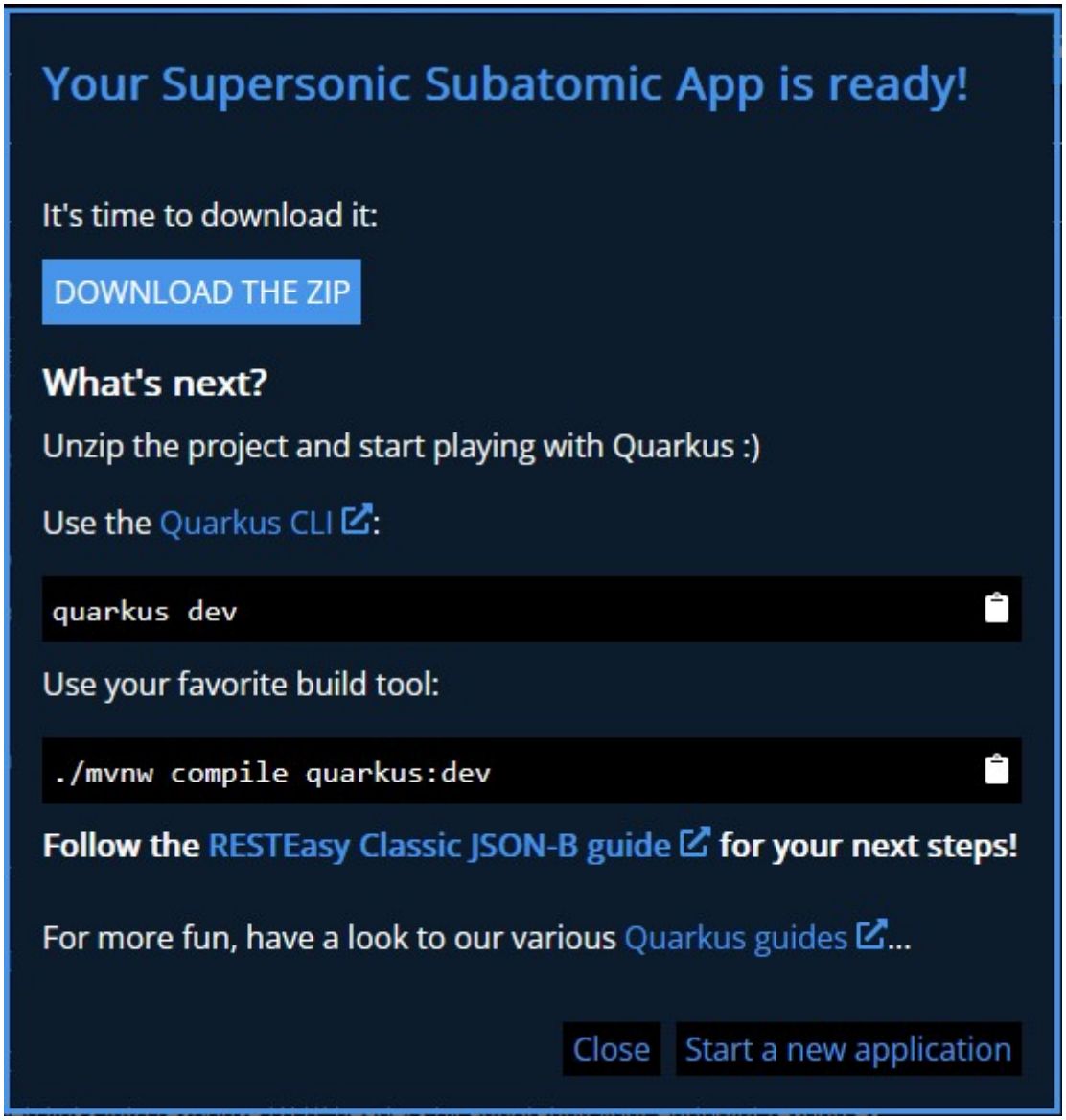
Finally you are able to decide which build tool to use (either `Maven`, `Gradle` or `Gradle with Kotlin DSL`), and whether you want any "Starter code" generated or a blank project. After that you are welcomed with a list of Quarkus Extensions you are able to choose for your generated project. These can be extended, removed, depending on your needs. More of that, you'll find in our next lessons.

2.1.1. Creating your first project

For the sake of ease, you can follow the configurations as shown above. Then add the following extension to your project:

RestEasy with JSON-B are that are the first libraries that we are going to use, in order to look through our first Java EE specification - JAX-RS.

Once you have checked this extension, go and click the `Generate your application button`. A modal window will give you the different ways you can obtain your project.



Following the instructions, there will be three options:

1. Directly download the generated project tree, zipped.
2. Use the Quarkus CLI

3. Use maven

If you want to go pro, you can try and create your application using the Quarkus CLI or maven.

2.2. Creating your app with Quarkus CLI

Quarkus has provided a convenient command line interface, that allows easy interactions with your project configuration, so you can create, personalize and modify your Quarkus extensions in a jiffy. If you feel more experienced with terminals and command lines, this would be a better approach for you to quickly generate your project.

To generate a project with Quarkus CLI, simply execute the following code:

```
quarkus create app com.vidasoft:magman \
    --extension=quarkus-resteasy-jsonb
```

2.3. Creating your app with Maven

A more verbose approach to generate your first Quarkus application would be by using Maven. To achieve the same result, you can simply execute the following line:

```
mvn io.quarkus.platform:quarkus-maven-plugin:3.7.1.Final:create \
    -DprojectGroupId=com.vidasoft \
    -DprojectArtifactId=magman \
    -Dextensions="quarkus-resteasy-jsonb"
```

To view more possibilities to generate and personalize your Quarkus project, please visit https://quarkus.io/guides/rest-json

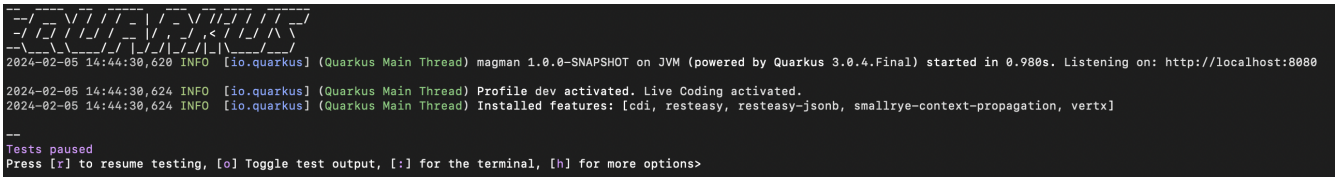
2.4. Let's start your project.

2.4.1. What do we need to get started?

- Java 17 or greater
- Maven 3.9.3 or greater
- A computer with internet connection

If you want to work with Quarkus CLI, you would need to install it first. Read this article to learn more.

Now that you have the project created, let's see it running. To run your code, simply execute `mvn quarkus:dev`.



Once you see this in the command line, you can be assured your application is up and running.

Now let's test that, shall we?

Execute the following cURL:

```
curl --location --request GET "localhost:8080/hello"
```

You should be greeted with the following response:

```
Hello RESTEasy
```

2.5. Developing with Quarkus during runtime

One of the cool things about Quarkus is that you can develop your applications and run tests while the application is still running. Let's put that in test.

Start up your Quarkus application if you haven't already. Then let's call our REST endpoint to make sure it's working again.

```
curl --location --request GET "localhost:8080/hello"
```

Now let's try and change the response. Go to `src\main\java\com\vidasoft\GreetingResource.java` and edit the return String. Change the message to something else.

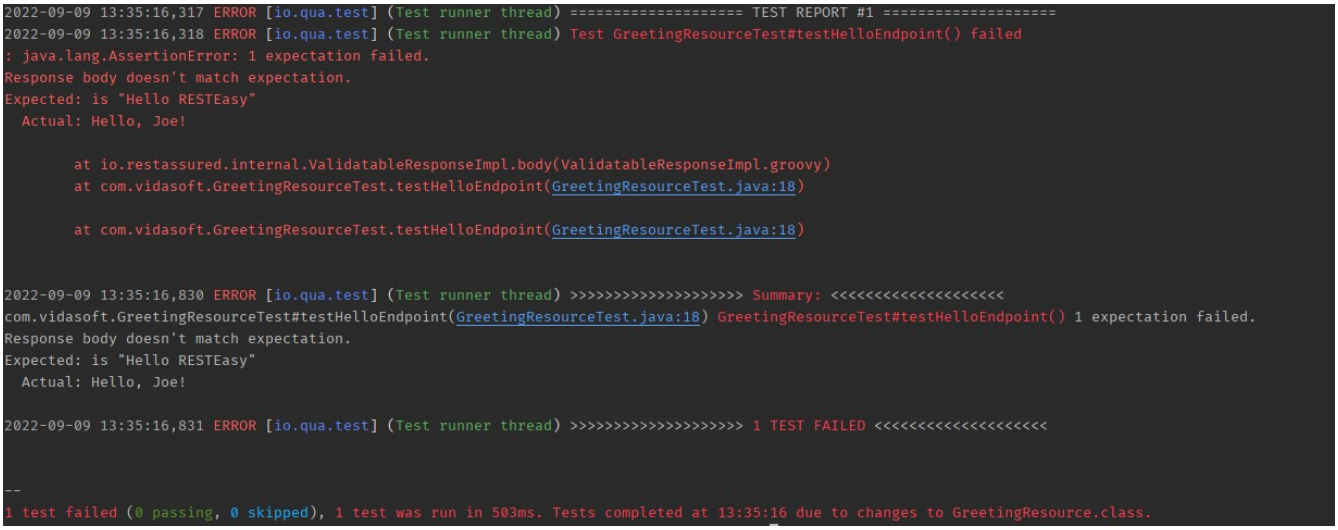
```
...
return "Hello, Joe";
...
```

Now curl rhe endpoint again. Did it work? You should be seeing the new response in your console.

2.5.1. Running tests in runtime

The startup project comes with a test for the hello endpoint built in. Let's try and run that test while the app is running. Press `r` inside the window where your project is running.

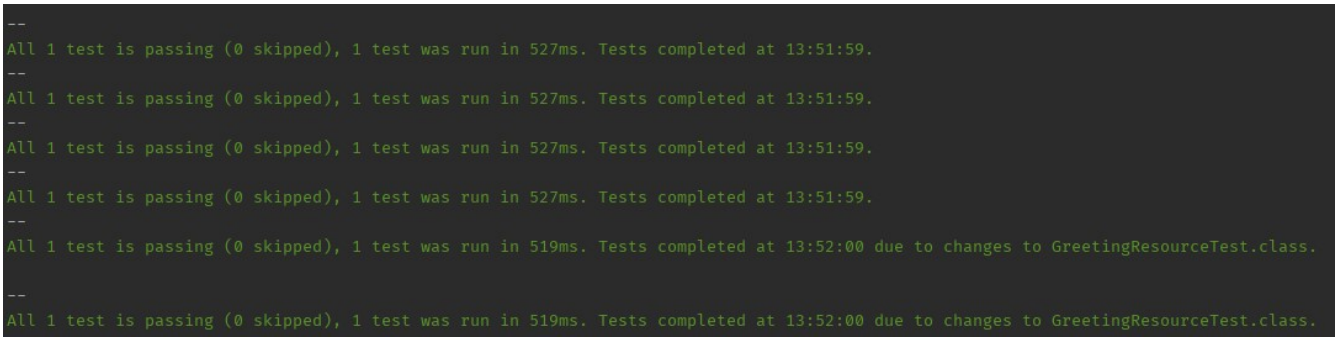
If you followed the previous section, the test should fail.



Now let's fix that and let the test pass. Go to `src\test\java\com\vidasoft\GreetingResourceTest.java` and change the expected message to the new response you have set.

```
    @Test
    public void testHelloEndpoint() {
        given()
          .when().get("/hello")
          .then()
             .statusCode(200)
             .body(is("Hello, Joe!"));
    }
```

Press `r` in the application's window again, and observe the results.



Chapter 3. Project structure with Quarkus and Maven

Now that we have successfully created our project, let's get to know its structure, so we can comfortably know our way around.

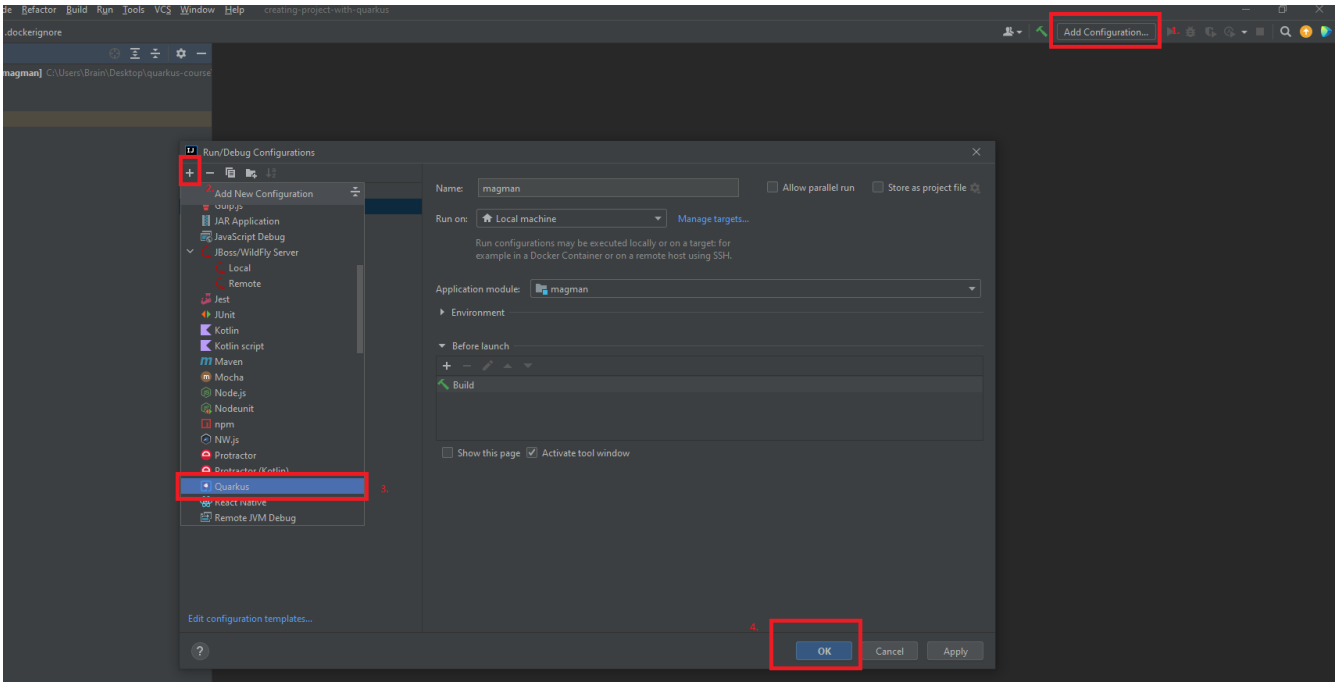
3.1. Configuring your IDE for the project

Since we are working with abstract and sophisticated matter, let's first load our project into our IDE. All Java IDEs are supporting the Java/Jakarta EE annotations, and most popular ones such as IntelliJ IDEA and Eclipse support the Quarkus framework out of the box, which means the IDE will know how to set up your environment the moment you load the project there.

For the purpose of this course we are going to use IntelliJ IDEA, but you can go along with any IDE, you feel more comfortable with.

To load the project, simply click `Open` from the project selection window or `File → Open` and choose the folder of your project. IntelliJ should quickly recognize your project and framework and create a runner. If no runner has been created, you can easily add it yourself.

Click on `Add Configuration` → `+` → `Quarkus` → `OK`

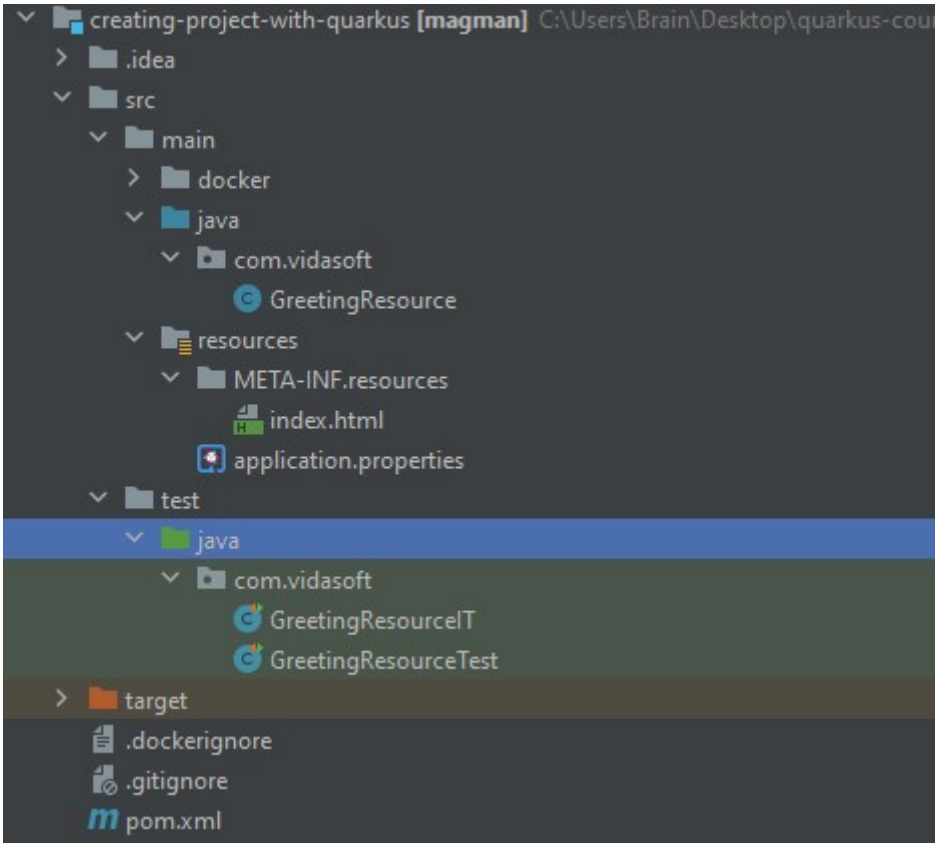


Other than that, no other configurations are needed.

Now let's start your project and make sure it's configured properly. First make sure you have stopped all other processes, you previously started running `mvn quarkus:dev`. Then run the project from your IDE. To test that your application is running fine, let's make that curl call once more:

3.2. Looking at our project file tree

Having our project configured, now it is time to look at our file tree.



Most of the project's structure is common to a lot of Java Enterprise projects, but we are going run through it anyways, to make sure, we are on the same page.

Starting off with the `src/main` folder, we have three important directories:

- `docker` is the folder where your Docker configurations will be. Quarkus has provided an easy way to create docker images for your project, so you can use Maven and build/deploy those with a single command.
- `java` will be the directory where your code will be. Folders under this one are considered as packages. It is recommended that you write your code under the package of your project domain, as shown in the image.
- `resources` is the folder where your project's resources will be. Here you can see stuff as configuration properties (`application.properties`), and other templates, presets or files, your project will use for a particular purpose, based on the project's needs.
- `application.properties` is the file where all system configurations will go. We will look at how these properties come in hand in a further chapter.
- `META-INF/resources` is a directory specific for the Quarkus framework. This directory would contain server content that needs rendering. If you try to load http://localhost:8080 in a browser, you will be welcomed to a greeting page, generated by Quarkus, to assure you that the project is up and running. We are going to modify our project and change that as we go to serve as a demo page to our front-end for our back-end server.

- `test` folder as you guessed serves as a folder where our tests will be. Here the structure is mirrored to the `main` folder and we can use it to write tests for our application.

3.3. Adding extensions to Quarkus

Quarkus extensions are Maven dependencies which enable the framework to use annotations and functionalities, based on the application's needs. During our project we will add and modify those extensions, for our needs, when we want to enable different Jakarta EE features. These extensions/dependencies can be managed manually from the project's `pom.xml` or we can use the maven cli to add them with a simple one-line command.

Let's first go to our `pom.xml` and see what we have in our `<dependencies>` block.

```
<dependencies>
  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-jsonb</artifactId>
  </dependency>

  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-arc</artifactId>
  </dependency>

  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy</artifactId>
  </dependency>

  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-junit5</artifactId>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

These are the extensions we chose to add during our project setup with code.quarkus.io. They will enable us to use CDI (with `quarkus-arc`) and JAX-RS (with `quarkus-resteasy`). The dependencies with `<scope>test</test>` were automatically added by Quarkus to enable us to write tests for our project, using the extensions we added in the first place.

3.4. What’s next?

In the next chapter we are going to look through our first Jakarta EE specification, the Java Persistence API or JPA for short.

Chapter 4. Building REST endpoints with JAX-RS

The first Jakarta EE specification we are going to look through is JAX-RS. JAX-RS is a specification that defines annotations for building restful applications. With JAX-RS you are able to easily build your REST endpoints, which are the entry points of your web application.

The extensions we already have included

Going back to our introduction part, we included the RestEasy with JSON-B extension.

RestEasy is an API that implements the JAX-RS specification. Using the annotations by themselves won't work as Jakarta EE, only provides interfaces that need implementation. Each application server provider decides how to implement those specifications, and for Quarkus, RestEasy is the solution.

JSON-B is a library used to convert Java objects (POJOs) into JSON objects. Used with RestEasy, it allows implicit conversion of request/response bodies from and to JSON, so you don't have to interact with the API directly. You will see how this comes in action during this chapter.

4.1. Analyzing what we already have

Having all that cleared out, let's see how this will come in handy for our project. Let's first look at our hello endpoint and analyze every part of it:

```
@Path("/hello") ①
public class GreetingResource {

    @GET ②
    @Produces(MediaType.TEXT_PLAIN) ③
    public String hello() {
        return "Hello RESTEasy"; ④
    }
}
```

① `@Path` annotation tells the application server where the endpoint is located. This annotation can be used on a class or on a method within that class, which has a specific path/subpath. In our example this would be `localhost:8080/hello`.

② `@GET` annotation defines the method of the for the request. As you may know the most commonly used REST methods are `GET`, `POST`, `PUT`, `DELETE`. Each of those are defining the action you intent to do with the data.

③ `@Produces` is responsible to tell the server what type of data is going to be returned. This helps the server to choose what value to set to the `Content-Type` header, and how to convert the data, passed through the response. The value can be set to a method or a class, if all the methods are

returning the same response time within it.

④ In JAX-RS you have two ways to return a response. If your request's response is simple, you can directly set the return value, which will populate the response's body. But if you would like to return different types of responses, then you'll have to use `jakarta.ws.rs.Response` and create your custom responses. So when you need to return void or an object, `JAX-RS` will behave in the following way:

- When your method returns `void`, the response will be `204 NO CONTENT`.
- If you return an object, you'll get `200 OK` with either JSON or text as a body, depending on the `@Produces` annotation.
- Throwing exceptions like `NotFoundException` or `BadRequestException` will respectively result in `404 NOT FOUND` and `400 BAD REQUEST`.

Using `@Produces` annotation is optional. If you forget or intent not to add it, most of the times Quarkus will be able to define the content type automatically. Despite that it is recommended to use this annotation, because automatic recognition does not work all the time, and you can never know when and for what reason it will fail.

4.2. Creating our own endpoints

Now that we know how REST endpoints are working with JAX-RS, let's try and create our own endpoints, and put them into action.

One of the most important parts of a magazine manager is being able to create and modify Articles. Let's try and create the foundation of the `ArticleResource` - our entry point for accessing articles.

First off, let's add two new packages to our project. Under `src\main\java\com\vidasoft` add the following classes:

- `magman.article.ArticleResource`
- `magman.article.ArticleDTO`

`ArticleResource` *will contain all the endpoints needed for our article.*

```
@Path("/article")
public class ArticleResource {

}
```

`ArticleDTO` *is the object we are going to pass through requests and responses.*

```
public class ArticleDTO {

    private Long id;
    private String title;
    private String content;
    private String publishDate;
```

```
    private String author;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public String getPublishDate() {
        return publishDate;
    }

    public void setPublishDate(String publishDate) {
        this.publishDate = publishDate;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }
}
```

With Java 14 and later you can use records, instead of POJOs, which will free you up from some boilerplating. This will make definitions of DTOs more clean and straightforward. Do keep in mind though, that records are immutable and they don't support inheritance, meaning that you'll have to define the same properties for common classes and won't be able to change a property's content, once it's been set. Read more about records here.

You may have noticed that your IDE is signalling for error when you defined `@Path` on the `ArticleResource` class. This is because there are no endpoints defined for that resource. Let's create our first endpoint:

```
@Path("/article")
public class ArticleResource {

    private static Map<Long, ArticleDTO> articles = new HashMap<>(); ①

    @POST
    @Consumes(MediaType.APPLICATION_JSON) ②
    public Response createArticle(ArticleDTO article) {
        Long articleId = Math.abs(new Random().nextLong());
        article.setId(articleId);
        articles.put(articleId, article);
        return Response.created(URI.create(String.format("article/%s", article.getId(
))))).build(); ③
    }

}
```

① We are going to use a Map to store all of our articles. In the next chapter we will introduce database for this purpose.

② The `@Consumes` annotation defines what type of data we are going to pass to the request's body. It has similar behavior as `@Produces`. If the annotation isn't used RestEasy will use whatever type is defined in the `Accept` header of the request.

③ Here we use `jakarta.ws.rs.Response` to return a custom response to the client `201 CREATED`. This response is the most appropriately accepted when creating new entities. It is specified to contain a path for `GET` requests regarding that entity.

4.2.1. Creating an article

Now that we have our first endpoint created, let's try and create an article.

```
curl -i --location --request POST 'localhost:8080/article' \
--header 'Content-Type: application/json' \
--data-raw '{
    "title": "Ipsum Lorem",
    "content": "The quick brown fox runs over the lazy dog.",
    "publishDate": "2022-01-12",
    "author": "Cave Johnson"
}'
```

As a response you should get:

```
HTTP/1.1 201 Created
Location: http://localhost:8080/article/2559794960439759963
```

```
content-length: 0
```

4.2.2. Getting an article

Let's try and cURL the responded location now:

```
curl -i --location --request GET 'http://localhost:8080/article/2559794960439759963'
```

What we get as a response is:

```
HTTP/1.1 404 Not Found
Content-Type: application/json
content-length: 0
```

Why? Because we don't have a GET endpoint for that yet. Let's create one, shall we?

```
@Path("/article")
public class ArticleResource {

    public Response createArticle(ArticleDTO article) {...}

    @GET
    @Path("/{id}") ①
    @Produces(MediaType.APPLICATION_JSON)
    public Response getArticle(@PathParam("id") Long articleId) { ②
        if (articleId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build(); ③
        } else if(articles.get(articleId) == null) {
            return Response.status(Response.Status.NOT_FOUND).build(); ④
        } else {
            return Response.ok(articles.get(articleId)).build(); ⑤
        }
    }

}
```

① Here we use the `@Path` annotation in combination with `{id}`. `{id}` is used as a placeholder to tell RestEasy, that there is a variable going to be placed there, pointing to the article id.

② In order to get the contents of our `{id}` placeholder we use the `@PathParam` annotation, which will tell our application that the value of `{id}` will be passed to the `articleId` variable.

③ It is recommended that we do checks of the content we are receiving to avoid any errors or exceptions in our app. If the article id is invalid, we should return response `400 BAD REQUEST`.

④ Other check we are making is whether this article exists at all. If the article does not exist, we should return response with status `404 NOT FOUND`.

⑤ When all the checks have completed it is safe to get the requested article from our map. Here we will return `200 OK` with our article as a response.

Now let's try requesting that article again. But keep in mind that you need to create the article again, as our list of articles will reset for its next deployment.

```
curl -i --location --request GET 'http://localhost:8080/article/2559794960439759963'
```

And our response should be:

```
HTTP/1.1 200 OK
Content-Type: application/json
content-length: 137

{"author":"Cave Johnson","content":"The quick brown fox runs over the lazy
dog.","id":"2559794960439759963","publishDate":"2022-01-12","title":"Ipsum Lorem"}
```

Try and create other GET article requests. Observe how the application behaves when you pass invalid article id.

4.2.3. Editing an article

One other important functionality is to be able to edit articles. Let's implement that into our REST demo:

```
@Path("/article")
public class ArticleResource {

    public Response createArticle(ArticleDTO article) {...}

    public Response getArticle(@PathParam("id") Long articleId) {...}

    @PUT
    @Path("/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response editArticle(@PathParam("id") Long articleId, ArticleDTO article) {
        if (articleId < 1 || !articleId.equals(article.getId())) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else if (articles.get(articleId) == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        } else {
            articles.put(articleId, article);
            return Response.ok(article).build();
        }
    }
```

```
}
```

Again it is recommended to do some checks, before performing the operation. This will guarantee us, that we are editing the right article and limit the risk of obvious errors.

4.2.4. Deleting an article

Creating and editing articles is great! So far so good, but sometimes we want to be able to get rid of them. Let's create a method that deletes an article.

```
@Path("/article")
public class ArticleResource {

    public Response createArticle(ArticleDTO article) {...}

    public Response getArticle(@PathParam("id") Long articleId) {...}

    public Response editArticle(@PathParam("id") Long articleId, ArticleDTO article)
{...}

    @DELETE
    @Path("/{id}")
    public void deleteArticle(@PathParam("id") Long articleId) {
        articles.remove(articleId); ①
    }
}
```

① Here we are not interested whether the article exists or not, as we want to delete it, and we are not interested what the response will be. That's the reason why we don't return a custom response here. In JAX-RS void methods will automatically return `204 NO CONTENT`.

Let's now execute the cURL:

```
curl -i --location --request DELETE
'http://localhost:8080/article/2559794960439759963'
```

Did you get `HTTP/1.1 204 No Content`? If yes, than your work is finished. Now try and perform a GET request to see if that article is still there. You should be getting `404` now.

4.2.5. Getting a list of articles

We will need to preview a list of our articles. Getting them one by one, wouldn't be a good solution. Let's create an endpoint to get all articles.

```
@Path("/article")
public class ArticleResource {
    ...
```

```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Collection<ArticleDTO> getArticles() {
        return articles.values();
    }
}
```

Here you may have noticed two things. We are not using the `@Path` annotation on the method, and we are not returning `Response`, but `List<ArticleDTO>` instead. As we said earlier, if we do not have any deeper path for the request, the method will take whatever path is defined on class level. As for the return value, in this example we have no reason to return a custom response. There is nothing to check, nothing to validate, nothing more to add than just the list of articles. But don't worry. This is about to change very soon.

Now that you have created the `getArticles` endpoint, let's make some articles and try to call that endpoint.

```
curl -i --location --request GET 'http://localhost:8080/article'
```

As a response you will see a JSON array with all the articles you have created.

4.2.6. Creating a custom request

Currently we are able to create, get, update and delete articles. That's great! We have all the CRUD operations, needed to perform basic actions for our articles. But there's one issue with that method getting all the articles. When we call it, we are going to get either all articles or no articles, in the case where no articles have been created. In a situation where we have 1000 articles, this wouldn't be useful, would it?

Instead of returning all the data to the client, it'd be better to give it some data, and if it asks for more, then we give more. This would make our server perform better, leading to better user experience at the end.

Let's touch our `getArticles` method a bit, shall we?

```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getArticles(@QueryParam("page") @DefaultValue("1") int page,
                                @QueryParam("size") @DefaultValue("10") int size) { ①
        if (page < 1 || size < 0) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            var articles = this.articles.values().stream()
                    .skip(((page - 1L) * size)) ②
                    .limit(size)
                    .collect(Collectors.toList());
            return Response.ok(articles).build();
```

```
        }
    }
```

① `@QueryParam` tells JAX-RS, that the variable will be taken from the query parameter, named `page`. `@DefaultValue` tells JAX-RS to set that value to the argument, if the query parameter has not been passed.

② The formula here is simple. Since arrays are zero-based, we subtract 1 from the page number and multiply it by the size, to calculate how many results we want to get from the list. For example if the page is `1`, and the size `10` then we will skip 0 entries and take only 10; if the page is 2, we will skip 10 entries and get 10 again, and so on, and so forth.

Now let's create our articles again and try the new query and its parameters:

```
curl -i --location --request GET 'http://localhost:8080/article?page=1&size=5'
```

Try to create different types of requests and observe what kind of results you get here.

4.2.7. Conclusion

We are going to look though more of the JAX-RS features during our next sessions, while continuing our journey into making the magazine manager.

Chapter 5. Java Persistence API (JPA)

One of the core functions of an application server is being to work with a database. Functions as extracting data from the database, adding or removing data on demand are crucial to all business oriented web applications.

We'll continue our journey into building a magazine manager web application with setting up its data model. The standard way to do that and to then persist the modelled data is with the Java Persistence API (JPA).

JPA Terminology

Entity - A non-final plain old Java object (POJO) with a non-argument constructor. It represents a database table with its member variables representing the table columns.

Persistence context, **EntityManager** - Both terms mean one and the same thing. It is the interface that you use to load and store entities from and to the database. It acts also as a first level cache for your application.

Persistence unit, **EntityManagerFactory** - The factory which you use to obtain the entity manager. There you configure the way to connect to the database - a server datasource or direct JDBC properties.

JPQL - Object-oriented query language, based on SQL. It works on objects and their fields, rather than on tables, columns and relationships.

As a start we will need to add the proper configurations and extensions to make our project work with a database.

5.1. Adding extensions to support JPA

As we mentioned in the beginning, our project will include only the stuff we need, and right now although we have annotations for JPA, they won't work, without the implementations, required to run them.

In order to work with a database, we will need to add a couple more extensions to our project:

- **quarkus-hibernate-orm** - will allow to use Hibernate's JPA implementation.
- **quarkus-jdbc-h2** - the database driver, used to communicate with the database using Hibernate's APIs

We'll use H2 as embedded database in this installment of our series.

To add extensions to quarkus, simply execute the following command in the terminal:

```
mvn quarkus:add-extension -Dextensions="io.quarkus:quarkus-hibernate-orm,
```

```
io.quarkus:quarkus-jdbc-h2"
```

Now if you open `pom.xml`, you'll see that we have two new dependencies added to the list:

```
    <dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-jdbc-h2</artifactId>
    </dependency>

    <dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-hibernate-orm</artifactId>
    </dependency>
```

5.2. Adding configuration to the database

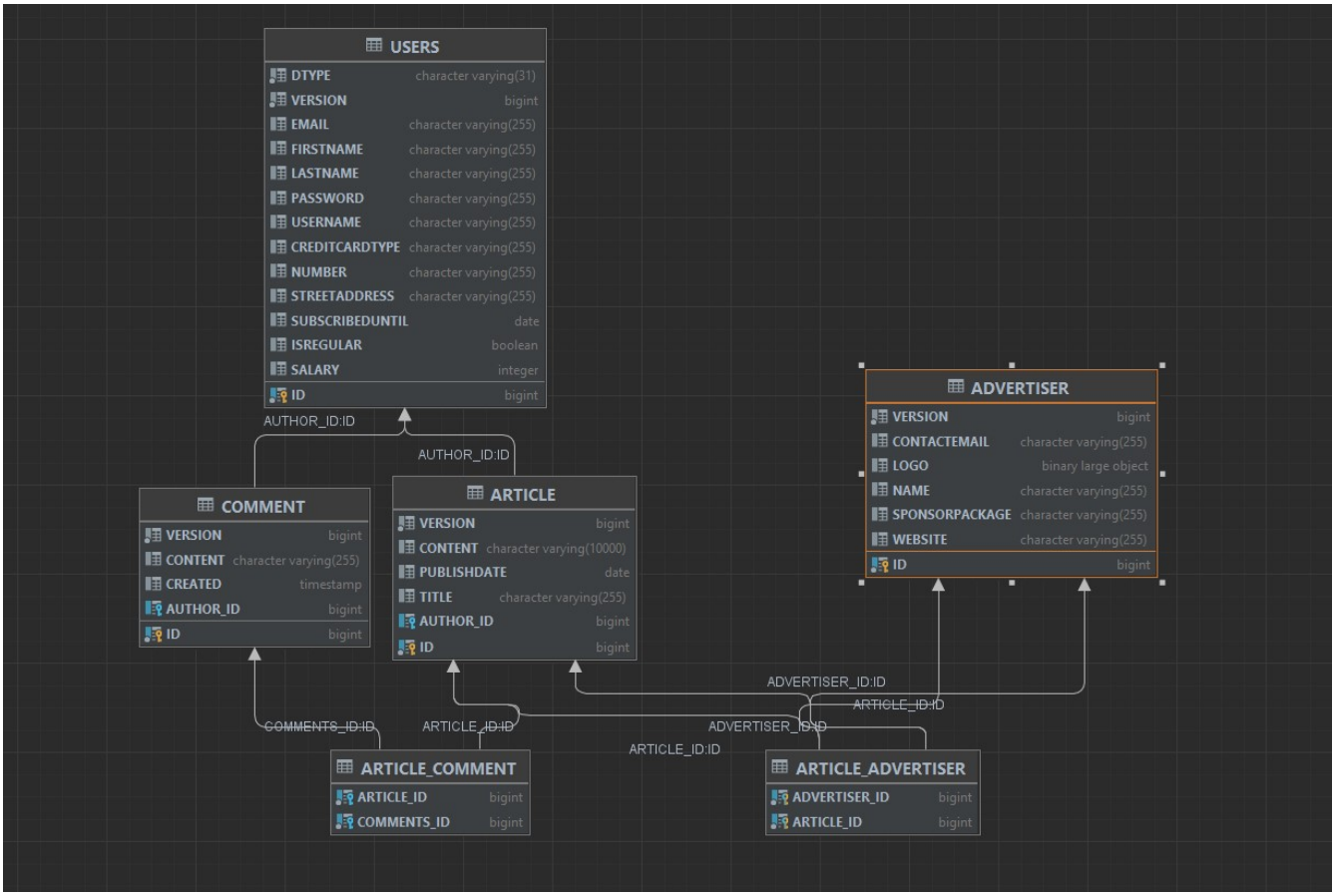
Having just the extensions added, won't help the project communicating with the database. We also need to point it to where and what the database is. To do so, we are going to add some properties into our `/src/main/resources/application.properties` file.

```
# configure your datasource
quarkus.datasource.db-kind = h2
quarkus.datasource.username = sarah
quarkus.datasource.password = connor
quarkus.datasource.jdbc.url = jdbc:h2:file:./data/magman;DB_CLOSE_DELAY=-1

# drop and create the database at startup (use `update` to only update the schema)
quarkus.hibernate-orm.database.generation = drop-and-create
quarkus.hibernate-orm.log.sql=true
```

Depending on the database, your configuration might vary. In a real world scenario, you will need to configure database username and password or set a different type of generation mechanism. You can view other examples for database configuration here.

5.3. The database model

Now that we have our database configured, let's upgrade our project by creating our database model. For that purpose, we are going to follow the diagram below.



First off, create a new package, named `model` under `com.vidasoft.magman`. This will help us separate our database ORM layer from the rest of the project. Then, let's build the core classes for the database:

```
public class User {
    ...
}

public class Article {
    ...
}

public class Comment {
    ...
}

public class Advertiser {
    ...
}
```

After that's been done, let's start shaping our model classes...

5.4. Defining object-relational mapping with JPA

Depending on the version of Java you use, the following annotations could be

found either in packages `jakarta.persistence` or `jakarta.persistence`.

Starting off with the User class, the first and most important thing we'll need to set is the annotation `@Entity`. This is an annotation, telling JPA, that this class is used in relation with a database table.

```
@Entity
@Table(name = "Users") ①
public class User {}
```

① Here we are also adding an optional annotation defining the name of the table.

In most databases the word "user" is reserved to the language, as it refers to the user of the database and not some name of a table. Therefore you should avoid using the name "user" for a table.

Sometimes you will have to model your Java objects or tables, based on an already existing database. Usually JPA will match the name of tables and columns by their Java names, but in the case where the database already exists, names might not be exactly the same.
For example you might have a table, named `CUSTOMER_USERS`, but you want to map it to a class called `User`. To address that, you'll need to use the `@Table` annotation and set the name to the table name, related.
The same thing can be done for the columns, using the `@Column` annotation.

Next important thing is to define a property for `id`, which will be our primary key to the database.

```
public class User {

    @Id ①
    @GeneratedValue(strategy = GenerationType.AUTO) ②
    public long id;
}
```

① `Id` annotation tells JPA that this property maps the id column of the table.

② `GeneratedValue` is an annotation, that tells Hibernate how to handle id generation for new entities. In this example `GenerationType.AUTO`, would tell Hibernate or any other ORM implementation to handle Id generation with its default setting four auto generation. The most common approach is generating a database object, called sequence, which keeps track of the last id used and assign the next one to the entity.

You may be tempted to encapsulate the properties with getters and setters, but hold on for a minute and think about why? When we are trying to access/modify the properties of our variables, we are always going to get or set the exact values we want to assign, so using getters and setters is not necessary in such scenario.

One of the things we also need do decide during database modeling is how do we want the database

to behave, when two or more people are trying to access it. Do we want the access to be one user at a time, or do we want all users to have access at any time. This decision is usually based on how strict we want to be on database integrity, and how crucial is that to the project. The procedure we use to control that access is called locking.

Optimistic vs pessimistic locking

When the access to the database is limited to one user at a time, we use the term **pessimistic locking**. This means that when a user performs `SELECT`, `INSERT`, `UPDATE`, `DELETE`, etc. operations, other users are not allowed to access the database, and will have to wait for the initial operations to finish. This type of locking guarantees that everyone is working with the most recent and up to date data and prevents conflicts, where two users might be editing the same table or column.

The other most commonly used method is called **optimistic locking**. This type of locking relies that every table row has a specific column, that will keep track on the data's recency by versioning. That way if two users edit the same data, the first one who's going to save it, prevents the others from overwriting it, by version comparison. When the database has been modified, Hibernate will check the version of the persisted entity compare it against the version of the next modification wanting to be persisted. If the new entity has a version lower than the version in the database, it won't be persisted and the transaction will be rolled back, meaning all changes, no matter their relation to that entity will be dismissed.

For our project we are going to use **optimistic locking**. It is the most commonly used type of locking as it is sufficient enough to keep the data's integrity intact.

```
public class User {
    ...
    @Version ①
    long version;
}
```

① With the `Version` annotation we tell our ORM that this is the field to compare when modifying data, to check whether we are working on an old invalid data or new one.

Now that we have that set, we have the bare minimum for creating object-relational mapping for our database. We can go and add those fields to the other model classes OR there is a more neat solution to that. If we copy-paste `id` and `version` fields to every class, we're avoiding one of the main principles of using an OOP language - abstraction.

To include abstraction and allow every entity to use the same fields, let's create the following class:

```
@MappedSuperclass ①
public abstract class AbstractEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
```

```
    public long id;

    @Version
    public long version;

}

@Entity
@Table(name = "Users")
public class User extends AbstractEntity {
    //id and version are shifted to AbstractEntity, so this class is empty now.
}
```

① `MappedSuperclass` is an annotation crucial for how we define abstractions in our ORM. It tells Hibernate, that we are going to have these columns in every table.

Abstraction in JPA

There are four types of entity abstraction and depending on the data model structure we have to work with, we get to choose which one is more appropriate to use:

- **MappedSuperclass** – the parent classes, can't have the `@Entity` annotation. This approach will set all the columns of the superclass to the inheriting entities.
- **Single Table** – this approach will use one table for all inheriting entities. Querying an entity, inheriting this class, will return only the columns regarding that entity.
- **Joined Table** – this will create one common table for all entities, containing the common properties. Querying an entity, inheriting such a class, will create a join query.
- **Table per Class** – the resulting schema is similar to the one using `@MappedSuperclass`. But Table per Class will indeed define entities for parent classes, allowing associations and polymorphic queries as a result.

You can find more about ORM abstraction here.

5.5. Modeling the rest of our entities

Now that we have the main objects for our database model, let's add all of their columns as properties to our classes.

5.5.1. The `User` class

For our project we are going to have three types of users:

1. Author - will create and comment on articles
2. Subscriber - will be able to read articles, add comments and subscribe to an Advertiser package.
3. Manager - will have root access to the website and be able to control the application.

Just like the `AbstractEntity`, we're again facing another abstraction dilemma. This time, instead of using `@MappedSuperclass`, we are going to use the Single table strategy.

```
@Entity
@Table(name = "Users")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE) ①
public class User extends AbstractEntity {...}

@Entity
public class Author extends User {}

@Entity
public class Subscriber extends User {}

public class Manager extends User {}
```

① With the `Inheritance` annotation, we're telling Hibernate how to treat this abstraction. In the case of Single table, we can omit the annotation, as it is the default choice in JPA, when it comes to inheritance. Here, we are using it, just to demonstrate how it is usually done.

Having the abstractions set, we're ready to fully define our `User` entity:

```
@Entity
@Table(name = "Users")
public abstract class User extends AbstractEntity {
    public String userName;
    public String password;
    public String firstName;
    public String lastName;
    public String email;
}
```

Now let's define the other three classes:

```
@Entity
public class Author extends User {
    public boolean isRegular;
    public int salary;
}

@Entity
public class Subscriber extends User {
    public String streetAddress;
    public LocalDate subscribedUntil;

    @Embedded ①
    public CreditCard creditCard;
}
```

```
@Entity
public class Manager extends User {
    //This entity has no additional properties for now
}
```

① In the database properties from this class are stored in the same table. Using the `Embedded` annotation helps us to achieve a `has-a` relationship, without creating an additional table to the database.

If you look at our database model at the beginning, you will see the column `DTYPE` for the `Users` table. This is a column, generated automatically by the ORM, to tell what kind of user class the row is related to. Its value can be either Author, Subscriber or Manager.

5.5.2. The `CreditCard` class

The `@Embeddable` annotation

Just like abstraction, we get to choose what kind of composition we want, regarding to our ORM model. We may have the database structured in one way, but want to use it in another.

In our example the credit card would typically be one-to-one relationship, and we would have two ways to build this relation in our object model

- Add one-to-one relationship, using `OneToOne` annotation. This will add another table for the credit card entity.
- Create an embeddable object, which is going extend and bind columns of the same table (the `Users` table in our case). This is used to allow a more object-oriented experience, in a scenario where the database is not modeled the same way.

To satisfy the `Subscriber` class, let's create the `Embeddable CreditCard`:

```
@Embeddable①
public class CreditCard {

    public static final CreditCard DEFAULT = new CreditCard("", CreditCardType.VISA);

    public String number;

    @Enumerated(EnumType.STRING)②
    public CreditCardType creditCardType;
}

public enum CreditCardType {

    VISA("Visa"), MASTER_CARD("MasterCard"), AMERICAN_EXPRESS("American Express");
```

```
    private String displayName;

    CreditCardType(String displayName) {
        this.displayName = displayName;
    }

    public String getDisplayName() {
        return displayName;
    }
}
```

① Annotating a class with `Embeddable` annotation, means that it cannot be an `Entity`. Therefore no table will be created for it. Take it as an extension to another entity. Other than that, it behaves like any other JPA class. On the other side, where the embeddable class is going to be embedded, we just need to add the `@Embedded` annotation (just like we did in the `Subscriber` class). It can contain relationships and have all types of annotations supported.

② The `Enumerated` annotation helps the ORM to define how this field is going to be shaped into a column. There are two strategies:

- `EnumType.STRING` (Recommended) - will persist the enum as varchar string. (e.g. `CreditCardType.VISA` would be persisted as `"VISA"`)
- `EnumType.ORDINAL` - will persist the enum as an integer, depending on the position of the enumerated value.

Using `EnumType.ORDINAL` is prone to bugs. This is due to the nature those ordinals are assigned. If for example we have `VISA, MASTERCARD`, their ordinals would be `0, 1`, but if somebody swaps those, the order of the ordinals will remain the same. This means that it is very easy to change the Credit card type by a mistake and the ORM will not be able to recognize that. To avoid it, it is better to use `EnumType.STRING`.
When it comes to changing the name of the enum, it is better to get a runtime error, when casting that name to enum, than getting seemingly unrelated errors, due to switched enum order.

5.5.3. The `Article` class

We are going to define the `Article` class as follows:

```
@Entity
public class Article extends AbstractEntity {
    public String title;

    @Column(length = 10_000) ①
    public String content;

    public LocalDate publishDate;

    @ManyToOne ②
```

```
    public Author author;

    @OneToMany ③
    public List<Comment> comments = new ArrayList<>();
}
```

① Dy default varchar fields in databases are with a length of 255. An article's content would probably contain a lot more characters than that. With the `@Column` annotation we can define different length, suitable to store an article.

② Defines that one author has many articles

③ Defines that one article has many comments

The proper way to define relationships in ORM

The common way of mapping one-to-many or many-to-one relationships would be to have reference in both entities, meaning that `Article` should have many-to-one relationship with the `Author`, and the author should have one-to-many relationship with the `Article`. This would mean that we would also need to add `List<Article> articles` to the `Author` entity, just like we're doing with the comments.

But wait! That's not all!

We actually don't want to do that.

And why?

Imagine the following scenario...

You want to get a list of all authors. But along with that, you might also be getting the list of all of their articles. This would add a lot of load to the database. You want to get articles only when you need them, as much as you need of them. With the one-to-many approach you are able to get all or none. Imagine if the amount of articles is 1000 or more...

But why did we leave the one-to-many relationship for the comments?

Consider this relationship as one-to-**few**. In our scenario when we ask for an Article, we would also be wanting all of its comments.

But there's more. Loading child data, by default, is lazy, meaning that it will be queried only if it's needed. More of that, later in this chapter.

To learn more about why many-to-one is the better way to approach large 1:n relationships, you can also read this article here.

5.5.4. The `Comment` class

We're almost at the end of our data modeling. Here is how our `Comment` class should look like:

```
@Entity
public class Comment extends AbstractEntity {

    public String content;

    @ManyToOne
    public User author;

    public LocalDateTime created;

}
```

Nothing particular to add here, let's move on.

5.5.5. The `Advertiser` class

This class will introduce advertisers or sponsors to our magazine.

```
@Entity
public class Advertiser extends AbstractEntity {

    public String name;

    public String website;

    public String contactEmail;

    @ManyToOne
    public Article article;

    @Lob ①
    public byte[] logo;

    @Enumerated(EnumType.STRING)
    public 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;
    }
}
```

① `Lob` is an annotation pointing that the data passed to the database will be of a binary type. It is recommended to use primitive arrays here, because in memory they will keep all values in one array and will be faster to access. Using wrapped object array, such as `Byte`, will add performance delay, as the references to the values will be scattered throughout the system's memory.

5.6. Putting our database into action

Now that we have our database model created, let's put it in use. Starting with our previous topic, let's start saving our articles into the database.

To make things easy, we are going to change each endpoint one-by-one.

5.6.1. Preparing our articles for real use

As you saw in our database model all of our articles have authors. One author has many articles, meaning that an article cannot and should not exist without an author. This means that in order to create an article, we need an author to assign it to. Let's use our JAX-RS knowledge from the previous chapter and create the class `user.AuthorResource`

```
@Path("/user/author")
public class AuthorResource {

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createAuthor(AuthorDTO authorDTO) {

    }

}
```

Along with that, we are going to need a transfer object for the author:

```
public abstract class UserDTO {

    private String userName;
    private String password;
    private String firstName;
    private String lastName;
    private String email;

    //getters and setters
}
```

```
public class AuthorDTO extends UserDTO {

    private boolean isRegular;
    private int salary;

    //getters and setters
}
```

And finally let's add constructors to our `User` and `Author` classes.

```
public abstract class User extends AbstractEntity {
    // properties

    public User() {
    }

    public User(String userName, String password, String firstName, String lastName,
String email) { ①
        this.userName = userName;
        this.password = password;
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
    }
}

public class Author extends User {
    //properties

    public Author() { ①
    }

    public Author(String userName, String password, String firstName, String lastName,
String email, boolean isRegular, int salary) {
        super(userName, password, firstName, lastName, email);
        this.isRegular = isRegular;
        this.salary = salary;
    }
}
```

① When you define a constructor in a JPA class, it is important to add a default constructor as well. JPA is using reflection in order to set the values of the properties from the database. Missing a default constructor will prevent JPA from creating an instance for your class.

5.6.2. The `EntityManager`

In order to be able to access the database, we will need to use a special interface, called `EntityManager`. The entity manager has all the methods needed to read, persist, update and delete entities.

Let's add the `EntityManager` to our `AuthorResource`.

```
public class AuthorResource {

    EntityManager entityManager;

    public AuthorResource(EntityManager entityManager) { ①
        this.entityManager = entityManager;
    }

    public Response createAuthor(AuthorDTO authorDTO) {}

}
```

① Thanks to the Quarkus ARC extension and CDI, we are able to provide instance of the entity manager, by just adding it to the class' constructor. This approach is not that common and we are going to look at a more Jakarta EE way to inject dependencies in our CDI chapter.

Now let's define our `createAuthor` method and bind the DTOs to the entities.

```
@Path("/user")
public class AuthorResource {

    EntityManager entityManager;

    public AuthorResource(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createAuthor(AuthorDTO authorDTO) {
        Author author = new Author(authorDTO.getUserName(), authorDTO.getPassword(),
authorDTO.getFirstName(),
                authorDTO.getLastName(), authorDTO.getEmail(), authorDTO.isRegular(),
authorDTO.getSalary());
        entityManager.persist(author);

        return Response.created(URI.create(String.format("/user/author/%d",author.id)
)).build(); ①
    }
}
```

① When the persist method is called and the `id` field is set to `null` or `0`, the persist method will assign it a new value. This is the reason why we can use the id of the newly created object, without setting it ourselves.

All looks great, now let's deploy our application and create our first author.

```
curl -i --location --request POST 'localhost:8080/user/author' \
--header 'Content-Type: application/json' \
--data-raw '{
    "userName": "cj_123",
    "password": "cj_pass",
    "firstName": "Cave",
    "lastName": "Johnson",
    "email": "cj@apperture-science.io",
    "isRegular": true,
    "salary": 10000
}'
```

`HTTP/1.1 500 Internal Server Error` Why? If you look at the console, you may find the following error.

```
Transaction is not active, consider adding @Transactional to your method to
automatically activate one.
```

5.6.3. The `@Transactional` annotation

Using the `EntityManager` on its own wouldn't help if there is no database session and transaction created. As you know the database is a component detached from our project and requires connection to a server and data transmission. In a real life scenario, the database can be physically separated from the web application, running on a different computer. And since the database is our main origin of truth, we need to be sure what we wite to it is valid and consistent. That's why transactions come in action. The lifecycle of a database transaction is as follows:

1. Create a database session (connect to the database)
2. Start a transaction
3. Execute select, update, delete operations
4. Commit/Roll-back the transaction
5. End the database session

Using JPA on its own will require you to create those sessions and transactions manually, by invoking methods from the `EntityManager` interface. But having manual control over the transactions is a bit advanced for simple things, that could be managed automatically. Jakarta EE and Quarkus have us fortunately covered. Quarkus implements the JTA specification, which allows the transaction to be managed automatically by the application. The only thing we need to do is to annotate our method with `@Transactional`.

The Java™ Transaction API (JTA)

JTA is a general API that allows transactions in Java to be managed in a neutral way. The session and transaction is managed by the application server, instead of the developer, which means the time a transaction opens and closes is managed by the server as well.

The alternative to JTA is called *local transaction*. The local transaction allows the developer to decide when a session and transaction should be created and when it should be committed/rolled back.

Read more about JTA in this article.

Knowing all that, let's make our method transactional...

```
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Transactional
    public Response createAuthor(AuthorDTO authorDTO) {
        //code...
    }
```

That's all. Now if you try and execute your request again, the author should be persisted.

```
HTTP/1.1 201 Created
Location: http://localhost:8080/user/author/1
content-length: 0
```

- `@Transactional` can be used on method and on a class level. Using `@Transactional` on a class level is recommended only if all the methods in your class should be transactional.
- Queries that read from the database do not need to be transactional, hence they do not require the `@Transactional` annotation. You need to use `@Transactional` only on methods, that induce changes on the database or when you need a high level of consistency when reading data to make sure nobody is changing it while you read it.

To make sure that our author exists, let's create a query to get it from the database.

But first, let's define some constructors:

```
public abstract class UserDTO {

    //definitions

    public UserDTO() { ①
    }

    public UserDTO(User user) {
        userName = user.userName;
        password = user.password;
        firstName = user.firstName;
        lastName = user.lastName;
```

```
        email = user.email;
    }

    //getters and setters
}

public class AuthorDTO extends UserDTO {

    //definitions

    public AuthorDTO() { ①
    }

    public AuthorDTO(Author author) {
        super(author);
        isRegular = author.isRegular;
        salary = author.salary;
    }

    //getters and setters
}
```

① Just like JPA, JSON-B uses reflection to convert JSON into POJO and backwards. For that reason access to a default constructor is needed.

And here is the definition of our `getAuthor` method.

```
@Path("/user/author")
public class AuthorResource {

    EntityManager entityManager;

    public AuthorResource(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    public Response createAuthor(AuthorDTO authorDTO) { ... }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAuthor(@PathParam("id") Long authorId) {
        if (authorId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            Author author = entityManager.find(Author.class, authorId);
            if (author == null) {
                return Response.status(Response.Status.NOT_FOUND).build();
            } else {
                return Response.ok(new AuthorDTO(author)).build();
```

```
            }
        }
    }
}
```

And if we try to call this endpoint, we should be able to get our author.

```
curl -i --location --request GET 'http://localhost:8080/user/author/1'
HTTP/1.1 200 OK
Content-Type: application/json
content-length: 147

{"email":"cj@apperture-
science.io","firstName":"Cave","lastName":"Johnson","password":"cj_pass","userName":"c
j_123","regular":false,"salary":10000}
```

5.6.4. Upgrading the `ArticleResource`

Now that we are ready with the author, let's integrate it with the article. First off, it's time for some changes:

1. Add constructors to the `Article` entity

```
@Entity
public class Article extends AbstractEntity {

    // definitions

    public Article() {
    }

    public Article(String title, String content, LocalDate publishDate, Author
author) { ①
        this.title = title;
        this.content = content;
        this.publishDate = publishDate;
        this.author = author;
    }
}
```

① For now we will decide not to assign comments to the article.

2. Next step is changing `author` to `authorId` inside the `ArticleDTO`, and adding constructors as well.

```
public class ArticleDTO {

    // definitions
```

```
    public ArticleDTO() {
    }

    public ArticleDTO(Article article) {
        this.id = article.id;
        this.title = article.title;
        this.content = article.content;
        this.publishDate = article.publishDate.toString();
        this.authorId = article.author.id;
    }

    // getters and setters
}
```

3. Now that we're done with that, it's time to refactor `ArticleResource` class method by method.

5.6.5. Creating an article

Let's see how we can apply our knowledge from creating an author into creating an article.

First we need to introduce the `EntityManager` to the class:

```
...
    private EntityManager entityManager;

    public ArticleResource(EntityManager entityManager) {
        this.entityManager = entityManager;
    }
...
```

Then we refactor the `createArticle` method:

```
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Transactional ①
    public Response createArticle(ArticleDTO articleDTO) {
        Author author = entityManager.find(Author.class, articleDTO.getAuthorId());
        if (author == null) {   ②
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            Article article = new Article(articleDTO.getTitle(), articleDTO.
getContent(),
                    LocalDate.parse(articleDTO.getPublishDate()),author);
            entityManager.persist(article);

            return Response.created(URI.create(String.format("article/%s", article.id
))).build();
        }
```

```
    }
```

① Don't forget that this action is `@Transactional`

② It is important to make sure that the article has an author to avoid any relational errors.

This should be enough to create articles. Now the call for creating an article would change to:

```
curl -i --location --request POST 'http://localhost:8080/article/' \
--header 'Content-Type: application/json' \
--data-raw '{
    "title": "Article for the soul.",
    "content": "The quick brown fox runs over the lazy dog.",
    "publishDate": "2022-01-12",
    "authorId": 1
}'
```

Other than that, everything should remain the same.

5.6.6. Getting the article

If you try to call the `getArticle` endpoint, though you will be met with `404` response. This is because we're now saving the articles to the database, not within the map. So let's get articles from our database instead.

```
    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getArticle(@PathParam("id") Long articleId) {
        if (articleId < 1>) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        Article article = entityManager.find(Article.class, articleId);

        if(article == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        } else {
            return Response.ok(new ArticleDTO(article)).build();
        }
    }
```

Mentioning again, getting an article, does not induce any changes to the database so `@Transactional` here is not needed. Query this endpoint again, and you should be seeing the results.

5.6.7. Getting a list of articles (Pagination)

Since we are in the getting part, let's see what kind of options we have when it comes to getting all

articles by pages.

A paginated query to the entity manager would look like that:

```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getArticles(@QueryParam("page") @DefaultValue("1") int page,
                                @QueryParam("size") @DefaultValue("10") int size) {
        if (page < 1 || size < 0) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            List<Article> articles = entityManager.createQuery("select a from Article
a", Article.class) ①
                    .setFirstResult((page - 1) * size) ②
                    .setMaxResults(size) ③
                    .getResultList();

            List<ArticleDTO> articleDTOs = articles.stream()
                    .map(ArticleDTO::new)
                    .collect(Collectors.toList());

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

① The create query method will take an JPQL language query and convert it to the specific language the database works with. Since the entity manager does not have a method to request a list of all entities out of the box, we need to specifically query that.

Now if we try and call the endpoint, we are going to get the same result, like we did when we used tha map.

But let's not just limit ourselves with just getting all the articles. What if our users want to see the articles of a specific author?

It's time to touch our `getArticles` even a bit more and make it smarter. Why don't you try and modify it yourself?

Here are some tips: . Add a new query parameter to the endpoint, called `authorId` . Check if that id is valid - if the id is null, get all articles, if the id is not null, create a query with a where clause...

Example 1. Using the `where` clause with entity manager.

Using the `entityManager.createQuery()` method we can define any query that we want, but what is the proper way to define a query like that:

```
select a from Articles where a.author.id = 1
```

First thing that would come to the untrained mind is to use the `String.format()` method, or any

type of concatenation where the variable will be. But there is a more lean way to do that.

With the `setParameter()` method, you can neatly set parameters to your query. In order to use the `setParameter` method, we need to define our query like this:

```
select a from Article a where a.author.id = :authorId
```

or if we do not want to give the parameter a name

```
select a from Article a where a.author.id = ?1
```

Then in Java we would use the set parameter as follows:

```
List<Article> articles = entityManager.createQuery("select a from Article a where
a.author.id = :authorId")
    .setParameter("authorId", authorId)
    .getResultList();

// or in the case of param number

List<Article> articles = entityManager.createQuery("select a from Article a where
a.author.id = :authorId")
    .setParameter(1, authorId)
    .getResultList();
```

Using this knowledge, now you can properly define your query for getting articles by author.

Now that you are able to query articles by authors, you have a more real-life endpoint that is suitable to work with.

Adding `insert.sql` script

All's good, but don't you get tired from creating authors and articles every time you need to test something? Let's change that a bit.

Add a file, called `import.sql` into `main/resources` folder. And add the following script:

```
insert into Users (version, email, firstName, lastName, password, userName,
isRegular, salary, DTYPE, id)
values (0, 'cj@apperture-science.io', 'Cave', 'Johnson', 'cj_pass', 'cj_123',
true, 10000, 'Author', 1),
       (0, 'jane@apperture-science.io', 'Jane', 'Doe', 'jd_pass', 'jd_123', true,
10000, 'Author', 2);
```

```
insert into ARTICLE (ID, VERSION, CONTENT, PUBLISHDATE, TITLE, AUTHOR_ID)
values (3, 0, 'The quick brown fox runs over the lazy dog.', '2022-01-12',
'Article for the soul.', 1),
       (4, 0, 'This is an article by the same author, who created Ipsum Lorem',
'2022-02-12', 'The aitor that created',
        1),
       (5, 0, 'This is how I got my hands into Java long time ago. Long article
here...', '2020-01-10',
        'The way I became Java developer', 2),
       (6, 0, 'This is my extreme enjoyment of Quarkus, written in an article',
'2022-09-13',
        'I love Quarkus and Quarkus loves me back', 2);


alter sequence USERS_SEQ restart with 10;
alter sequence ARTICLE_SEQ restart with 10;
alter sequence COMMENT_SEQ restart with 10;
alter sequence ADVERTISER_SEQ restart with 10;
```

Now every time the application has started, reloaded it will insert those entries, so the database will always be filled with some data.

5.6.8. Updating an article

Continuing out progress, let's see how an article is updated.

```
    @PUT
    @Path("/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    @Transactional
    public Response editArticle(@PathParam("id") Long articleId, ArticleDTO
articleDTO) {
        if (articleId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        Article article = entityManager.find(Article.class, articleId);

        if (article == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        } else {
            article.content = articleDTO.getContent();
            article.publishDate = LocalDate.parse(articleDTO.getPublishDate());
            article.title = articleDTO.getTitle();

            entityManager.merge(article); ①
            return Response.ok(new ArticleDTO(article)).build();
        }
```

```
    }
```

① Usage of `entityManager.merge()` here is optional. As long as the invocation of the `find` method has been made within a `@Transactional` method, you needn't call `entityManager.merge()`. When the transaction is finished, all the modifications on the article class will be automatically persisted.

`persist()` vs `merge()`

Getting an article object doesn't always mean that you get an article entity. This is because the objects have different states when JPA is regarded. The below figure shows how this works:

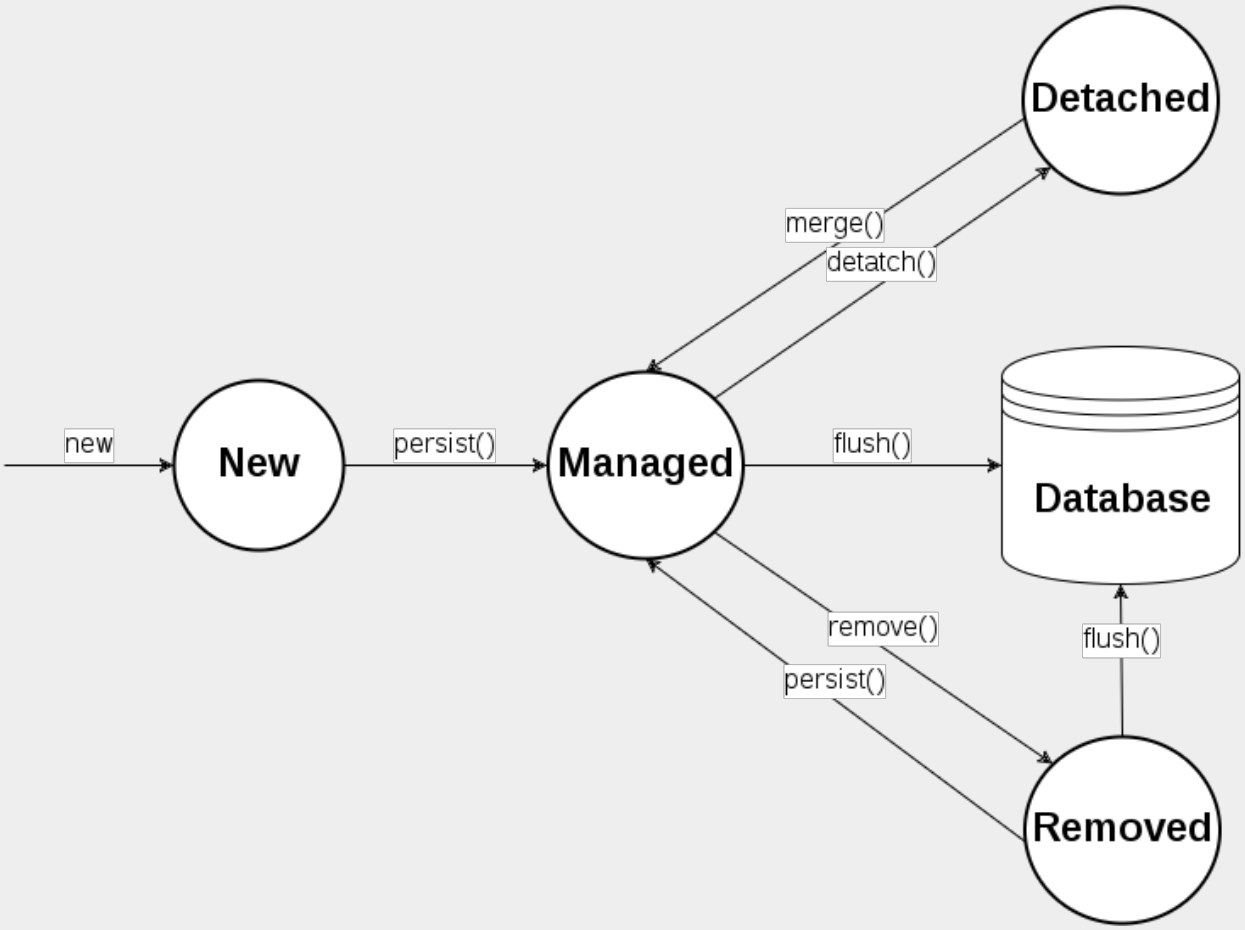


Figure 1. Entity state diagram

When calling the `persist()` on an entity class, the entity manager, will always try to create a new entity. The entity then is tracked by the entity manager (merged) and all the changes done to it, within the `@Transactional` context will be executed upon transaction flushing.

If the entity is detached, when the instance of an entity is passed outside a transactional context for example, you will need to call `entityManager.merge()` in order to connect it to an entity inside the database. The method will look for the entity by its id and apply all the changes to that entity. This method should be used only on already existing entities.

Due to the way our endpoints are constructed, having to merge an entity is a rare occasion, but not unfeasible. If you happen to stumble upon a situation, where you need to call the `merge` method, and you didn't you'll probably get an exception, stating that the entity has not been merged to be persisted.

Then you'll need to refactor your implementation and invoke `merge()` where needed.

5.6.9. Deleting an article

Moving forward with our last CRUD operation - deleting an article. One way to define it is like that:

```
@DELETE
@Path("/{id}")
@Transactional
public void deleteArticle(@PathParam("id") Long articleId) {
    if (articleId != null && articleId >= 1) {
        Article article = entityManager.find(Article.class, articleId);
        entityManager.remove(article);
    }
}
```

Although there is a problem here. Instead of making one query, we are making two. First we select an article, that might be existing in the database, then we invoke delete on it.

Why don't you try to create a query for deleting the article with a single query? This is your task.

And here are some hints:

1. Use what you already know about creating custom queries
2. To execute your query, invoke the method, called `executeUpdate()`

5.6.10. Optimizing our queries

Using the `entityManager.createQuery()` method, might add some delay to our queries. In our scenario we don't see that, because we are working with small amount of data, but in a broad scenario, it would take some time to interpret the query and translate it to the native SQL language for the regarded database. Imagine putting this in a loop, and every time the method is invoked, it has to interpret your query again and again. It's impractical.

Using this method is only recommended when the query is quite dynamic. In our case all of our queries are mostly static. The only difference is a variable here and there. Thankfully JPA offers another more optimized approach for static queries. Presenting "**Named queries**". Named queries are queries, which are specified on class level and are interpreted at the startup of the application. So every time you create a query, you are going to use its native language under the hood.

Let's now go to our `Article` class and create a couple of named queries, that we use for our endpoints.

```
@NamedQuery(name = Article.GET_ALL_ARTICLES, query = "select a from Article a") ①
@NamedQuery(name = Article.GET_ALL_ARTICLES_FROM_AUTHOR, query = "select a from
Article a where a.author.id = :authorId")
```

```
@NamedQuery(name = Article.DELETE_ARTICLE, query = "delete from Article a where a.id =
:articleId")
public class Article extends AbstractEntity {

    //It is recommended to use constants, so if the name changes, it should be only in
one place.
    public static final String GET_ALL_ARTICLES = "Article.getAllArticles"; ②
    public static final String GET_ALL_ARTICLES_FROM_AUTHOR =
"Article.getAllArticlesFromAuthor";
    public static final String DELETE_ARTICLE = "Article.deleteArticle";

    //rest of the code
}
```

① With the `@NamedQuery` annotation, we are telling JPA/Hibernate, to interpret a query for that JPQL query at server startup.

② Names of the queries should be unique, as it does not matter where the annotation is placed. You can literally place it on a different class, and it will work without any problems. The only way for JPA to find your query is by its unique name.

Now let's replace all occurrences of `entityManager.createQuery()`, with `entityManager.createNamedQuery()` in our `ArticleResource`.

Try and do it yourself. Use the constants for passing the name of the query.

5.6.11. Conclusion

As with JAX-RS that's not all. There is more to learn for the JPA spec, and we are going to give you more examples along this course. But never stop being curious! Go onto the internet, play with the code we have created and see how it all works.

Our next step is to see how Quarkus Panache comes in action and simplify those queries even further.

Chapter 6. Quarkus Panache and Active record pattern

The next step of our JPA journey is to upgrade our knowledge and add something Quarkus to the mix. One awesome feature that comes with Quarkus is their Panache API. Panache is a programming interface that adds extensions to the entities and allows the Active record pattern, where instead of using the `EntityManager` for managing your entities, you can directly call methods such as `find()`, `persist()`, `delete()`. This would make your code look cleaner, better to understand and easier to work with.

To prove that, let's see it in action.

But first...

6.1. Extensions, extensions, extensions

If you are familiar with Maven, you can avoid the steps below, by just opening the `pom.xml` file and adding `-panache` to `<artifactId>quarkus-hibernate-orm</artifactId>`.

Using Panache is optional for the project. You may prefer the old ways and use the entity manager with some DAO (Data Access Object) layerings to call the database, but in order to use Panache, you'll need to add the extension for it.

To add the panache extension, simply execute the following line on your project's folder:

```
mvn quarkus:add-extension -Dextensions="io.quarkus:quarkus-hibernate-orm-panache"
```

And since this extension comes with hibernate, we can remove the previous `quarkus-hibernate-orm` extension by executing

```
mvn quarkus:remove-extension -Dextensions="io.quarkus:quarkus-hibernate-orm"
```

Now if we look at the dependencies inside the `pom.xml`, we should be left only with this:

```
    <dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-hibernate-orm-panache</artifactId>
    </dependency>
```

6.2. The `PanacheEntityBase` class

Converting an entity to an active record is very easy. In our case, even easier. What you need to do

is simply extend the entity class with PanacheEntityBase, and ta-da! You have an acive record entity.

```
public abstract class AbstractEntity extends PanacheEntityBase {...}
```

Extending AbstractEntity with the PanacheEntityBase class allowes active record for all of our entities.

Panache also offeres the PanacheEntity class, but here we would reather not use it as it has its own definition of the entity id, and we want to define it as our needs or database requires it.

Now that we have extended our AbstractEntity, we can go to our AuthorResource and edit it to support Active record.

First, let's get rid of the EntityManager initialization. Remove the constructor and the definition of EntityManager.

Then, let's touch our methods a bit...

```
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Transactional
    public Response createAuthor(AuthorDTO authorDTO) {

        Author author = new Author(authorDTO.getUserName(), authorDTO.getPassword(),
authorDTO.getFirstName(),
                authorDTO.getLastName(), authorDTO.getEmail(), authorDTO.isRegular(),
authorDTO.getSalary());
        author.persist(); ①

        return Response.created(URI.create(String.format("/user/author/%d",author.id)
)).build();
    }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAuthor(@PathParam("id") Long authorId) {
        if (authorId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            Author author = Author.findById(authorId); ②
            if (author == null) {
                return Response.status(Response.Status.NOT_FOUND).build();
            } else {
                return Response.ok(new AuthorDTO(author)).build();
            }
        }
```

```
    }
```

① The `persist()` method will either create or update entity. You can use it with existing entities or new entities. The option to update the entity without calling `persist()` stays the same, as long as the entity is attached to the transactional scope.

② The `findById()` method will obviously search for that entity by its id and return it, or return null if it is not present.

Instead of using `findById()`, you can use `findByIdOptional()` and simplify your `getAuthor` method to look like this:

```
    public Response getAuthor(@PathParam("id") Long authorId) {
        if (authorId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build(
);
        } else {
            return Author.<Author>findByIdOptional(authorId) ①
                    .map(author -> Response.ok(new AuthorDTO(author))
.build())
                    .orElseGet(() -> Response.status(Response.Status
.NOT_FOUND).build());
        }
    }
```

① The generic find methods from `PanacheEntity` class are not able to always recognize what kind of return type you expect at the end. Therefor sometimes explicitly mentioning the class is needed.

Alright! The easy part has been completed. Let's move to the author resource and do smoe changes there too.

6.3. Using Panache in `ArticleResource`

Let's see how we can use panache for every other endpoint in our `ArticleResource`.

6.3.1. Creating an article:

Here, just like creating an author, we can simply replace the `entityManager.persist()`, with `article.persist()`

```
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Transactional
    public Response createArticle(ArticleDTO articleDTO) {
        Author author = Author.findById(articleDTO.getAuthorId());
        if (author == null) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
```

```
            Article article = new Article(articleDTO.getTitle(), articleDTO.
getContent(),
                    LocalDate.parse(articleDTO.getPublishDate()), author);
            article.persist();
            return Response.created(URI.create(String.format("article/%s", article.id
))).build();
        }
    }
```

6.3.2. Updating an article

To update an existing article we can litterally do the same, or even better. As mentioned in our JPA chapter, we don't need to use `.persist()` at all.

```
    public Response editArticle(@PathParam("id") Long articleId, ArticleDTO
articleDTO) {
        if (articleId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        Article article = Article.findById(articleId);

        if (article == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        } else {
            article.content = articleDTO.getContent();
            article.publishDate = LocalDate.parse(articleDTO.getPublishDate());
            article.title = articleDTO.getTitle();

            return Response.ok(new ArticleDTO(article)).build();
        }
    }
```

6.3.3. Removing an article

Removing an article is also very simple, you don't even have to define a speciffic query for it. You need to simply invoke the `delete()` method.

```
    public void deleteArticle(@PathParam("id") Long articleId) {
        Article.delete("id", articleId); ①
    }
```

① In the first argument of the delete method you can either place a whole JPQL query or define only the properties, you are searching by.

6.3.4. Getting an article

Here comes the fun part. In fact why don't you try to edit the `getArticle()` method to support the Panache active record aproach, while we explain the other queries from `getArticles()`?

Looking at `getArticles()` method, reworking it would look like this:

```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getArticles(@QueryParam("page") @DefaultValue("1") int page,
                                @QueryParam("size") @DefaultValue("10") int size,
                                @QueryParam("author") Long authorId) {
        if (page < 1 || size < 0 || authorId != null && authorId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            String query = authorId == null ? Article.GET_ALL_ARTICLES : Article
.GET_ALL_ARTICLES_FROM_AUTHOR;
            List<Article> articles = (authorId == null ? Article.findAll() :
                    Article.find("#" + query, Map.of("authorId", authorId))) ①
                            .page(page - 1, size) ②
                            .list();

            return Response.ok(articles.stream().map(ArticleDTO::new).collect
(Collectors.toList())).build();
        }
    }
```

① Referring to a `NamedQuery` in Panache would require adding a `#` to the sting.

② The page and size is defined through the `page()` method, which will accutatrly then calculate how many results to pull from the database. As always pages start with a zero-based index, so you need to subtract 1 from the inputted content.

There is also another more preferred way. In fact most of where queries do not require creating a named query when using panache. Quarkus has optimized that for you, so you can change the query to:

```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getArticles(@QueryParam("page") @DefaultValue("1") int page,
                                @QueryParam("size") @DefaultValue("10") int size,
                                @QueryParam("author") Long authorId) {
        if (page < 1 || size < 0 || authorId != null && authorId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            List<Article> articles = (authorId == null ? Article.findAll() :
                    Article.find("author.id = ?1", authorId)) ①
                            .page(page - 1, size)
                            .list();
```

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

① Instead of using named queries, for a simple query like that, you could just put your where clause checks inside the first argument of `find()`

Now with all that that you are not using any NamedQueruies, you can go and get rid of them. Also it is safe to remove the `EntityManager`, as you won't need it for now.

6.4. Conclusion

And we're done. This is all the basics you need to know using Panache with JPA. With our next section we are going to integrate both JPA and Panache into deeper operations, so you can find out more tricks and techniques using tohose APIs.

Chapter 7. JPA. Going deep

In this chapter we are going to look at more sophisticaded queries so we can better understand how to get some things done using JPA.

7.1. Adding comments to an article. The `one-to-many` relationship

The beginning of our journey will start with our one-to-many relationship `Article → Comment`. For a start, let's extend our `import.sql` by adding a few subscribers.

```
insert into Users (version, email, firstName, lastName, password, userName,
streetAddress, subscribedUntil,
                   creditCardType, number, DTYPE, id)
values (0, 'sarah@google.space', 'Sarah', 'Connor', 'sarAPass', 'sarah_9645',
'Hamburger Str.', '2024-01-12', 'VISA',
        '3698521479456746', 'Subscriber', 7),
       (0, 'peter@linked.io', 'Peter', 'Blanca', 'pb&^%', 'peter_998', 'Mustard str.',
'2024-01-13', 'MASTER_CARD',
        '8774662321', 'Subscriber', 8),
       (0, 'kchuck@pongo.eu', 'Chuck', 'Keith', 'chUk', 'chuck_0998', 'Sausage Str.',
'2024-01-14', 'AMERICAN_EXPRESS',
        '3698521479', 'Subscriber', 9);

alter sequence USERS_SEQ restart with 10;
alter sequence ARTICLE_SEQ restart with 10;
alter sequence COMMENT_SEQ restart with 10;
alter sequence ADVERTISER_SEQ restart with 10;
```

Having that set, let's create our endpoint for adding comments in a new resource, called `CommentResource`

1. Create the class `magman.comment.CommentResource` and assign the path `/article/{id}/comment.
2. Create the class `CommentDTO` in the same package and add all required fields, getters/setters and constructors.
3. Add a list of comments to the `ArticleDTO`.
4. Add a POST endpoint method called `createComment()` to the comment resource. Decorate it with all required annotations needed to access it.

At the end our endpoint should look like this:

```
    @POST
    @Transactional
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createComment(@PathParam("id") Long articleId, CommentDTO
commentDTO) {
```

```
        if (articleId < 1 || commentDTO.getAuthorId() == null || commentDTO
.getAuthorId() < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        User author = User.findById(commentDTO.getAuthorId()); ①
        if (author == null) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        Article article = Article.findById(articleId);
        if (article == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        }

        Comment comment = new Comment(commentDTO.getContent(), author, LocalDateTime
.now());

        article.comments.add(comment); ②
        comment.persist(); ③

        return Response.created(URI.create(String.format("/article/%d/comment/%d",
articleId, comment))).build();
    }
```

① Any user could type be an author of a comment. Since we are not interested into the additional attrobutes of the type of user, we can simply query with the `User` entity. Here we are also getting the `userId` from the comment as we still don't have mechanisms to get the user from a session.

② When we have `@OneToMany` relationship between an article and a comment, we first need to add the comment to the article's comments list.

③ Before we return a response, we first need to persist the comment.

In this example we do not need to add `article.persist()`, because as we mentioned before during a transaction context, modifications on **attached** entities will be applied automatically.

Now when we have added comment to the article, invoking that article will return its comment also. Here is an example query:

```
curl --location --request POST 'localhost:8080/article/3/comment' \
--header 'Content-Type: application/json' \
--data-raw '{
    "content": "The quick brown fox jumps over the lazy dog.",
    "authorId": 7
}'
```

All that we need for now in the query is the content and the user's id. In the future we are only going to need the content, as all other data will be set by the web

> server, as it is not a concern for the client.

Perfect! But have you ever seen a comment section of any type of content? Sometimes they might be thousands, depending on how viral the conent is. This is the reason why we don't want to get all entities in one-to-many relationship entity. We want to get them in a moderate fashion, give the user as many as they would probably read. To show off the different capabilities of JPA, let's do some modifications...

7.2. Separating the comments from the article, but keeping them intact

We do not want to load all the comments when querying articles. So let's do something about it.

First we can remove the list of comments from the `Article` entity. Remove the list, remove the `@OneToMany` annotation.
Then we might be faing the issue that our `ArticleDTO` constructor is not receiving comments. Here you can simply remove the initialization of comments from the constructor or wherever you defined comments for article in your code, but leave the peopery and getters/setters fro `CommentDTO`. As for adding a comment to an article, set the article to the comment.

```
 public Response createComment(@PathParam("id") Long articleId, CommentDTO commentDTO)
{
    // Validation checks

    Comment comment = new Comment(commentDTO.getContent(), author, LocalDateTime.now(
));
    comment.article = article;
    comment.persist();

    return Response.created(URI.create(String.format("/article/%d/comment/%d",
articleId, comment.id))).build();
}

@Entity
public class Comment extends AbstractEntity {

    //Definitions

    @ManyToOne
    public Article article;

    //constructors
}
```

Now when you query to get an article, you won't be able to see any comments.

7.2.1. Getting list of speciffic columns in s query

Let's say that we don't want to get the comments of an article, when we query a list of articles, but we want those comments when we want a single article.

There's two approaches we can look throigh here:

- Create a separate query and assign those values additionally
- Create a composite query and then assign the values ourselves

Let's look at the second one as it is complex more. We suggest that you know what happens in the first option.

For a start, let's add a query parameter to the `getArticle()` method, called `withComments`.

```
public Response getArticle(@PathParam("id") Long articleId, @QueryParam("withComments
") boolean withComments) {
    //implementation
 }
```

Next, let's create a named query that will get the article with comments:

```
select a, c from Article a left join Comment c on c.article = a where a.id =
:articleId
```

And at the end our `getArticle` implementation will look like this

```
    public Response getArticle(@PathParam("id") Long articleId, @QueryParam(
"withComments") boolean withComments) {
        if (articleId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        Optional<Article> article = Optional.empty();
        List<Comment> comments = new ArrayList<>();
        if (withComments) {
            List<Object[]> articleWithComments = Article.getEntityManager() ①
                    .createNamedQuery(Article.GET_ARTICLE_WITH_COMMENTS, Object[]
.class)
                    .setParameter("articleId", articleId)
                    .getResultList();

            if (!articleWithComments.isEmpty()) {
                article = Optional.of((Article) articleWithComments.get(0)[0]);
                comments = new ArrayList<>();
                for (var set : articleWithComments) { ②
                    if (set[1] != null) {
                        comments.add((Comment) set[1]);
```

```
                    }
                }
            }
        } else {
            article = Article.findByIdOptional(articleId);
        }

        if (article.isPresent()) {
            ArticleDTO articleDTO = article.map(ArticleDTO::new).get();
            List<CommentDTO> commentDTOS = comments.stream().map(CommentDTO::new)
.collect(Collectors.toList());
            articleDTO.setComments(commentDTOS);
            return Response.ok(articleDTO).build();
        } else {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
    }
```

① When we ask for more than one object in a query, the entity manager will return array of objetcs. We have to then manually cast those objects into whatever type we need.

② The result of our query will give us a list of the same article with different comment. Since we want to hava a single article with a list of comments, we have to loop through the results and join them as single list of comments.

There is even more neat way to avoid all that casting. With JQL we can add java objects to the query. To do so, let's create a wrapping object in the `model` package, called `ArticleWithComment`.

7.2.2. Inserting additional objects to a JQL query

Our `ArticleWithComment` entity should look like this:

```
public class ArticleWithComment {

    public Article article;
    public Comment comment;

    public ArticleWithComment(Article article, Comment comment) {
        this.article = article;
        this.comment = comment;
    }
}
```

After we have implemented it, we need to upgrade our JQL query to support it.

```
select new com.vidasoft.magman.model.ArticleWithComment(a, c) from Article a left join
Comment c on c.article = a where a.id = :articleId
```

And now implement it in our resource method:

```
//checks
if (withComments) {
    List<ArticleWithComment> articleWithComments = Article.getEntityManager()
            .createNamedQuery(Article.GET_ARTICLE_WITH_COMMENTS, ArticleWithComment
.class)
            .setParameter("articleId", articleId)
            .getResultList();

    if (!articleWithComments.isEmpty()) {
        article = Optional.of(articleWithComments.get(0).article);
        comments = new ArrayList<>();
        for (var set : articleWithComments) {
            if (set.comment != null) {
                comments.add(set.comment);
            }
        }
    }
}

//returns
```

Unfortunately with the way JPA has been designed and the complexity of our query, we are yet not able to get a list of comments, so we have to create it manually.

7.2.3. Creating native queries with JPA

Sometimes JPA can't perform queries speciffic for the database. Then you will need to create a native query that is capable to perform the operations you need.

To demonstrate that, let's implement an endpoint for getting a comment by its id. But this time, instead of using JQL or panache, we are going to do it with a native query.

First off, create a `@NativeNamedQuery` to get the comment by its id:

```
@Entity
@NamedNativeQuery(name = Comment.GET_COMMENT_BY_ID,
query = "select ID, CONTENT, AUTHOR_ID, CREATED from COMMENT where id = :commentId")
public class Comment extends AbstractEntity {
    //implementation
}
```

Then to simplify things, add a constructor to the `CommentDTO` that takes all properties.

```
public class CommentDTO {

    //properties
```

```
    public CommentDTO(Long id, String content, Long authorId, String created) {
        this.id = id;
        this.content = content;
        this.authorId = authorId;
        this.created = created;
    }

    //more constructors, getters and setters
}
```

And finally, let's implement our `getComment` endpoint:

```
    @GET
    @Path("/{commentId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getCommentById(@PathParam("commentId") Long commentId) {
        if (commentId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            Object[] commentResult = (Object[]) Comment.getEntityManager()
.createNamedQuery(Comment.GET_COMMENT_BY_ID) ①
                    .setParameter("commentId", commentId)
                    .getResultStream()
                    .findFirst()
                    .orElse(null);
            if (commentResult == null) {
                return Response.status(Response.Status.NOT_FOUND).build();
            } else {
                CommentDTO comment = new CommentDTO(    ②
                        ((BigInteger) commentResult[0]).longValue(), //id ③
                        (String) commentResult[1], //content
                        ((BigInteger) commentResult[2]).longValue(), //author id
                        commentResult[3].toString() // created
                );
                return Response.ok(comment).build();
            }
        }
    }
```

① Just like with composite queries, what we get in return here is an array of `Object[]`, which we have to cast into the types we require.

② The order in which the result array is constructed is the same as the one in our query. This is how you are supposed to know which value is which.

③ Sometimes the object that JPA decides to use for the types in our database for native queries may differ from what we actually want, so we need to address that, by additional casting.

You can argue that this doesn't look that neat, right? Thankfully there is another way to save

ourselves from all that casting.

7.2.4. The `@SqlResultMapping` annotation

In order to make that result mapping automatic by JPA, there is also an annotation that is going to help us out. The usage of this annotation is not going to work with all scenarios, so this is the reason why we are left with the option to do the casting ourselves. Let's add this annotation above our `NamedNativeQuery`.

```
@Entity
@SqlResultSetMapping(name = Comment.GET_COMMENT_BY_ID, ①
        classes = {
                @ConstructorResult(targetClass = CommentDTO.class, columns = { ②
                        @ColumnResult(name = "ID", type = Long.class),
                        @ColumnResult(name = "CONTENT"),
                        @ColumnResult(name = "AUTHOR_ID", type = Long.class),
                        @ColumnResult(name = "CREATED", type = String.class)
                })
        })
@NamedNativeQuery(name = Comment.GET_COMMENT_BY_ID,
        query = "select ID, CONTENT, AUTHOR_ID, CREATED from COMMENT where id =
:commentId",
        resultSetMapping = Comment.GET_COMMENT_BY_ID) ③
public class Comment extends AbstractEntity {
    //implementation
}
```

① Here we can use the same name as the name of the query

② The order in which the variables are assigned is the same in which the `CommentDTO` constructor was previously defined.

③ In the native query we place the name of the mapper in `resultSetMapping`

Now when we go back to our `CommentResource` our `getComment` implementation may look like this:

```
    @GET
    @Path("/{commentId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getCommentById(@PathParam("commentId") Long commentId) {
        if (commentId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            CommentDTO comment = Comment.getEntityManager().createNamedQuery(Comment
.GET_COMMENT_BY_ID, CommentDTO.class)
                    .setParameter("commentId", commentId)
                    .getResultStream()
                    .findFirst()
                    .orElse(null);
            if (comment == null) {
```

```
                return Response.status(Response.Status.NOT_FOUND).build();
            } else {
                return Response.ok(comment).build();
            }
        }
    }
```

Native queries are suitable when you use the same database through all your environments. This means what you should use the same database even for testing, as you cannot test a native query in the same SQL dialect for different databases, unless they have the same dialect.

7.3. Going even deeper

There's even more stuff to look at. Unfortunately the scope of our project will not allow us to show appropriate examples. But this doesn't stop you to go online and look for some JPA tips and tricks for yourself. Here are a couple of topics to begin with:

- Many-to-many relationships
- One-to-one relationships
- Ellement collections

All the articles have been written by top developers, who have activerly contributed for and stay behind Hibernate, Eclipse and the JPA specifications. You can always trust and use their guides as foundations of project's data model.

Chapter 8. Dependency injection with Quarkus and Jakarta EE

One of the most relevant and long-lasting concepts in object-oriented programming, embedded into the foundation of all web server frameworks is the Dependency injection. These techniques also allow to achieve the so called "inversion of control" (IoC for short), where the chain of classes, depending on one another is reversed and decoupled. This allows the developer to change the implementation of one module with another, without having to retouch the whole chain of dependencies, but it also helps instantiating dependencies, without concerning about the initialization, such as constructors, variables and other related dependencies. A way to achieve the inversion of control is having an IoC container. The IoC container represents an API that stores, manages and controls instantiations of dependencies. One such container is the Jakarta EE CDI.

There is a very good example on what dependency injection is, much easier to understand. Imagine your class is a small child, who wants to take something out from the upper shelf of a fridge. They can either do it themselves and risk making a mess, like dropping and breaking stuff, or they can ask its parent (IoC container), to give it to them. The parent has the knowledge, height and strength to be capable to know what is good for the child, whether they should have that item, or a substitute and pass the item to the child safely.

8.1. What is CDI

As mentioned above CDI (Context and Dependency Injection) is one of the specifications of Jakarta EE, that is capable of managing dependencies within your application using scopes (we'll talk about them in a bit). Instead of using constructors to create instances and connections between the modules of your application, you can rely on the CDI, to provide your classes with the instance they need.

During any application development one huge role, that plays into good practices is the separation of concerns. This means that a class or a method should do exactly the thing they are supposed to do, without any additional jobs, which we call side effects. So for example a Resource class is supposed to accept requests, validate queries and return responses, but not look into connecting with the database, building a query or executing complex operations directly on the database, such as performing multiple queries at once. This is business logic, that is suitable to live on a separate layer in our server. This layer is usually called a service. The service class will do all the hard job regarding all additional operations out request has to perform, keeping our resource method clean from any additional distractions.

Layering within your application is subjective. You are able to do whatever you like with your code, with one rule on the thumb - keep it clear and consistent. So depending on the framework you are using, you have to figure out how to better structure your project to make it easy to understand and maintain.

If your business logic is doing more than just getting some data as is, than maybe this is suitable for a service. Depending on the framework you are using, you

> might need to also have a Data Access Object layer (DAO) or Repository layer where you keep all your queries to the database. With Quarkus and Panache those types of layers are not needed, as the Panache Active Record pattern already serves as such a layer.

In Quarkus, the CDI implementation is provided by Quarkus ArC. Let's see how we can apply CDI in action for our project and explain how everything works.

8.2. The Application and Request scopes

To begin, we are going to create the `UserResource` class, which is going to contain two methods:

- `registerUser`
- `loginUser`

```
@Path("user")
public class UserResource {

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response registerUser(NewUserDTO newUserDTO) {

    }

    @POST
    @Path("login")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response loginUser(LoginDTO login) {

    }

}

public class NewUserDTO extends UserDTO {

    private String password;
    private UserType userType;

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public UserType getUserType() {
        return userType;
    }
```

```
    public void setUserType(UserType userType) {
        this.userType = userType;
    }
}

public class LoginDTO {

    private String userName;
    private String password;

    public String getUserName() {
        return userName;
    }

    public void setUserName(String username) {
        this.userName = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}

public enum UserType {

    AUTHOR, SUBSCRIBER, MANAGER;

}
```

As you may notice `NewUserDTO` extends `UserDTO`, which used to contain the password property. This property was previously being used by the `AuthorDTO`, which also extends the `UserDTO`, in order to create the author. But now as we have the `NewUserDTO`, we need to move that field there.
We also don't need the `AuthorResource.createAuthor` method, as all user creations will be managed from `UserResource` from now on.

With the current knowledge we have, our implementation would look something like this:

```
@Path("user")
public class UserResource {

    @POST
    @Transactional
    @Consumes(MediaType.APPLICATION_JSON)
    public Response registerUser(NewUserDTO newUserDTO) {
        if (checkIfNull(newUserDTO.getUserName(), newUserDTO.getPassword(),
```

```
                newUserDTO.getFirstName(), newUserDTO.getLastName(), newUserDTO
.getUserType(), newUserDTO.getEmail())) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        Optional<User> existingUser = User.find("userName = ?1 or email = ?2",
newUserDTO.getUserName(),
                newUserDTO.getEmail()).firstResultOptional();
        if (existingUser.isPresent()) {
            return Response.status(Response.Status.CONFLICT).build();
        }

        User createdUser = null;
        switch (newUserDTO.getUserType()) {
            case AUTHOR:
                createdUser = new Author(newUserDTO.getUserName(), newUserDTO
.getPassword(), newUserDTO.getFirstName(),
                        newUserDTO.getLastName(), newUserDTO.getEmail(), true, 0);
                break;
            case MANAGER:
                createdUser = new Manager(newUserDTO.getUserName(), newUserDTO
.getPassword(), newUserDTO.getFirstName(),
                        newUserDTO.getLastName(), newUserDTO.getEmail());
                break;
            case SUBSCRIBER:
                createdUser = new Subscriber(newUserDTO.getUserName(), newUserDTO
.getPassword(), newUserDTO.getFirstName(),
                        newUserDTO.getLastName(), newUserDTO.getEmail(), null,
LocalDate.now().plusYears(1),
                        null);
                break;
        }

        createdUser.persist();
        return Response.status(Response.Status.CREATED).build();
    }

    @POST
    @Path("login")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response loginUser(LoginDTO login) {
        if (checkIfNull(login.getUsername(), login.getPassword())) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            User loggedUser = User.find("userName = ?1 and password = ?2", login
.getUsername(), login.getPassword())
                    .firstResult();
            if (loggedUser == null) {
                return Response.status(Response.Status.UNAUTHORIZED).build();
            } else {
                return Response.ok(new UserDTO(loggedUser)).build();
```

```
            }
        }
    }

    private boolean checkIfNull(Object... fields) {
        for (var field : fields) {
            if (field == null) {
                return true;
            }
        }

        return false;
    }

}
```

But from what you can see here, there is too much stuff in one place. We're doing data validations, database lookups, conditional user creations... Oh! And did you notice? We're storing the user's password IN PLAIN TEXT!

As mentioned earlier a resource class should do simple things. It should take data and return data. Any additional operations, are better to happen upon a separate layer. So let's create that layer, shall we?

Create a class in `user` package, called `UserService`. This class will contain all of our additional operations, regarding user registration and login.
Then move some of the code of `UserResource` to the user service.

```
public class UserService {

    public User registerUser(String firstName, String lastName, String email, String
username, String password, UserType userType) {
        User createdUser = null;
        switch (userType) {
            case AUTHOR:
                createdUser = new Author(username, password, firstName, lastName,
email, true, 0);
                break;
            case MANAGER:
                createdUser = new Manager(username, password, firstName, lastName,
email);
                break;
            case SUBSCRIBER:
                createdUser = new Subscriber(username, password, firstName, lastName,
email, null, LocalDate.now().plusYears(1), null);
                break;
        }

        createdUser.persist();
        return createdUser;
```

```
    }

    public Optional<User> loginUser(String username, String password) {
        return User.find("userName = ?1 and password = ?2", username, password)
.firstResultOptional();
    }

}
```

And now let's see how we can call this class from `UserResource`. Truth is, it's fairly simple.

1. Add `@ApplicationScoped` annotation on `UserService` class

```
@ApplicationScoped
public class UserService {...}
```

2. In `UserResource`, define `UserService` as a global property and add `@Inject` on that property.

```
@Path("user")
public class UserResource {

    @Inject
    UserService userService;

    @POST
    @Transactional
    @Consumes(MediaType.APPLICATION_JSON)
    public Response registerUser(NewUserDTO newUserDTO) { ... }

    @POST
    @Path("login")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response loginUser(LoginDTO login) { ... }
}
```

3. Now let's start using `userService` inside our code.

```
    @POST
    @Transactional
    @Consumes(MediaType.APPLICATION_JSON)
    public Response registerUser(NewUserDTO newUserDTO) {
        if (checkIfNull(newUserDTO.getUserName(), newUserDTO.getPassword(),
                newUserDTO.getFirstName(), newUserDTO.getLastName(), newUserDTO
.getUserType(), newUserDTO.getEmail())) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        Optional<User> existingUser = User.find("userName = ?1 or email = ?2",
```

```
newUserDTO.getUserName(),
                newUserDTO.getEmail()).firstResultOptional(); ①
        if (existingUser.isPresent()) {
            return Response.status(Response.Status.CONFLICT).build();
        }

        userService.registerUser(newUserDTO.getFirstName(), newUserDTO.getLastName
(), newUserDTO.getEmail(),
                newUserDTO.getUserName(), newUserDTO.getPassword(), newUserDTO
.getUserType()); ①
        return Response.status(Response.Status.CREATED).build();
    }

    @POST
    @Path("login")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response loginUser(LoginDTO login) {
        if (checkIfNull(login.getUsername(), login.getPassword())) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            Optional<User> loggedUser = userService.loginUser(login.getUsername(),
login.getPassword()); ①
            return loggedUser
                    .map(u -> Response.ok(new UserDTO(u)).build())
                    .orElseGet(() -> Response.status(Response.Status.UNAUTHORIZED)
.build());
        }
    }
```

① As you can see, we can refer to `UserService` like we would with any other instance of a class.

Scopes in CDI

In CDI every dependency, also called bean, has its own rate of instantiation and lifespan. This is called "scope". In vanilla CDI, the CDI, defined by Jakarta/Java EE, there are five scopes:

- `@RequestScoped` - upon every new request to the server a new instance of that class is created. When the response has been returned, the instance is destroyed. This allows multiple clients to request the same endpoint, with their own data and parameters, without interfering with each-other. Upon creations of these requests, the CDI implementation will decide how to manage the load and might create separate threads to run multiple requests simultaneously.
- `@ApplicationScoped` - one instance of the class is created for the whole lifespan of the server's runtime. Once an ApplicationScoped dependency has been injected into a class, it will create a single instance (singleton), which will be passed serverwide every time it's been injected, regardless of the scope of the depending class.
  For example, if the class is RequestScoped, each instance of that class will get the same instance of ApplicationScoped class. This means that the ApplicationScoped annotation is suitable for stateless classes, which are not subject to changes. It is not recommended to

define stateful properties into an ApplicationScoped class, such as user data.
Service classes are good example suitable for the `@ApplicationScoped` annotation. They mostly contain methods, which do the same job every time and we do not need more than one instance of such a class.

- `@SessionScoped` - those type of dependencies have active instance during the lifetime of a user session. When an user opens the page of a website for the first time, they will get a session cookie with item, called `JSESSIONID`. The `JSESSIONID` is an identifier which the client then passes to the server upon each request. With it CDI is capable of recognizing the user and binding their data with the `@SessionScoped` class instance. `@SessionScoped` will create as many instances, as active clients are on the server. It is suitable for storing user data, such as their email, permissions, preferences and so on.
- `@ConversationScoped` - this scope allows for the developer do control the creation and destruction of the instance. Followed with injection of `jakarta.enterprise.context.Conversation` and methods `conversation.begin()` and `conversation.end()`, the developer is able to decide when the instantiation should start and where it should end.

Both `@SessionScoped` and `@ConversationScoped` classes need to implement the `Serializable` interface, due to their long-lasting as those instances are stored int the user's HTTP session, which is sometimes saved as a file on the disk.

- `@Dependant` - this scope inherits the scope of the depending class. If you inject a Dependant class into `@RequestScoped` bean, CDI will create a new instance with Request scope, if you do it in `@ApplicationScoped` bean, it will do it with a single instance, same as the depending class.

The instances passed to each class, injecting those dependencies are proxies. In order to guarantee safety, you can never access and modify the real instance through reflection. This is important for a service that is meant to be active 24/7 and be robust.

Due to the way proxies are implemented, mixing scopes in some CDI implementations may, or may not work. For example injecting a `@RequestScoped` dependency into `@ApplicationScoped` bean on some servers may result in that instance being injected once for the whole lifespan of the `@ApplicationScoped` bean. This may cause issues, such as getting the same data on each new request, instead of getting newer data for that request.

In Quarkus ArC there are 3 scopes implemented and one non-proxied scope, respectively - `Request`, `Application`, `Dependant` and Singleton scopes. Dependencies, annotated with `@Singleton` act simmliarly to the `@ApplicationScoped`, with the difference that they are not proxied, meaning that the real instance of the entity is injected. This allows better performance in some occasions. Read more about `@Singleton` beans here.

Due to the fact, that Quarkus servers are meant to be stateless. There's no out-of-the-box support for `SessionScoped` and `ConversationScoped` annotations, meaning all the user data and configurations should come from outside in the form of a token (JWT for example).

Usually Resource endpoints should be annotated with some kind of scope in order for CDI to establish access to them. The most commonly used scope is `@RequestScoped`. With Quarkus ArC, though, resources are reconized and by default the `@Singleton` scope is given to them, unless you overide it explicitly, by adding your scope. This works only for endpoints, so if you want to inject any other dependency it must have at least `@Dependant` scope.

Going back to our project, we mentioned that we are saving the user with plain text password. As you may suggest this is very bad idea, so let's do something about it.

First add a new field to the user entity, called `salt`. Now let's create a new class and package `security.PasswordService`. You should be guessing what comes next...

```
@ApplicationScoped
public class PasswordService {

    public String encryptPassword(String password, String salt) {

    }

    public String generateSalt() {

    }

}
```

Just encrypting the password will not be enough to have a fully protected password. It will still be susceptible to "Rainbow attacks", which are a way to reverse engineer the password. Adding salt to the password, which is a random string of characters, then using encryption algorithm before saving the password to the database will make the password unbreakable. Even if somebody got the user's hashed password and the salt, they will have a hard time to figure out the password, as due to encryption, which produces a way different hash, than what encrypting a single password would do.

Having said that, the implementation of our password service, would look like this:

```
@ApplicationScoped
public class PasswordService {

    private static final String ENCRYPTION_KEY = "$oME$anD0mKey!@#";
    private static final int SALT_LENGTH = 8;

    public String encryptPassword(String password, String salt) {
        var saltedPassword = password + salt;

        Key aesKey = new SecretKeySpec(ENCRYPTION_KEY.getBytes(), "AES");
        try {
```

```
            Cipher cipher = Cipher.getInstance("AES");
            cipher.init(Cipher.ENCRYPT_MODE, aesKey);
            byte[] encrypted = cipher.doFinal(saltedPassword.getBytes());
            return new String(encrypted);
        } catch (Exception e) {
            return password;
        }
    }

    public String generateSalt() {
        StringBuilder sb = new StringBuilder();
        Random random = new Random();
        for (int i = 0; i < SALT_LENGTH; i++) {
            sb.append((char) random.nextInt());
        }

        return sb.toString();
    }

}
```

Now it's time to add that to our `UserService`.

```
@ApplicationScoped
public class UserService {

    @Inject
    PasswordService passwordService; ①

    public User registerUser(String firstName, String lastName, String email, String
username, String password, UserType userType) {
        User createdUser = null;
        switch (userType) {
            case AUTHOR:
                createdUser = new Author(username, password, firstName, lastName,
email, true, 0);
                break;
            case MANAGER:
                createdUser = new Manager(username, password, firstName, lastName,
email);
                break;
            case SUBSCRIBER:
                createdUser = new Subscriber(username, password, firstName, lastName,
email,
                        null, LocalDate.now().plusYears(1), null);
                break;
        }

        var salt = passwordService.generateSalt(); ②
        createdUser.salt = salt;
```

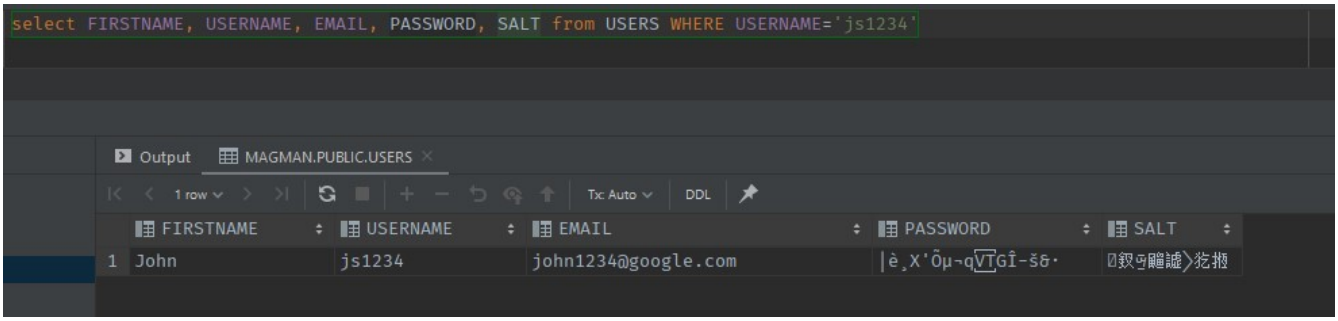
```
        createdUser.password = passwordService.encryptPassword(password, salt);

        createdUser.persist();
        return createdUser;
    }

    public Optional<User> loginUser(String username, String password) {
        User user = User.find("userName", username).firstResult(); ③
        if (user != null) {
            var hashedPassword = passwordService.encryptPassword(password, user.salt);
            if (hashedPassword.equals(user.password)) {
                return Optional.of(user);
            }
        }

        return Optional.empty();
    }

}
```

① Just like injecting `UserService` into `UserResource`, here we are injecting the `PasswordService`. When the `PasswordService` proxy is called for the first time, CDI will instantly create a singleton instance, which will remain active through the whole runtime of the server.

② Here, before we persist the newly created user, we first need to encrypt their password. First we need to generate the salt and set it to the user, then we need to call the `encryptPassword()` method and set the password to the new user. Finally we are able to persist, the user.

③ Now that we store the encrypted password, we cannot directly compare the plain text password with the one stored into the database. To be able to validate the user's password, we first need to encrypt it, and to do so, we need the same salt we used during the encryption in the first place, so this requires to pull out the user from the database to get their salt. If both the encrypted passwords match, then we can return that user to the resource.

Now if we try to create a new user and try to query the database, to see how they were created, we'll stumble upon the following result:



8.3. Injecting `EntityManager` with CDI

Remember when we instantiated the `EntityManager` from the constructor of a resource? The reason why this worked, is because in Quarkus, CDI works even on constructor level. And having said that JTA-managed `EntityManager` is controlled by the CDI container as well.

Let's go back to our `CommentResource` and see how we can optimize our class. Knowing that we can separate more complex operations into a different layer of our package (called service), let's move some stuff to the `CommentService`...

```
@ApplicationScoped
public class CommentService {

    @Inject
    EntityManager entityManager; ①

    public Comment createComment(String content, User author, Article article) {
        Comment comment = new Comment(content, author, LocalDateTime.now());
        comment.article = article;
        comment.persist();
        return comment;
    }

    public Optional<CommentDTO> getCommentById(long commentId) {
        return entityManager.createNamedQuery(Comment.GET_COMMENT_BY_ID, CommentDTO
.class)
                .setParameter("commentId", commentId)
                .getResultStream()
                .findFirst();
    }

}
```

① As you can see `EntityManager` could be injected using the same techniques as the one used for other dependencies. It's worth mentioning that the instance of JTA managed `EntityManager` is RequestScoped.

As for the `CommentResource` class, it will end up looking like this:

```
@Path("/article/{id}/comment")
public class CommentResource {

    @Inject
    CommentService commentService;

    @POST
    @Transactional
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createComment(@PathParam("id") Long articleId, CommentDTO
commentDTO) {
        if (articleId < 1 || commentDTO.getAuthorId() == null || commentDTO
.getAuthorId() < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }
```

```
        User author = User.findById(commentDTO.getAuthorId());
        if (author == null) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        Article article = Article.findById(articleId);
        if (article == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        }

        Comment comment = commentService.createComment(commentDTO.getContent(),
author, article);

        return Response.created(URI.create(String.format("/article/%d/comment/%d",
articleId, comment.id))).build();
    }

    @GET
    @Path("/{commentId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getCommentById(@PathParam("commentId") Long commentId) {
        if (commentId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            return commentService.getCommentById(commentId)
                    .map(c -> Response.ok(c).build())
                    .orElseGet(() -> Response.status(Response.Status.NOT_FOUND).build
());
        }
    }
}
```

8.4. `@PostConstruct` and `@PreDestroy`

In a typical class, the way you initialize an instance and its properties is by using the constructor. But in CDI managed class there is more safe and convenient way to do so. By creating a method and annotating it with `@PostConstruct`, CDI will invoke that method, the moment, the class is ready to use all of it's dependencies.

For example, let's say we want to log every attempt to access the `UserResource` endpoints.

```
@Path("user")
public class UserResource {

    Logger logger = Logger.getLogger(this.getClass().getName());

    @Inject
    UserService userService;
```

```
    @Inject
    HttpServerRequest request;

    @PostConstruct
    void init() {
        String requestPath = request.uri();
        var originIp = request.remoteAddress().toString();
        logger.log(Level.INFO, "URL call attempt {0} from {1}", new String[]
{requestPath, originIp});
    }

    //Endpoints
}
```

As you can see, we're taking the request data from the class `io.vertx.core.http.HttpServerRequest`, which is injected by CDI. If we did that in the constructor of the class, we wouldn't have been able to use that class. This comes to our conclusion, that the execution of `@PostConstruct` happens after all of our dependencies are injected.

`@PreDestroy` works in the same manner, but to see it in work, you'll need to annotate your resource with `@RequestScoped`, as the scope of resource by default is `@Singleton` and `@PreDestroy` never gets called. Then we can create a method like this:

```
@RequestScoped
@Path("user")
public class UserResource {

    Logger logger = Logger.getLogger(this.getClass().getName());

    @Inject
    UserService userService;

    @Inject
    HttpServerRequest request;

    @PostConstruct
    void init() {...}

    @PreDestroy
    void destroy() {
        String requestPath = request.uri();
        var originIp = request.remoteAddress().toString();
        logger.log(Level.INFO, "Scope completed for {0} from {1}", new String[]
{requestPath, originIp});
    }

    //Endpoints
}
```

Once the response of the endpoint has been returned, the `destroy()` method will be called, and execute all the code inside. You can use this to close certain kind of connections or execute some kind of a task. Or like in our case, just log something.

8.5. Interceptors and decorators

One other useful tool CDI provides is the ability to intercept or decorate the executions of a CDI-managed method. Basically both do the same thing, but serve a different purpose.

An example of where an interceptor or decorator could be used, would be in the cases where something's meant to happen, but it is neither a concern for the resource, nor for the service class. In our project when we create an article or a comment, we use to set the publish date inside the constructor of the Article/Comment. There's nothing wrong with that, but in theory it's an action that is not a concern for either of the service classes, making the article. It is also repetitive. Actions or operations which do not belong to the purpose of the single responsibility of a method are called **cross-cutting concerns**. So what can we do about it?

8.5.1. Interceptors

First, since both articles and comments have the same property called `publishDate` (or `created`), we can unify those in an abstract class, to inherit them on both classes.

```
@MappedSuperclass
public class PublishedContent extends AbstractEntity {

    public LocalDateTime publishDate;
    public LocalDateTime lastModified;

}

//And then extend our entities, and remove their date property

public class Article extends PublishedContent {...}

public class Comment extends PublishedContent {...}
```

Now in order to attach the appropriate dates to our content, we need to create some annotations. To make things cleaner, let's add a new package, called `interceptors`, and add two new annotations:

- `@CreatesContent`
- `@ModifiesContent`

```
import jakarta.interceptor.InterceptorBinding;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
```

```
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Inherited
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface CreatesContent {
}

@Inherited
@InterceptorBinding
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface ModifiesContent {
}
```

Then we need to place these annotations on the service methods we are going to intercept.

```
@ApplicationScoped
public class ArticleService {

    @CreatesContent
    public Article createArticle(String title, String content, Author author) {
        Article article = new Article(title, content, author);
        article.persist();
        return article;
    }

    @ModifiesContent
    public void editArticle(Article article, String title, String content) {
        article.title = title;
        article.content = content;
    }

}
```

You can do the same with the comment.

But we're not finished. Although we have the annotations we are still not intercepting anything, as we have not implemented any logic for that. Let's do it.

Inside the `interceptors` package, let's create two new classes - `CreatesContentInterceptor` and `ModifiesContentInterceptor`.

```
@Interceptor ①
@CreatesContent ②
@Priority(Interceptor.Priority.APPLICATION) ③
```

```
public class CreatesContentInterceptor {

    @AroundInvoke
    public Object contentCreated(InvocationContext invocationContext) throws Exception
{
        Object returnedObject = invocationContext.proceed(); ④
        if (returnedObject instanceof PublishedContent) {
            var content = (PublishedContent) returnedObject;
            content.publishDate = LocalDateTime.now();
            content.lastModified = LocalDateTime.now();
        }

        return returnedObject;
    }

}

@Interceptor ①
@ModifiesContent ②
@Priority(Interceptor.Priority.APPLICATION) ③
public class ModifiesContentInterceptor {

    @AroundInvoke
    public Object contentModified(InvocationContext invocationContext) throws
Exception {
        Object[] arguments = invocationContext.getParameters(); ⑤
        for (var argument : arguments) {
            if (argument instanceof PublishedContent) {
                var content = (PublishedContent) argument;
                content.lastModified = LocalDateTime.now();
            }
        }

        return invocationContext.proceed(); ⑥
    }
}
```

① With the `@Interceptor` annotation, we tell our application server that this class serves as an interceptor.

② The annotation we created will tell CDI to which classes it should listen to invoke them.

③ The `@Priority` annotation defines the order of invocation of annotations. Its value is of type integer. The higher the priority the least the chance of this annotation being called first. This is helpful for example when one method has two interceptors. With the `@Priority` annotation you can define which interceptor gets called first. `Interceptor.Priority.APPLICATION` equals 0, meaning that this interceptor it will be called with highest priority.

④ `invocationContext.proceed()` means to execute the intercepted method. You can run your method during interception. Here this is done, so we can get the return value of that method, which we are interested in. Then we are checking if the returned type is the type we need and set the date of the content.

⑤ In this case we are interested of the passed entity, which is going to be updated. We are considering that all of our update methods will contain the entity which is going to be updated. If that's not the case, this interceptor won't do anything.

⑥ Here we are not interested of the interceptor's result, so we are directly returning whatever needs to be returned. (In object oriented programming `void` is also a return type)

Keep in mind that in our examples the `@Transactional` scope is running during this operations, as it is annotated on the resource level. So any changes created to the entities in the scope of the interceptor are going to reflect on the entity. If your implementation is different, you'll need to adjust it to ensure transactions are occurring and entities are attached.

Knowing all that, we can understand how the `@Transactional` annotation is working. It is an interceptor responsible for managing the transactions during the execution of a method.

8.5.2. Decorators

Unlike interceptors, which are meant to be used mostly to modify unrelated to the resource or service data, decorators have a bit more different purpose. The use of decorators, as the name suggests, is meant do *decorate* the data. A scenario where a decorator would be useful is if you make some kind of request, but the response contains stuff that no service or resource is responsible to provide.

Let's develop our `Advertiser` entity to see how it can come in handy, when using decorators.

- Create `AdvertiserResource` that supports some type of CRUD operations
- Create `AdvertiserService` if needed
- Create some `Advertiser` entities

In most blogs or magazines you'll see sponsored messages or a list of sponsors. In our Magazine Manager, the advertiser is a sponsor who has donated to support an article. We want to list those advertiser names into the content of an article. This creates a challenge as we neither want to edit the content of the article, nor do we want to introduce a logic to the `ArticleService` that is not a concern for it. So what do we do? We create a decorator.

First, let's move the `getArticle()` method into `ArticleService`. To simplify stuff, we're going to ditch the `withComments` part. It was used only to demonstrate how `NamedQueries` work, but in real practice you wouldn't do it like that for something as simple.

This would make our `ArticleResource` and `ArticleService` look like this:

```
@RequestScoped
@Path("/article")
public class ArticleResource {

    @Inject
    ArticleService articleService;
```

```
    //endpoints

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getArticle(@PathParam("id") Long articleId) {
        if (articleId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        return articleService.getArticle(articleId)
                .map(article -> Response.ok(new ArticleDTO(article)).build())
                .orElseGet(() -> Response.status(Response.Status.NOT_FOUND).build());
    }


    //endpoints
}

@ApplicationScoped
public class ArticleService {

    public Optional<Article> getArticle(long articleId) {
        return Article.findByIdOptional(articleId);
    }

    @CreatesContent
    public Article createArticle(String title, String content, Author author) {
        Article article = new Article(title, content, author);
        article.persist();
        return article;
    }

    @ModifiesContent
    public void editArticle(Article article, String title, String content) {
        article.title = title;
        article.content = content;
    }
}
```

Then we recommend to rename the `ArticleService` into `ArticleServiceImpl`, so you can turn the `ArticleService` class into interface. Add all the methods of `ArticleServiceImpl` into that interface.

```
public interface ArticleService {

    Optional<Article> getArticle(long articleId);

    Article createArticle(String title, String content, Author author);

    boolean editArticle(Article article, String title, String content);
```

```
}

@ApplicationScoped
public class ArticleServiceImpl implements ArticleService {

    public Optional<Article> getArticle(long articleId) {...}

    @CreatesContent
    public Article createArticle(String title, String content, Author author) {...}

    @ModifiesContent
    public boolean editArticle(Article article, String title, String content) {...}

}
```

Creating a decorator will require all interface methods to return a value. That's why we set the return type of `editArticle` to boolean. Otherwise it will throw an error `java.lang.VerifyError: Method expects a return value`.

Then it's time to build our decorator class.

```
@Decorator ①
public abstract class ArticleDecorator implements ArticleService { ②

    @Inject
    @Delegate ③
    ArticleService articleService;

    @Override
    public Optional<Article> getArticle(long articleId) {
        return articleService.getArticle(articleId)
                .map(this::decorateArticle);
    }

    private Article decorateArticle(Article article) {
        var advertisers = article.advertisers;
        var message = String.format("\nThis article has been sponsored by: %s",
                advertisers.stream().map(a -> a.name).collect(Collectors.joining(", "
)));
        var decoratedArticle = new Article(article.title, article.content + message,
article.author); ④
        decoratedArticle.id = article.id;
        return decoratedArticle;
    }
}
```

① We use the `@Decorator` annotation to notify CDI that this class will be called to wrap the real instance of `ArticleServiceImpl`.

② The decorator should be abstract and implement `ArticleService` as well. Making it abstract let us control on which methods should we implement decorators.

③ The `@Delegate` annotation is mandatory for CDI to inject the appropriate implementation for `ArticleService`.

Usually when you pass Inject on an interface, CDI will try and look for implementations with CDI context. We'll talk more about that in our "*Producers and Alternatives*" section.

④ At the ned we create a new instance of `Article` in order to detach it from the database, so we don't cause any changes on the attached entity we might be working with. As you can see by the parameters, we can't be sure when is this method called. It might be called in `@Transactional` scope, so we have to make sure that we are not working directly with the entity.

Now add some advertisers to some articles, so you can experience the decoration magic.

```
@RequestScoped
@Path("/article")
public class ArticleResource {

    //Other endpoints

    @PATCH
    @Transactional
    @Path("{id}/advertiser/{advertiserId}")
    public Response addAdvertiserToArticle(@PathParam("id") Long id, @PathParam(
"advertiserId") Long advertiserId) {
        if (id < 1 || advertiserId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        Article article = Article.findById(id);
        Advertiser advertiser = Advertiser.findById(advertiserId);

        if (article == null || advertiser == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        }

        article.advertisers.add(advertiser);
        return Response.status(Response.Status.NO_CONTENT).build();
    }
}
```

Now when you make a call for getting an article by id, you should get a response similar to this:

```
{
    "authorId": 1,
    "content": "The quick brown fox runs over the lazy dog.\nThis article has been
```

```
sponsored by: Google",
    "id": 3,
    "title": "Article for the soul."
}
```

Want to do some more decorators?

Why don't you try and make a decorator which is going to extract the image of the sponsor upon getting it and convert that image to Base64 string?

Here are some tips.

- Create an interface `AdvertiserMapper` which is going to convert `Advertiser` to `AdvertiserDTO`
- Create an implementation for that mapper
- Create a decorator which is going to take the blob byte content of the advertiser's logo and convert it to base64 string.
- Attach that string logo to the DTO of the advertiser.

8.6. Producers and alternatives

So far we have injected an instance of a single class. But what happens when you have multiple implementations of a dependency or you want to use CDI to provide e certain type of information? In a typical program, when facing such a scenario, you would likely use the factory pattern and let the factory class decide which instance to return. In CDI on the other hand you can use the power of annotations to control what and when to be injected into the annotated class.

8.6.1. Producers

When you have two or more implementations of a dependency, or you want to inject something that by default is not a CDI bean, you can use CDI producers, which will generate the value you need and inject it. For example let's have a look at our `ArticleDecorator` class. We are getting all the advertisers and decorate an article with them. But let's say that we want to decorate the article only with `GOLD` advertisers. One way to do it, would be to pull out the advertisers with a query, filtering them by their type. Other way to do it, would be by creating a producer.

For a start, create a package, called `producers` inside the `advertiser` package. Inside, create the class `AdvertiserProducer`. Next, create three new annotations - `@Gold`, `@Silver`, `@Bronze`.

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Gold {

    SponsorPackage SPONSOR_PACKAGE = SponsorPackage.GOLD; ①

    @Nonbinding ②
```

```
    int limit() default 0;

}

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Silver {
    SponsorPackage SPONSOR_PACKAGE = SponsorPackage.SILVER; ①

    @Nonbinding ②
    int limit() default 0;
}

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Bronze {

    SponsorPackage SPONSOR_PACKAGE = SponsorPackage.BRONZE; ①

    @Nonbinding ②
    int limit() default 0;

}
```

① We are defining a constant which we are later going to use later for our implementation.

② The `@Nonbinding` annotation tells CDI not to compare the value, that might be set to it, with the annotation, which the producer will expect. If this annotation is not present, our producer would not work, as it will try to search for matching annotation that has `@Gold(limit = 2)` annotation for example.

Now it's time to go back to the `AdvertiserProducer` and implement the methods.

```
public class AdvertiserProducer {

    @Gold ①
    @Produces ②
    @Dependent ③
    public List<Advertiser> produceGoldAdvertisers(InjectionPoint ctx) {
        var limit = ctx.getAnnotated().getAnnotation(Gold.class).limit(); ④
        return getAdvertisers(Gold.SPONSOR_PACKAGE, limit);
    }

    @Silver ①
    @Produces ②
    @Dependent ③
    public List<Advertiser> produceSilverAdvertisers(InjectionPoint ctx) {
        var limit = ctx.getAnnotated().getAnnotation(Silver.class).limit(); ④
        return getAdvertisers(Silver.SPONSOR_PACKAGE, limit);
```

```
    }

    @Bronze ①
    @Produces ②
    @Dependent ③
    public List<Advertiser> produceBronzeAdvertisers(InjectionPoint ctx) {
        var limit = ctx.getAnnotated().getAnnotation(Bronze.class).limit(); ④
        return getAdvertisers(Bronze.SPONSOR_PACKAGE, limit);
    }

    private List<Advertiser> getAdvertisers(SponsorPackage sponsorPackage, int limit)
{
        var query = Advertiser.<Advertiser>find("sponsorPackage = ?1", sponsorPackage
);
        if (limit > 0) {
            query = query.page(0, limit);
        }

        return query.list();
    }

}
```

① We use our qualifier annotation to help CDI choose which method to call. This is mandatory, when there's more than one option to provide instance of a class.

② This annotation tells CDI that this is a producer method. It is invoked when the annotation above is used with `@Inject` within the definition of a CDI managed class.

When you import `@Produces`, be sure to check the package, you are importing from. It should be either `jakarta.enterprise.inject.Produces` or `jakarta.enterprise.inject.Produces`. You must be careful not to confuse it with `jakarta.ws.rs.Produces`/`jakarta.ws.rs.Produces`, as you already know that this annotation is used to describe the return type for JAX-RS, which has nothing to do with CDI producers.

③ Each producer method should have a scope, so CDI can decide what instance to create for the injection point. The reason we use `@Dependant` scope here is because in order to be able to get the data of that injection point, hence the name of the parameter `InjectionPoint`, we need to have this scope on our producer.

④ We are able to read the current value of the annotation of the injection point. You will see in the example below how we can use it later.

Now that our provider is implemented, we can add it to the `ArticleDecorator` class like that:

```
@Decorator
public abstract class ArticleDecorator implements ArticleService {

    @Inject
    @Delegate
```

```
    ArticleService articleService;

    @Inject ①
    @Gold(limit = 10) ②
    List<Advertiser> goldAdvertisers;

    @Override
    public Optional<Article> getArticle(long articleId) {
        return articleService.getArticle(articleId)
                .map(this::decorateArticle);
    }

    private Article decorateArticle(Article article) {
        var message = String.format("\nThis article has been sponsored by: %s",
                goldAdvertisers.stream().map(a -> a.name).collect(Collectors.joining(
", "))); ③
        var decoratedArticle = new Article(article.title, article.content + message,
article.author);
        decoratedArticle.id = article.id;
        return decoratedArticle;
    }
}
```

① When we want to call a producer, engaged with an annotation, we always use the `Inject` annotation. This is the way CDI will know what to look for.

② As we defined, we can optionally set a limit to our results. So when it comes to the execution of the producer, it will take this value into account.

③ When it comes to the implementation of the decorator, we can use the instance of `goldAdvertisers` as any other object ot type list. We rely that CDI has successfully produced the data required.

8.6.2. Alternatives

When going through different implementations of the same class, there will be a point where you will need to create more than just one implementation. In this type of scenario, you will need to use the power of alternatives. The most common scenario where you will need an alternative will be within a test environment.

For example if you have a service which requires external actions such as contacting remote service or executing actions, depending on real live environments, such as payment service, which needs to connect to a payment provider. When you need to test that service you could create an alternative implementation, annotating it with `@Alternative`, which mocks the invocation of the real instance. To find our more about alternatives, read this article.

8.7. Events and Observers

In our final feature of CDI, we are going to talk about Events and Observers. With CDI, you can achieve event-driven development, while implementing the event-observer pattern into you code.

Let's see an example, how we can achieved this in our code.

When a subscriber wants to extend their subscription, we usually need to charge them some amount of money in order to allow them to keep using our magazine manager. In order to do so, we will need to create some form of subscription resource. And that's what we are going to do...

- Create the class `subscription.SubscriptionResource`
- Add endpoints, such as:
  - `PUT /subscription/{userId}` - adds/updates payment method to the subscriber
  - `POST /subscription/{userId}` - updates subscriber's subscription for a certain period
- Create a `PaymentService, which is going to process payments`

At the end you should have something like this:

```
@RequestScoped
@Path("subscription")
public class SubscriptionResource {

    @Inject
    PaymentService paymentService;

    @PUT
    @Transactional
    @Path("{userId}")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response addPaymentMethod(@PathParam("userId") Long userId, CreditCardDTO
creditCardDTO) {
        if (userId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        if (creditCardDTO.getType() == null || creditCardDTO.getNumber() == null) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        CreditCard creditCard = new CreditCard(creditCardDTO.getNumber(),
creditCardDTO.getType());
        return Subscriber.<Subscriber>findByIdOptional(userId)
                .stream().peek(s -> s.creditCard = creditCard)
                .findFirst().map(s -> Response.status(Response.Status.NO_CONTENT)
.build())
                .orElseGet(() -> Response.status(Response.Status.NOT_FOUND).build());
    }

    @POST
    @Path("{userId}")
    public Response chargeSubscriber(@PathParam("userId") Long userId) {
        if (userId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
```

```
        }

        return Subscriber.<Subscriber>findByIdOptional(userId)
                .map(paymentService::chargeSubscriber)
                .map(result -> result? Response.status(Response.Status.NO_CONTENT)
.build() : Response.status(Response.Status.NOT_ACCEPTABLE).build())
                .orElseGet(() -> Response.status(Response.Status.NOT_FOUND).build());
    }
}

@ApplicationScoped
public class PaymentService {

    private static final Logger LOGGER = Logger.getLogger(PaymentService.class.
getName());

    boolean chargeSubscriber(Subscriber subscriber) {
        if (subscriber.creditCard != null) {
            LOGGER.log(Level.INFO, "Charging subscriber with id: {0}  and card type
{1} of number: {2}",
                    new Object[]{subscriber.id, subscriber.creditCard.creditCardType,
subscriber.creditCard.number});

            return true;
        }

        return false;
    }

}
```

Here we are not going to implement the real thing, let's say that we just check if the user has credit card linked, and we log that we have charged them. As you can see we don't have the most important part of the subscription renewal - updating the expiration date. In practice some actions such as renewing a subscription would trigger more than one operation in action. For example you need to update the user's subscription expiration date, send email to the user with confirmation and receipt, send notification to the manager that a user has renewed their subscription, and so on. We can't put all of these side operations into the `chargeSubscriber()` method. It would kill the purpose of the name of this method, which is to only charge the subscriber. It doesn't say email subscriber, extend expiration to the subscriber, notify that person & etc. Those type of side actions occur upon a certain event. And CDI offers a way to handle events occurring outside the scope of the `chargeSubscriber` method.

First create an annotation, called `@ChargedSubscriber` in the subscription package:

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface ChargedSubscriber {
```

```
}
```

Then let's add couple of handlers which are going to handle this event.

```
public class SubscriptionExtensionHandler {

    private static final Logger LOGGER = Logger.getLogger(
SubscriptionExtensionHandler.class.getName());

    @Transactional
    @ActivateRequestContext ①
    public void observeSubscriptionExtension(@Observes @ChargedSubscriber Subscriber
subscriber) { ②
        subscriber.subscribedUntil = subscriber.subscribedUntil.plusYears(1);
        LOGGER.log(Level.INFO, "Extended subscription for user {0}, till {1}",
                List.of(subscriber.id, subscriber.subscribedUntil.toString()).toArray
());
    }

    public void sendEmail(@Observes @ChargedSubscriber Subscriber subscriber) { ②
        LOGGER.log(Level.INFO, "Sent email to subscriber {0}, about their subscription
renewal.", subscriber.id);
    }

}
```

① Events might get invoked outside a request context. The request context is fired upon request and transactional methods can run only within it, meaning that if the context is not present, `@Transactional` may not work. To make sure that we are running in a `@Transactional` method, we need to add `@ActivateRequestContext` annotation, which will ensure the request context on the

② With the `@Observes` annotation we are signalling CDI to listen for the occurrence ot an event being fired. Then we use our newly made annotation, which specifies in which scenarios those methods should be invoked. The object we pass as an argument, also known as the event payload, is the object we are expecting to get when the event is fired.

Making the annotation for an event is optional. You could create events and fire them without annotation, but in this occasion you may also fire event listeners which you might not know of, that also are listening for that event.

Next in order to fire those events upon making a payment, we need to update our payment service.

```
@ApplicationScoped
public class PaymentService {

    private static final Logger LOGGER = Logger.getLogger(PaymentService.class.
getName());

    @Inject ①
```

```
    @ChargedSubscriber ②
    Event<Subscriber> onSubscriberCharged; ③

    boolean chargeSubscriber(Subscriber subscriber) {
        if (subscriber.creditCard != null) {
            LOGGER.log(Level.INFO, "Charging subscriber with id: {0}  and card type
{1} of number: {2}",
                    new Object[]{subscriber.id, subscriber.creditCard.creditCardType,
subscriber.creditCard.number});

            onSubscriberCharged.fire(subscriber); ④
            return true;
        }

        return false;
    }

}
```

① As any other CDI managed dependency, in order to signal the server that this property is CDI managed, we need to use the `@Inject` annotation.

② To deffer which type of event handlers we are going to call, we add our annotation here. It's important to mention that if we do not add the annotation, our handlers won't get called.

③ The we define our event with the type of payload we are going to pass. Here we use `jakarta.enterprise.event.Event` or `jakarta.enterprise.event.Event`, depending on your flavour of enterprise edition of Java.

④ To fire the event, we simply call the `.fire()` method, passing the instance of our payload.

Let's test that and see the results with a user we previously created.

```
curl --location --request POST 'http://localhost:8080/subscription/7'
```

Now it you look at the logs, you will find the following:

```
2022-09-27 13:23:18,965 INFO  [com.vid.mag.sub.PaymentService] (executor-thread-0)
Charging subscriber with id: 7  and card type VISA of number: 3698521479
2022-09-27 13:23:18,973 INFO  [com.vid.mag.sub.SubscriptionExtensionHandler]
(executor-thread-0) Extended subscription for user 7, till 2025-01-12
2022-09-27 13:23:18,973 INFO  [com.vid.mag.sub.SubscriptionExtensionHandler]
(executor-thread-0) Sent email to subscriber 7, about their subscription renewal.
```

Now this is great, and it seems that the order of invocations are working as expected, but what if we want to flip things around? What if we want to first send an email to the user, then update the entity?
This could be done with the `@jakarta/jakarta.annotation.Priority` annotation:

```
    @Transactional
    @ActivateRequestContext
    public void observeSubscriptionExtension(@Priority(Priorities.APPLICATION + 2000)
@Observes @ChargedSubscriber Subscriber subscriber) {
        subscriber.subscribedUntil = subscriber.subscribedUntil.plusYears(1);
        LOGGER.log(Level.INFO, "Extended subscription for user {0}, till {1}",
                List.of(subscriber.id, subscriber.subscribedUntil.toString()).toArray
());
    }

    public void sendEmail(@Priority(Priorities.APPLICATION + 1000) @Observes
@ChargedSubscriber Subscriber subscriber) {
        LOGGER.log(Level.INFO, "Sent email to subscriber {0}, about their subscription
renewal.", subscriber.id);
    }
```

- `APPLICATION` priority, marks the priority of invocations of interceptors, defined by user applications. When defining a priority, it is recommended to start with that priority.
- Setting the priority will be a sum of a number and that `APPLICATION` priority value (5000). The method lower priotity will be executed first.

8.8. Conclusion

This concludes the basics of using CDI within Quarkus. In the next chapter, we are going to get rid of all those checks on requests, by implementing bean validation.

Chapter 9. Bean validation

This chapter is dedicated to figuring out a better way to validate our requests and the data we pass from one service to another overall. Jakarta EE specifies annotations, used to validate data. So instead of you making those checks upon each and every request, there are annotations specifically made for that purpose. As we are always mentioning a method should have a single responsibility. Having to check the integrity and correctness of the data kind of invalidates that purpose. The way to achieve this is through Bean Validation.

As mentioned earlier "*bean*" in Java Enterprise world is interchangeable word for CDI managed dependency.

9.1. Integrating bean validation to our project.

Just like any other Jakarta EE specification, the annotations for bean validation are just interfaces, provided by the community, and implemented by web server providers. This means that we have to use the extension with which Quarkus allows bean validation for your project. Since the extension is not provided into our project, we first need to install it, like we did for JPA.

```
mvn quarkus:add-extension -Dextensions="io.quarkus:quarkus-hibernate-validator"
```

After the operation has succeeded, we are going to find a new dependency in our pom.xml.

```
    <dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-hibernate-validator</artifactId>
    </dependency>
```

9.2. Using bean validation annotations

Once start our project, after importing the bean validation extension, it will be on and will start looking for properties with annotations, using the spec. Since the previous chapter, you should have the knowledge how day work (it's interceptors).

Now it is time to introduce our project to these annotations. Let's start with the `ArticleResource`, our first resource ever.

9.2.1. Configuring bean validations REST resources

Our method for getting article by id, might currently look like this:

```
    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getArticle(@PathParam("id") Long articleId) {
```

```
        if (articleId < 1) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        return articleService.getArticle(articleId)
                .map(article -> Response.ok(new ArticleDTO(article)).build())
                .orElseGet(() -> Response.status(Response.Status.NOT_FOUND).build());
    }
```

Depending on your implementations, you might have more checks, like for example checking if the id is not less than 1 or something else. What's important is that we need at least two checks here:

- The passed article id should not be null
- The past article id should be of a positive number (greater than zero)

To achieve that, we simply need to add `@Positive` next to, or before, the `@PathParam` annotation, like so:

```
public Response getArticle(@Positive @PathParam("id") Long articleId) {...} ①
```

① All bean validation annotations come from the `javax/jakarta.validation.constraints` package.

Then we can get rid of the check within the method's implementation.

```
    public Response getArticle(@Positive @PathParam("id") Long articleId) {
        return articleService.getArticle(articleId)
                .map(article -> Response.ok(new ArticleDTO(article)).build())
                .orElseGet(() -> Response.status(Response.Status.NOT_FOUND).build());
    }
```

Now let's try it! First, let's call for a real article and make sure hat the expected outcome is still working:

```
curl -i --location --request GET 'http://localhost:8080/article/3'
```

```
HTTP/1.1 200 OK
Content-Type: application/json
content-length: 147

{"authorId":1,"content":"The quick brown fox runs over the lazy dog.\nThis article has
been sponsored by: ","id":3,"title":"Article for the soul."}
```

Good! Now let's try some invalid input:

```
curl -i --location --request GET 'http://localhost:8080/article/-3'
```

What do you get?

If your response is:

```
HTTP/1.1 400 Bad Request
validation-exception: true
Content-Type: application/json
content-length: 206

{
    "classViolations": [],
    "parameterViolations": [
        {
            "constraintType": "PARAMETER",
            "message": "must be greater than 0",
            "path": "getArticle.articleId",
            "value": "-3"
        }
    ],
    "propertyViolations": [],
    "returnValueViolations": []
}
```

It means that it's working and you're all set.

Now let's apply that knowledge and add some annotations to all of our methods.

```
public Response createArticle(@NotNull ArticleDTO articleDTO) {...}

public Response editArticle(@Positive @PathParam("id") Long articleId, @NotNull
ArticleDTO articleDTO) { ... }

public void deleteArticle(@Positive @PathParam("id") Long articleId) { ... }

public Response getArticles(@QueryParam("page") @DefaultValue("1") @Positive int page,
                            @QueryParam("size") @DefaultValue("10") @Positive int
size,
                            @QueryParam("author") @Positive Long authorId) { ... }

public Response addAdvertiserToArticle(@Positive @PathParam("id") Long id, @Positive
@PathParam("advertiserId") Long advertiserId) {...}
```

`@NotNull` is another validation annotation, which as the name suggests makes sure that the data you are passing is not null.

Using `@NotNull` checks on a path parameter in practice is useless, as there is no way to pass a null path parameter. We also needn't check if the input is a number, as RestEasy is deferring letters as another paths.

9.3. Configuring bean validation on POJOs

We found out how to set bean validations on our resource methods, but what if we want to validate the body we are putting into our query?
For example when we create an article, we want to be sure that the article has at least `title` and `content`. Other properties, such as publish and modification date, id or author id are not required, as those are properties, given automatically by our application.

So how to we validate out `ArticleDTO`?
With annotations!

```
public class ArticleDTO {

    private Long id;

    @NotBlank ①
    @Size(min = 1, max = 225) ②
    private String title;

    @NotBlank ①
    @Size(min = 1, max = 10_000) ②
    private String content;

    private String publishDate;

    private String lastModified;
    private Long authorId;

    private List<CommentDTO> comments;

    // constructors, getters, setters
}
```

① With `@NotBlank` annotation, we can check if the property is null or empty.

② With `@Size`, we can add constraints on the size of the content. Since we defined into our database that the content of the article will be 10 000 characters at max, we cannot accept an article with more than that.

One final step, to make the bean validation work, is to add `@Valid`, next to the resource we are going to validate the DTO with.

```
public class ArticleResource {
```

```
    public Response createArticle(@Valid @NotNull ArticleDTO articleDTO) { ... }

    public Response editArticle(@Positive @PathParam("id") Long articleId, @Valid
@NotNull ArticleDTO articleDTO) { ... }

}
```

The `@Valid` annotation works when you want to run validation on object, passed on a resource, or when you want ot validate a nested object within the POJO. So for example if our article was supposed to be saved with some comments, and we had some validation annotations on the comments, we would need to put `@Valid` on the comments definition.

9.4. Setting custom messages on validations

Let's try another type of validation. We want to validate the password of a new user. A strong password is considered to have:

- At least 8 characters
- Combination of upper and lower-case letters
- Digits
- Special characters

Doing such a check could be done, with a single regular expression.

```
^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$
```

Now that we have that expression figured out, we can add it as validation rule, onto our `NewUserDTO`.

```
public class NewUserDTO extends UserDTO {

    @Pattern(regexp = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[@$!%*?&])[A-Za-z
\\d@$!%*?&]{8,}$")
    private String password;

    @NotNull
    private UserType userType;

    //getters and setters
}
```

Afterwards, as we did previously we need to add `@Valid` annotation to the `UserResource.registerUser()` method.

```
    public Response registerUser(@Valid @NotNull NewUserDTO newUserDTO) {...}
```

Don't forget to add validations on other properties inside `UserDTO`, so you can get rid of the manual checks.

And then let's test:

```
curl -i --location --request POST 'localhost:8080/user' \
--header 'Content-Type: application/json' \
--data-raw '{
    "firstName": "John",
    "lastName": "Smith",
    "password": "12345",
    "userName": "js1234",
    "email": "john1234@google.com",
    "userType": "SUBSCRIBER"
}'
```

We get the following response body:

```
{
    "classViolations": [],
    "parameterViolations": [
        {
            "constraintType": "PARAMETER",
            "message": "must match \"^(?=.*[a-z])(?=.*[A-Z])(?=.*
\\d)(?=.*[@$!%*?&])[A-Za-z\\d@$!%*?&]{8,}$\"",
            "path": "registerUser.newUserDTO.password",
            "value": "12345"
        }
    ],
    "propertyViolations": [],
    "returnValueViolations": []
}
```

Seems pretty messy, doesn't it?
Unfortunately we are not computers, and although programmers are kind of able to read RegEx, most people are not. We need a better, more user-friendly, way to tell the client that there's something wrong with the password. Fortunately all bean validation annotations come with a second property, called `message`. So if we add a custom message, this is what we are going to get, instead of the default value, defined by the validation annotation.

```
public class NewUserDTO extends UserDTO {

    @NotBlank
    @Pattern(regexp = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[@$!%*?&])[A-Za-z
```

```
\\d@$!%*?&]{8,}$",
    message = "The password must be at least 8 characters long, " +
            "contain at least one upper and lower-case character, at least one digit
and at least one special character.")
    private String password;

    // the other stuff
}
```

Now if we try the query again, we will get the following response.

```
{
    "classViolations": [],
    "parameterViolations": [
        {
            "constraintType": "PARAMETER",
            "message": "The password must be at least 8 characters long, contain at
least one upper and lower-case character, at least one digit and at least one special
character.",
            "path": "registerUser.newUserDTO.password",
            "value": "12345"
        }
    ],
    "propertyViolations": [],
    "returnValueViolations": []
}
```

That's ok, but what can you do if for example our system supports multiple countries. For example our service could run not only in english-speaking countries, but it may run in Spain, Japan, Germany. Just like RegEx, our users are not supposed to know english. To tackle this, we can create a localization bundle (A.K.A. l8n), to get different messages for different locales.

Create some files, called `ValidationMessages_en.properties`, `ValidationMessages_es.properties`, `ValidationMessages_ja.properties` into the `src/main/resources` folder. Your IDE my instantly recognize the group of these files as a Resource bundle. Then add a key-value pair of your message inside.

- In English:

```
invalid.password.format=The password must be at least 8 characters long, contain at
least one upper and lower-case character, at least one digit and at least one
special character.
```

- In Spanish

```
invalid.password.format=La contrase\u00f1a debe tener al menos 8 caracteres,
contener al menos un car\u00e1cter en may\u00fasculas y min\u00fasculas, al menos
```

```
un d\u00edgito y al menos un car\u00e1cter especial.
```

- And in Japanese

```
invalid.password.format=\u30d1\u30b9\u30ef\u30fc\u30c9\u306f \u0038
\u6587\u5b57\u4ee5\u4e0a\u3067\u3001\u5927\u6587\u5b57\u3068\u5c0f\u6587\u5b57\u3092
2 \u0031 \u6587\u5b57\u4ee5\u4e0a\u3001\u6570\u5b57\u3092 \u0031
\u6587\u5b57\u4ee5\u4e0a\u3001\u7279\u6b8a\u6587\u5b57\u3092 \u0031
\u6587\u5b57\u4ee5\u4e0a\u542b\u3080\u5fc5\u8981\u304c\u3042\u308a\u307e\u3059\u300
2
```

When using a language that contains characters outside ASCII, we need to escape those characters, because they won't be encoded to UTF-8.

Next step is to refer to that property inside the validation annotation.

```
public class NewUserDTO extends UserDTO {

    @NotBlank
    @Pattern(regexp = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[@$!%*?&])[A-Za-z
\\d@$!%*?&]{8,}$", message = "{invalid.password.format}")
    private String password;

    //some other properties I guess
}
```

When you put the content of the message into curly braces, Quarkus will detect that this is referring to a pair within the resource bundle, and will interpolate the message for you, depending on the user's locale.

One last thing that you need to do in Quarkus, is tell your application, that you're going to support several languages. You can do so, by adding `quarkus.locales=en,es,ja` into `application.properties`. If you miss that, you're going to spend a very long time, trying to figure our why it's not working.

Interpolation of values works not only for properties, but for values as well. For example, we are checking if the Article's title is between 1 and 255 characters. If we want, we can define the following message: `@Size(min = 1, max = 225, message = "The title of the article must be between {min} and {max} characters")`. This is a valid expression and will interpolate the values of `min` and `max` into the message. You can also use this within `ValidationMessages.properties` file.

Now let's see that in action...

If our user is Japanese and they try to create their account, the query would look like this:

```
curl -i --location --request POST 'localhost:8080/user' \
```

```
--header 'Accept-Language: ja' \ ①
--header 'Content-Type: application/json' \
--data-raw '{
    "firstName": "John",
    "lastName": "Smith",
    "password": "12345",
    "userName": "js1234",
    "email": "john1234@google.com",
    "userType": "SUBSCRIBER"
}'
```

① The `Accept-Language` header by default is optional. When it is not passed, our server will return response in the default language, which is usually English, unless something else is configured. If the user's operating system is different, the browser will pick that up and pass it to the header of every request.

Then in response we are going to get:

```
{
    "classViolations": [],
    "parameterViolations": [
        {
            "constraintType": "PARAMETER",
            "message": "パスワードは 8 文字以上で、大文字と小文字を 1 文字以上、数字を 1 文字以上、特殊文字を 1 文字以上含む必要があります。",
            "path": "registerUser.newUserDTO.password",
            "value": "12345"
        }
    ],
    "propertyViolations": [],
    "returnValueViolations": []
}
```

If Japanese is native to you and you see grammatical or lexical errors, please blame Google Translate for it, not us. If you see English errors, to begin with, blame it on the writer of this article.

9.5. Simplifying your validation error messages (Exception mappers)

Sometimes you may find the response object of validation violation too complex, or you may have agreement with your team how error messages should be formatted. So what can you do, when you have to agree on the format of the response, you need to return?

You can create exception mappers. Exception mappers are a bit more advanced part of JAX-RS and CDI, we have not touched yet, and we think it's time to introduce you to it.

Every time a bean validation is violated, what you get is a response of type `400 BAD REQUEST` with the

body of the violated property. You never see an exception inside the log of the application, you never get to handle the error by yourself. Usually when a violation happens, the hibernate bean validation will throw a `javax/jakarta.validation.ConstraintViolationException`. This exception is handled by an internal exception mapper, that takes the gathered data, and builds a response with the body of the violation information. We want to be able to control that.

Create a new package, called `exception` and add a new class, called `ConstraintViolationExceptionMapper`.

```
package com.vidasoft.magman.exception;

import jakarta.validation.ConstraintViolationException;
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.ext.ExceptionMapper;
import jakarta.ws.rs.ext.Provider;

@Provider
public class ConstraintViolationExceptionMapper implements ExceptionMapper
<ConstraintViolationException> {
    @Override
    public Response toResponse(ConstraintViolationException e) {
        return null;
    }
}
```

Now when a constraint is violated, we are going to get this handler invoked. If we try our bad request in this state, we are going to get `204 NO CONTENT` as a response, which means we have successfully overridden the behavior of our violation response.

Next step is to define our response. Let's say that we want our list of violations to look like this:

```
[
    {
        "property":"registerUser.newUserDTO.password",
        "message":"The password must be at least 8 characters long, contain at least
one upper and lower-case character, at least one digit and at least one special
character."
    },
    {
        "property":"registerUser.newUserDTO.userType",
        "message": "must not be null"
    }
]
```

First we need to define a POJO that is going to store that JSON.

```
public class ViolationMessage {
```

```
    private String property;
    private String message;

    public ViolationMessage() {
    }

    public ViolationMessage(String property, String message) {
        this.property = property;
        this.message = message;
    }

    public String getProperty() {
        return property;
    }

    public void setProperty(String property) {
        this.property = property;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Then we need to implement the handling of the exception into our mapper.

```
@Provider
public class ConstraintViolationExceptionMapper implements ExceptionMapper
<ConstraintViolationException> {
    @Override
    public Response toResponse(ConstraintViolationException e) {
        return Response.status(Response.Status.BAD_REQUEST)
                .entity(buildResponseContent(e))
                .build();
    }

    private List<ViolationMessage> buildResponseContent(ConstraintViolationException
violationEx) {
        var result = new LinkedList<ViolationMessage>();

        for (ConstraintViolation<?> violation : violationEx.getConstraintViolations())
{
            var property = violation.getPropertyPath().toString();
            var message = violation.getMessage();
            result.add(new ViolationMessage(property, message));
        }
```

```
        return result;
    }
}
```

Now if we try our request, we should be getting as a response our custom defined message.

9.6. Creating custom bean validations

The bean validation extension does not come with all scenarios and checks under the sun, you may come up with. For that reason the door is opened to add your own validations. To show you how, let's start with our password validation.

What if we don't want to use this wacky `@Pattern` annotation, but we want to use our own `@Password`.

Say no more!
Simply create a new annotation, called `Password` and implement it like this:

```
@Documented
@Retention(RUNTIME)
@Constraint(validatedBy = {}) ①
@Target({FIELD, METHOD, PARAMETER, ANNOTATION_TYPE})
@NotBlank @Pattern(regexp = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[@$!%*?&])[A-Za-z
\\d@$!%*?&]{8,}$") ②
public @interface Password {

    String message() default "{invalid.password.format}"; ③

    Class<?>[] groups() default {}; ③

    Class<? extends Payload>[] payload() default {}; ③
}
```

① With the `@Constraint` annotation, we're telling Quarkus that this annotation is going to be used for bean validation. The `validatedBy` property is used to define custom validators. We are going to touch this topic in a bit.

② All constraint validation annotations are able to inherit the behavior of other constraint validators. This is helpful when you want to combine couple of validations into one.

③ `message()`, `groups()` and `payload()` are mandatory fields for every constraint validation annotation. You need to add them every time.

Now on the `NewUserDTO` side, you can get rid ot the stack of annotations and use just `@Password`.

9.6.1. Creating custom validators

Sometimes you may not be able to validate something with the stock constraint validators. For example, we want to make sure the username of the user is unique, and we do not want to check

that inside our resource or service code. What can we do about it?

First we create an annotation.

```
@Documented
@Retention(RUNTIME)
@Constraint(validatedBy = {})
@Target({FIELD, METHOD, PARAMETER, ANNOTATION_TYPE})
public @interface Unique {

    String message() default "The entered username must be unique.";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

Next we need to create our validator.

```
public class UniqueUserValidator implements ConstraintValidator<Unique, String> { ①
    @Override
    public boolean isValid(String username, ConstraintValidatorContext
constraintValidatorContext) {
        return User.find("userName", username).firstResultOptional().isEmpty();
    }
}
```

① This generic interface here will ask for the annotation we are validating with and the type of the variable that is expected.

Then we go back to our `@Unique` annotation and inside `@Constraint(validatedBy = {})`, we refer to our validator.

```
@Documented
@Retention(RUNTIME)
@Constraint(validatedBy = {UniqueUserValidator.class})
@Target({FIELD, METHOD, PARAMETER, ANNOTATION_TYPE})
public @interface Unique { ... }
```

Now once we add this annotation to our `userName` property, the validator will work every time the bean validator is called.

9.7. Calling bean validation within the code

Sometimes for whatever reason you might want to call bean validation programmatically, instead of depending on Quarkus to do it. Let's create a service around it to see how it works.

Create a class, called `ValidationService`.

```
@ApplicationScoped
public class ValidationService {
    Validator validator;

    @PostConstruct
    void init() {
        validator = Validation.buildDefaultValidatorFactory().getValidator();
    }

    public <T> List<ViolationMessage> validateObject(T objectToValidate) {
        return validator.validate(objectToValidate)
                .stream()
                .map(v -> new ViolationMessage(v.getPropertyPath().toString(), v
.getMessage()))
                .collect(Collectors.toList());

    }
}
```

Then we can use that service for any of our resources.

```
@Path("/article/{id}/comment")
public class CommentResource {

    @Inject
    CommentService commentService;

    @Inject
    ValidationService validationService;

    @POST
    @Transactional
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createComment(@NotNull @Positive @PathParam("id") Long articleId,
CommentDTO commentDTO) {
        var commentViolations = validationService.validateObject(commentDTO);
        if (commentViolations.size() > 0) {
            return Response.status(Response.Status.BAD_REQUEST).entity
(commentViolations).build();
        }

        User author = User.findById(commentDTO.getAuthorId());
        if (author == null) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }

        Article article = Article.findById(articleId);
```

```
        if (article == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        }

        Comment comment = commentService.createComment(commentDTO.getContent(),
author, article);

        return Response.created(URI.create(String.format("/article/%d/comment/%d",
articleId, comment.id))).build();
    }

    // even more code
}
```

Now every time we try to create a comment, our `ValidationService` will make sure, we are not violating the `CommentDTO` rules.

9.8. What next?

Now it's your turn. Look through your code. Find possible places where you can add bean validation and make the world a slightly better place.

In the next chapter, we are going to talk a bit about system configurations, and how you can define your own properties with the `application.properties` file, and other ways, powered by the MicroProfile specifications.

Chapter 10. System properties with MicroProfile Config

One pretty neat thing about Quarkus is that it supports MicroProfile APIs, right our of the box.

What is MicroProfile

MicroProfile is a community-driven specification designed to provide a baseline platform definition that optimizes the Enterprise Java for microservices architecture and delivers application portability across multiple MicroProfile runtimes, including Open Liberty. It gives Java developers a platform to collaborate and innovate on areas of common interest. The MicroProfile specification consists of a collection of Enterprise Java APIs and technologies that together form a core baseline for microservices that aims to deliver application portability across multiple runtimes.

— *IBM Developer Blog*

MicroProfile provides APIs and extensions which will give more power to your applications, more popular of which include:

- **MicroProfile Config API** - set and configure system properties in an intuitive way, using CDI.
- **RestClient** - build REST client with the JAX-RS API annotations, to allow your app to connect with other services.
- **MicroProfile JWT** - integration for role-based access control (RBAC) to manage the access users have to the application.
- **Reactive messaging** - API for integrating messaging systems such as Kafka.
- **Fault tolerance** - API used for concurrent environments, helpful managing high load of queries, asynchronous code execution and connection issues.
- **OpenAPI** - APIs for building OpenAPI documentation, along with Swagger ui to help visualize ant test the endpoints listed by OpenAPI
- **Metrics** - this API allows to create metric custom counters to track key indexes crucial for the application's performance. The data could then be fed to services such as Elastic Search and used to create charts for analysis and every day monitoring.

These and many more are the goodies that MicroProfile brings on the table. Just like Java / Jakarta EE the spec is based on interfaces, which then have to be implemented by application server providers.

Through out the next chapters we are going to look into the listed features above.

With Quarkus, luckily, you have support for MicroProfile with the additional extensions. One specification that comes right out of the box is MicroProfile Config. Remember the `application.properties` file? It is already using MP Config. This is our entrypoint to this chapter.

10.1. What is MicroProfile Config?

Usually when you need to add custom configurations to your application, such as variables, that are needed for the proper initialization for your application, you would create an environment variable or pass that as a system property, or have a `.properties` file from where you would read that property.
This requires you to have implementations which will read those properties from all three sources and pass them safely to the application. It would require to check if the right properties are passed, if they exist, whether they should have a default value, which source of properties has the highest priority and so on.

MiroProfile provides a solution for this hassle, letting you, the developer, only worry about the names of the properties. With the use of a special annotation `@ConfigProperty` and the power of CDI, you can simply specify the name and type of the property, and the rest is handled by the application server. If the property is mandatory, and is not provided, the application won't start and will list all the properties needed to be set.

You could also choose for what type of environment, what properties to use. This would help you define how your application will behave on different environments, such as when your application is in `dev` mode or in `production` mode. For example if your application is doing some financial operations, such as payment service, you would want to connect to a testing server, instead of a real payment server, when it comes to `dev` environment, but when the application goes live, you would want to use the real services, and then you should have the stage configured for real life.

All of these concerns can be solved with MP Config, so let's see how.

10.2. Using MicroProfile Config in our application

The usage of the API is pretty straightforward. To test it out let's create our fist custom configuration property...

When it comes to management for our application, the one user who has all the power would be the manager.
So there's the question - can a magazine manager exist without the manager?
The answer should probably be no.

We need to have a default manager user created, who will lead our magazine manager platform. Let's set up an environment for one, shall we?

First off, create a service class inside the `user` package. Let's call it `ManagerInitiationService`. This class should serve to create the manager user, at the start of the application.

```
@ApplicationScoped
public class ManagerInitiationService {

    @Inject
    PasswordService passwordService;

    @Transactional
```

```
    public void createManager(@Observes StartupEvent startupEvent) { ①
        if (Manager.count() == 0) {
            var managerPassword = "manager";
            var managerSalt = passwordService.generateSalt();
            var manager = new Manager("manager", passwordService.encryptPassword
(managerPassword, managerSalt),
                    "Manager", "User", "manager@vida-soft.com");
            manager.salt = managerSalt;
            manager.persist();
        }
    }
}
```

① The `StartupEvent` is an empty payload event, fired by Quarkus, when the server is successfully initialized. When you want to do something with your application on startup, one of the ways it could and is recommended to be done is by listening for this event. Read more about application initialization here.

Now we have the manager user created if it's missing, but there is one problem we have not yet solved. What happens if we want to modify this property every time we need to change it? We don't want for example to allow the default manager password to be `manager` when talking about production environments or such. One thing we could do is ask for that property from the outside.

```
@ApplicationScoped
public class ManagerInitiationService {

    @Inject
    PasswordService passwordService;

    @Inject ①
    @ConfigProperty(name = "magman.manager.user", defaultValue = "manager") ②
    String managerUsername;

    @Inject ①
    @ConfigProperty(name = "magman.manager.password", defaultValue = "manager") ②
    String managerPassword;

    @Transactional
    public void createManager(@Observes StartupEvent startupEvent) {
        if (Manager.count() == 0) {
            var managerSalt = passwordService.generateSalt();
            var manager = new Manager(managerUsername, passwordService.
encryptPassword(managerPassword, managerSalt),
                    "Manager", "User", "manager@vida-soft.com");
            manager.salt = managerSalt;
            manager.persist();
        }
    }
}
```

① Config properties are injected by CDI producers. You already know how this works, so you should get the basic idea.

② With `org.eclipse.microprofile.config.inject.ConfigProperty` we tell our application where to look for the value of this property. The only mandatory field here is `name`. Providing `defaultValue` is optional, but if this is not configured before application startup, the application won't start.

Now if you try to run the same code, without any additional changes, and log int with the manager, you'll probably get the same result.

10.3. Ways to provide config properties

There's three ways to set config properties to your application.

1. By modifying the `application.properties` file (lowest priority)
2. By setting an environment variable
3. By passing the property as a system property with `-D` (highest priority)

All three methods are ordered from lowest to highest priority, meaning that the second will override the third and the third will override the second.

10.3.1. Modifying the `application.properties` file

This one is simple. Go to `src/main/resources/application.properties` and add the values of your choice like this:

```
magman.manager.user=admin
magman.manager.password=adm!n
```

Now when you redeploy your application, the default manager user will be created with those credentials.
One other thing to note here is that you can set different properties for different environments. This is also true for the properties we previously had as well. So for example if you want to have one user in `dev` environment and different one in `prod`, you could simply do it like this:

```
%dev.magman.manager.user=manager
%dev.magman.manager.password=mAnAggEr

%prod.magman.manager.user=admin
%prod.magman.manager.password=adm!n

magman.manager.user=admin
magman.manager.password=adm!n
```

Now when you start the app in the profile, named by `%{profile}`, the values in context will be applied to the application (unless you, of course, decide to use a higher priority setting). If your application is not running in any of the specified profiles, it will use the default ones, without a

profile. And if no default one were added, and no `defaultValue` is set to the `ConfigProperty` annotation, then you'll get a startup error, because no value of that property is provided.

Usually when you run your application with `mvn quarkus:dev` the default profile name that is loaded is `dev`. When you run tests, it is `test` and when you run a standalone jar, it will automatically set the profile `prod`.
To define a different profile, you will need to pass either system property `-Dquarkus.profile` or environment variable `QUARKUS_PROFILE`.

The same thing is valid for database connections and generation strategies, and anything Quarkus internal related.

When setting different database generation strategies for different profiles, you need to set `quarkus.profile` before compilation of the app, as Hibernate will need that ahead of time, otherwise it will use the default generation strategy that is set.

10.3.2. Passing config properties as system variables

This is even more straightforward. When you need to override the `application.properties` or environment file configurations for a compiled application, you need to simply pass the values with system arguments like so:

```
java -Dmagman.user=administrator -Dmagman.password=AdmInIstrAt9r -jar magman.jar
```

10.3.3. Passing config properties as environment variables

This is useful when you have automated containerized environments such as Docker, or you are using Terraform scripts or something that runs your applications automatically. Then you can set the application's properties as environment variables. Different operating systems do this differently, but most commonly for Windows you would do it by using the `SET` command and for Unix-based systems, you would use the `export` command.

So in our scenario, to configure those properties, you will have to do it like this.

On Windows:

```
SET MAGMAN_MANAGER_USER=admin
SET MAGMAN_MANAGER_PASSWORD=Adm!n
```

On Unix:

```
export MAGMAN_MANAGER_USER=admin
export MAGMAN_MANAGER_PASSWORD=Adm!n
```

When it comes to environment variables, all letters should be uppercase and all dots, and dashes

should be replaced with underscores. Quarkus will know how to handle and map those properties for you afterwards.

10.4. Config Properties additional info

You can pass most of the primitive types as config properties. So for example if you want to add a property as an integer, simply define an `int` variable inside the class and Quarkus will know how to convert it.

Configuration properties also support lists. So if you define a list of something, for example…

```
@Inject
@ConfigProperty(name = "names")
List<String> names;

@Inject
@ConfigProperty(name = "numbers")
Set<Integer> numbers;
```

Then you can specify the contents of these lits with comma-separated strings

```
names=John,George,Stephanie,Angela
numbers=1,2,3,4,5
```

Quarkus will be able to recognize the type of the properties, split and parse the numbers and names as expected.

10.5. Conclusion

In the next chapter, we are going to go through the next MicroProfile topic, RestClient, in attempt to build a small dummy payment service for our subscribers.

Chapter 11. MicroProfile REST Client

Most big web applications tend to be decentralized or rely on different third party servers in order to perform some task, not related to the application's core functionality. A good example are payment services. Not everybody is lucky to have a payments provider product in their company to charge their customers. And even if they did, the API for their payments system won't be implemented into every app they build. They will use a dedicated web application for payments, which they could reuse and sell out and is going to have some endpoints to perform payments.
This is how payment services, such as Stripe, PayPal or any banking API works. They have their own servers, host their own applications on the internet and publish their APIs (usually endpoints, but sometimes libraries for different languages) Then third party web servers, like us, can connect to them and perform financial operations.

In order to charge subscribers when they want to extend their subscription we need such payment provider too. But since this is a demo project for our course and we want to just demonstrate how REST client works, we are not going to integrate with the real thing, we are going to create our small dummy payment service.

11.1. Traditional REST communication with Java

As any other programming language, Java has support for REST communications out of the box. With the `java.net.http` package you can create any type of REST requests. A simple REST request in Java would look like this:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create("http://localhost:8080/hello"))
                .build();

HttpResponse<String> response = client.send(request,
                HttpResponse.BodyHandlers.ofString());
```

For small requests and small applications this is ok, but when you want to scale up, your application needs to connect to more and more services, establish more and more endpoints, this begins to turn counterintuitive. You will need to start following design patterns, create abstractions upon abstractions, so that the code looks clean and easy to use.

And here comes MicroProfile REST client. With it, you can build JAX-RS-like client endpoints and use CDI to inject them in your dependencies. It is time to build our small payments service and see how this will come in handy.

1. Create a small Quarkus application, called **"SpendPal"** (because spending means money, and we believe in it (non sponsored marketing campaign)).
2. Add just `quarkus-resteasy-jsonb`, `quarkus-resteasy` and `quarkus-hibernate-validator` as extensions, in order to expose endpoints to the outside world.
3. Create a single endpoint to charge the customer.

4. Don't forget to set the application to run on a different port to `8080`.

11.2. Integrating REST client into Magman

We are going to assume that your SpendPal endpoint looks like this.

```
@RequestScoped
@Path("/payment")
public class PaymentResource {

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    public ConfirmationDTO chargeCustomer(@Valid @NotNull CreditCardDTO creditCardDTO)
{
        return new ConfirmationDTO(true, LocalDateTime.now());
    }
}
```

Nothing complicated, just a simple endpoint that takes payment details and returns status response of the action.

Now that this is out of the way, it is time to prepare Magman for accessing `/payment` endpoint.

First, let's extend our application, by adding `quarkus-rest-client-jsonb` as an extension.

```
mvn quarkus:add-extension -Dextensions="io.quarkus:quarkus-rest-client-jsonb"
```

Next step is to create a new `spendpal` package and we'll add a new interface, called `SpendPalClient` and we are going to define a `chargeCustomer` method, similar to the one implemented inside SpendPal.

```
@Path("payment")
@RegisterRestClient ①
public interface SpendPalClient {

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    ConfirmationDTO chargeCustomer(CreditCardDTO creditCardDTO);

}
```

① With this annotation we are telling CDI that we are going to use this class as REST client.

As you can see it looks almost exactly as defining a JAX-RS endpoint. It is as simple as that. Now the next step is to configure the path and scope of that endpoint. By default these endpoints are of

scope `Singleton`, but if you'd like to change that, you can do it in `application.properties` file. Regarding the location of SpendPal's endpoints, we also need to define the path to the server. We can do it either inside `RegisterRestClient` annotation, or use the recommended approach and add property within `application.properties`

```
quarkus.rest-
client."com.vidasoft.magman.spendpal.SpendPalClient".url=http://localhost:8081
quarkus.rest-
client."com.vidasoft.magman.spendpal.SpendPalClient".scope=jakarta.inject.Singleton
```

In this scenario you'll need to exactly specify the path of the Client class. This should be enough to activate the rest client class inside CDI. Although it will work, it is not as flexible solution, when it comes to modifying these properties from system or environment perspective. And if you rename the interface, you'll have to change the name of the property here as well. Profile specific properties are also supported for REST clients, so you can specify different URLs for different environments, meaning that most of the time you'll have to change more than to occurrences of this property.
Good news is there is solution for that.

```
@RegisterRestClient(configKey = "spendpal-client")
public interface SpendPalClient {...}
```

With `configKey` you can give a nickname to your REST client interface, and make it easier to refer to, within the properties. Then your property file can look like this:

```
quarkus.rest-client.spendpal-client.url=http://localhost:8081
quarkus.rest-client.spendpal-client.scope=jakarta.inject.Singleton
```

Now our properties are shorter and easier to refer to.

Now if you want to configure the client path from outside application, you can:

- Pass is as system property like this: `-Dquarkus.rest-client.spendpal-client.url=http://localhost:8081`
- Set an environment variable like this: `QUARKUS_REST_CLIENT_SPENDPAL_CLIENT_URL=http://localhost:8081`

Just as we used to mention - replacing everything with underscores and capital letters.

After the interface's been configured, it is time to use it. Let's go to `PaymentService` and spice it up a little, shall we?

```
@ApplicationScoped
public class PaymentService {
```

```
    // definitions

    @Inject
    @RestClient①
    SpendPalClient spendPalClient;

    boolean chargeSubscriber(Subscriber subscriber) {
        if (subscriber.creditCard != null) {
            ConfirmationDTO paymentResult = spendPalClient.chargeCustomer(new
CreditCardDTO(subscriber.creditCard));
            LOGGER.log(Level.INFO, "Charging subscriber with id: {0}  and card type
{1} of number: {2}",
                    new Object[]{subscriber.id, subscriber.creditCard.creditCardType,
subscriber.creditCard.number});

            if (paymentResult.getSuccess()) {
                LOGGER.log(Level.INFO, "Successfully charged customer with id: {0}
and card type {1} of number: {2}",
                        new Object[]{subscriber.id, subscriber.creditCard
.creditCardType, subscriber.creditCard.number});
                onSubscriberCharged.fire(subscriber);

                return true;
            } else {
                LOGGER.log(Level.WARNING, "Unable to charge customer with id: {0}  and
card type {1} of number: {2}",
                        new Object[]{subscriber.id, subscriber.creditCard
.creditCardType, subscriber.creditCard.number});
                // Will probably email the customer, or, most likely, call the police.
                // Failed payments feel like federal crime after all ¯\_(ツ)_/¯

                return false;
            }
        }

        return false;
    }

}
```

① This is a producer endpoint which is going to tell CDI what instance to pass.

Now let's try our payment endpoint and see what happens

```
curl -i --location --request POST 'http://localhost:8080/subscription/7' ①
```

① We are going to assume that you already have subscribers in your database.

If your input data was correct and everything is configured properly, you should be getting a `204 NO`

`CONTENT` response and the following log.

```
2022-10-04 18:48:12,440 INFO  [com.vid.mag.sub.PaymentService] (executor-thread-0)
Charging subscriber with id: 7  and card type VISA of number: 3698521479456746
2022-10-04 18:48:12,441 INFO  [com.vid.mag.sub.PaymentService] (executor-thread-0)
Successfully charged customer with id: 7  and card type VISA of number:
3698521479456746
2022-10-04 18:48:12,442 INFO  [com.vid.mag.sub.SubscriptionExtensionHandler]
(executor-thread-0) Sent email to subscriber 7, about their subscription renewal.
2022-10-04 18:48:12,442 INFO  [com.vid.mag.sub.SubscriptionExtensionHandler]
(executor-thread-0) Extended subscription for user 7, till 2025-01-12
```

11.3. Handling errors in REST Client

Sometimes something might not go well. The data may you're passing to the REST client may be corrupted, something might be missing or the third party may be failing for whatever reason.
With the current configuration we do not have control over different than the expected responses of our REST client. This means that if we get `400 BAD REQUEST` or `500 INTERNAL SERVER ERROR` responses, our application will just throw an exception. You might have stumbled upon such errors, if you followed our examples from the beginning. In a real world scenario, you would most likely expect to encounter such scenarios and you'll need to prepare your application for it.

To start off, let's create a custom exception which we are going to handle within our code.

```
public class SpendPalException extends Exception {

    private int statusCode;
    private Object body;

    public SpendPallException(int statusCode, Object body) {
        this.statusCode = statusCode;
        this.body = body;
    }

    public int getStatusCode() {
        return statusCode;
    }

    public Object getBody() {
        return body;
    }
}
```

We prefer to extend the `Exception` class, instead of `RuntimeException`, as the developer will be required to handle it, if we add it to a method.

Next we need to create a mapper. This mapper is similar to JAX-RS exception mappers but it is

reversed. Instead of handling exceptions, it will throw exceptions.

```
public class SpendPalExceptionMapper implements ResponseExceptionMapper
<SpendPalException> {

    @Override
    public SpendPalException toThrowable(Response response) {
        return new SpendPallException(response.getStatus(), response.getEntity());
    }
}
```

The final step is to configure the interface to use this mapper.

```
@Path("payment")
@RegisterProvider(SpendPalExceptionMapper.class) ①
@RegisterRestClient(configKey = "spendpal-client")
public interface SpendPalClient {

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    ConfirmationDTO chargeCustomer(CreditCardDTO creditCardDTO) throws
SpendPalException; ②

}
```

① With `@RegisterProvider` annotation we tell CDI that there's an interceptor which is going to take the non expected responses to manage them.

② Here we add `throws` to make sure that all invocations of this method are handled properly.

After the refactoring we should have a payment resource method, looking like this:

```
@RequestScoped
@Path("subscription")
public class SubscriptionResource {

    @Inject
    PaymentService paymentService;

    public Response addPaymentMethod(@Positive @PathParam("userId") Long userId,
@Valid @NotNull CreditCardDTO creditCardDTO) {...} //not as interesting right now

    @POST
    @Path("{userId}")
    public Response chargeSubscriber(@Positive @PathParam("userId") Long userId) {
        Subscriber subscriber = Subscriber.findById(userId);
        if (subscriber != null) {
            try {
```

```
                var result = paymentService.chargeSubscriber(subscriber);
                return result ? Response.status(Response.Status.NO_CONTENT).build() :
                        Response.status(Response.Status.NOT_ACCEPTABLE).build();
            } catch (SpendPalException e) {
                return Response.status(e.getStatusCode()).entity(e.getBody()).build();
            }
        } else {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
    }
}
```

Go on! Give it a try. Create ot use a user with invalid credit card. Be a savage for a moment!

If your have configured everything properly, you should be getting a reflection of SpendPal's response.

You should not take this as a common practice, but as an example. The way you are going to handle responses in real life scenarios will be defined and decided by the project team you're working with.

11.4. Managing request headers in REST Client

So far we've spoken about the surface of REST endpoints, but what if you want ot modify the headers of the request. Most payment services will require some type of authorization token, to make sure that nobody is trying to steal money.

To begin with, let's add a header parameter to the payment endpoint, which will check the authority of the clients connecting to it.

```
@RequestScoped
@Path("/payment")
public class PaymentResource {

    public static final String VERY_SECURE_TOKEN = "mostSecureTokenEver";

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    public ConfirmationDTO chargeCustomer(@Valid @NotNull CreditCardDTO creditCardDTO,
@HeaderParam("authorization") String authorization) {
        if (VERY_SECURE_TOKEN.equals(authorization)) {
            return new ConfirmationDTO(true, LocalDateTime.now());
        } else {
            throw new UnauthorizedException();
        }
    }
}
```

Now every new request to that service will return `401 UNAUTHORIZED`, unless we provide `authorization` header to our requests.

There're two ways to set the headers of the request. The first approach would be, as you might guess, adding the header param to the rest client interface:

```
    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    ConfirmationDTO chargeCustomer(CreditCardDTO creditCardDTO, @HeaderParam(
"authorization") String authorization) throws SpendPalException; ①
```

This will require us to edit all occurrences where we call that method to support the newly added argument. If changes are small and seam reasonable, then why not, but if you'd like to add more complicated headers or a lot more headers, and you don't want to deal with this mess everywhere, there is another more abstract approach.

The second approach is to create client header factory, but keep in mind, that there can be only one such factory per class.
Create a class, called `SpendPalHeaderFactory`:

```
public class SpendPalHeaderFactory implements ClientHeadersFactory {

    @Inject
    @ConfigProperty(name = "spendpal.api.key")
    String spendpalApiKey;

    @Override
    public MultivaluedMap<String, String> update(MultivaluedMap<String, String>
multivaluedMap, MultivaluedMap<String, String> multivaluedMap1) {
        MultivaluedMap<String, String> result = new MultivaluedHashMap<>();
        result.add("Authorization", spendpalApiKey);
        return result;
    }
}
```

Afterwards, add a reference to the factory within the client interface:

```
// annotations which are already there
@RegisterClientHeaders(SpendPalHeaderFactory.class)
public interface SpendPalClient {

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    ConfirmationDTO chargeCustomer(CreditCardDTO creditCardDTO) throws
SpendPalException;
```

```
}
```

Now you should be able to perform payments again.
And finally a bonus approach. You can use the annotation `@ClientHeaderParam` for headers which are not susceptible to change.

11.5. What's next?

In the next chapter we are going to continue with MicroProfile specs and introduce you to JWT role-based access control. This will help us control the access of the endpoints of our application.

Chapter 12. Role-based access control with MicroProfile JWT

When you deal with web applications no matter, big or small, you would want to identify who is using the application and what they do with it. A typical approach to do this is with a web session. In the Java world a session is usually handled with a cookie, called "jsessionid". This cookie resembles a token, which maps to an object instance within the Java web application. Using that session token, we are able to identify users in our database, decide whether they should have access on a resource and what content should they receive.

There are some problems with approach. If the application gets restarted by a reason, the session will end, and the user will need to reauthenticate, so they can use the server again. Moreover the approach is not suitable for distributed systems. It works well for a single deployment of the application, but imagine if you had something more wide distributed, like Google. There are multiple Google servers all over the world, running the same code. And they exist, so that your queries can get to the nearest server, so that all the billion users get to have the best and fastest experience they could, instead of waiting millions of requests to be processed by a single server. In this kind of scenario one query can be processed by one server and another, by another server and all the servers will know who you are. Your session is everywhere.
Also since Quarkus is stateless and does manage sessions, we are not able to use that jsessoionid cookie to maintain user content and access. Fortunately, a more modern approach exists. This approach is being adopted by a lot of web services nowadays. And it's name is Json Web Token (JWT).

What is JWT?

Json Web Token is an open standard (RFC 7519), used to transfer claims between two parties. As the name suggests, JWT resembles JSON object and it consists of three parts:

- **Header** (algorithm and token type)

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

- **Payload** - combination of claims, user roles and other useful user data, such as issue time, expiration time and etc.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022
}
```

- **Signature** - generated signature, based on the algorithm and key/certificate. It is used to help parties verify the legitimacy of the claims. Think of it as the validity measures, placed on your id card.

At the end the key is encoded in base 64 and it looks like this

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4g
RG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

To find out more bout JWT, click here.

The payload of the token can be modified to store any data that is required for the web application to identify the user, although it is recommended to be brief. The more data you put inside your JWT, the bigger the request will get, which will cause slower processing.

12.1. Configuring Quarkus for JWT

A JWT is passed to the user upon successful login. The user makes a request to the server with their credentials and in response they get the token either in the response's header or inside the body. Once that token is obtained, the client should take it, and place it inside the `Authorization` header of every following request. MicroProfile JWT is capable to automatically parse that header, read its contents and validate the authority of the signature. With the annotations the specification provides, the developer is capable of then securing the application's endpoints however they please.

First thing we are going to do is prepare our project for JWT. Usually in applications, consisting of multiple microservices there will be one central service, responsible for issuing JWT tokens to the application. Keycloak is such an example. It provides multiple ways for users to authenticate, convenient role management and nice user interface to suit the needs of regular users.
In our project, though, we are not going to use centralized authentication service. We are going to implement something more simple, that is just going to suit our needs for the examples.

12.1.1. Adding JWT extension

The first step, as always, when it comes to something new, is to add extension.

```
mvn quarkus:add-extension -Dextensions="io.quarkus:quarkus-smallrye-jwt"
```

12.1.2. Creating SSL key pair

For this step you are required to have OpenSSL installed. If you are running on a Unix-based system, you probably already have it.

To sign our JWT tokens, we are going to need a pair of public and private keys, our application will be using. Usually the private key is needed for the issuer to sign the JWT. The public key is then

used by the web applications to verify the signature.

1. Create a folder called `jwt` somewhere in your system.
2. Create the private key

```
openssl genrsa -out /dir/to/private.pem 2048
```

3. Next step is to create the public key:

```
openssl rsa -in /dir/to/private.pem -pubout -outform PEM -out
/dir/to/public_key.pem
```

4. Finally we need a private PKCS8 key. This format is supported by Quarkus.

```
openssl pkcs8 -topk8 -inform PEM -in /dir/to/private.pem -out
/dir/to/private_key.pem -nocrypt
```

It's unnecessary to, but we should mention not to forget to set the `/dir/to/` part to your folder's directory, where you will store the keys.

Now that this is done, it is time to go to our `application.properties` file and point to the key's location.

```
mp.jwt.verify.privatekey.location=path/to/private_key.pem
mp.jwt.verify.publickey.location=path/to/public_key.pem
```

If you decide to store the keys in your application (to keep some default keys for development environments for example), you can do it inside `resources/META-INF` directory, and refer to it, by setting the value `META-INF/path/to/private_key.pem`

12.1.3. Configuring the issuing of JWT keys

Since JWT is normally issued by centralized service it is not part of the MicroProfile JWT spec to issue tokens. But for convenience we are gonna do it anyways.

Inside your `security` package, create a new class, called `JwtService`

```
@ApplicationScoped
public class JwtService {

    private static final Map<Class<? extends User>, String> ROLE_MAP = Map.of(
            Author.class, Author.ROLE_NAME,
            Subscriber.class, Subscriber.ROLE_NAME,
            Manager.class, Manager.ROLE_NAME
```

```
    );

    private PrivateKey privateKey;

    @Inject
    @ConfigProperty(name = "mp.jwt.verify.privatekey.location")
    String keyLocation;

    @PostConstruct
    public void initializePrivateKey() {
        try {
            privateKey = readPrivateKey();  ①
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public String generateJWT(User user, long expirationTime) {

        long currentTimeInSeconds = System.currentTimeMillis() / 1000;

        JwtClaimsBuilder claimsBuilder = Jwt.claims();
        claimsBuilder.issuer("http://localhost");
        claimsBuilder.upn(user.id + "");
        claimsBuilder.subject(ROLE_MAP.get(user.getClass())); ②
        claimsBuilder.groups(ROLE_MAP.get(user.getClass()));
        claimsBuilder.issuedAt(currentTimeInSeconds);
        claimsBuilder.expiresAt(currentTimeInSeconds + 1800); // 30 minutes ③

        claimsBuilder.claim(Claims.auth_time.name(), currentTimeInSeconds);

        return claimsBuilder.jws().sign(privateKey);
    }

    private PrivateKey readPrivateKey() {
        try (InputStream contentIS = getKeyStream(keyLocation)) {
            byte[] tmp = new byte[4096];
            try {
                int length = contentIS.read(tmp);
                return decodePrivateKey(new String(tmp, 0, length));
            } catch (Exception ex) {
                throw new RuntimeException("Could not read private key", ex);
            }
        } catch (Exception e) {
            return null;
        }
    }

    private InputStream getKeyStream(String keyLocation) throws IOException {
        var key = new File(keyLocation);
```

```
        return key.exists() ? new FileInputStream(key) :
                this.getClass().getClassLoader().getResourceAsStream(keyLocation);
    }


    private static PrivateKey decodePrivateKey(final String pemEncoded) throws
Exception{
        byte[] encodedBytes = toEncodedBytes(pemEncoded);

        PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec(encodedBytes);
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        return keyFactory.generatePrivate(keySpec);
    }

    private static byte[] toEncodedBytes(final String pemEncoded) {
        final String normalizedPem = removeBeginEnd(pemEncoded);
        return Base64.getDecoder().decode(normalizedPem);
    }

    private static String removeBeginEnd(String pem) {
        pem = pem.replaceAll("-----BEGIN (.*)-----", "");
        pem = pem.replaceAll("-----END (.*)----", "");
        pem = pem.replaceAll("\r\n", "");
        pem = pem.replaceAll("\n", "");
        return pem.trim();
    }

}
```

① As a start we need to manually parse the private key, from the file location we pointed in `application.properties`.

② Here we set the role both in `subject` and `groups`. Some clients may want to read it from either location.

③ Json Web Tokens are meant to be temporal, just like any user session. Just like cookies if the users are not careful, they might get their JWTs stolen. It is recommended that we keep sessions small, and ask the user to issue a new JWT. Thirty minutes is just about ok.

Other methods are just involved around reading the key and doing some encoding.

Next step is to invoke the method inside the `login` endpoint.

```
@RequestScoped
@Path("user")
public class UserResource {

    //some definitions

    @Inject
    JwtService jwtService;
```

```
    // some more methods


    @POST
    @Path("login")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response loginUser(@Valid @NotNull LoginDTO login) {
        Optional<User> loggedUser = userService.loginUser(login.getUserName(), login
.getPassword());
        return loggedUser
                .map(u -> Response.ok(new UserDTO(u))
                        .header("Authorization", jwtService.generateJWT(u)) // Don't
miss our jwt
                        .build())
                .orElseGet(() -> Response.status(Response.Status.UNAUTHORIZED).build(
));
    }

}
```

Now if we make login request, we should see our JWT in the response header.

```
curl -i --location --request POST 'localhost:8080/user/login' \
--header 'Content-Type: application/json' \
--data-raw '{
    "password": "manager",
    "userName": "manager"
}'
```

The response will be:

```
HTTP/1.1 200 OK
Authorization: eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL2xvY2FsaG9zd
CIsInVwbiI6Im1hbmFnZXJAdmlkYS1zb2Z0LmNvbSIsInN1YiI6Ik1BTkFHRVIiLCJpYXQiOjE2NjUwNzA2NDc
sImV4cCI6MTY2NTA3MjQ0NywiZ3JvdXBzIjpbIk1BTkFHRVIiXSwiYXV0aF90aW1lIjoxNjY1MDcwNjQ3LCJqd
GkiOiJhNWI0MDQzYS03YjA1LTRhNTktOWFiMS03YzM0MWI4MTMwZDUifQ.NLGCkTVAWjoRGxR40whId55NMQKL
mJVt3wvh-dliooHxkpXHcGBxa260QpMDJ_Aok6uSE2qS2r-POXyl7lsSvx_YkdTzEJEKrJlN-_9c6yqVYioa5
N9vaa83FYpyGHgxtzf67UnWTiDsM_A-_KhRrgN9f3uGaQCqhh4XIg3PbG2VGC2gVNTlhnFsyNBArC2igux00Z
iXeJrUFqaWskbN4nKVrv_ADXBVsL0GJj9A-ltqkbo8e-gOgRgS084IcAkHCOSbChTYTE4u9bNr65hGAgentZv
wIvr70SQvwlnY7JqXiYMTwln1acnamMUM5r4bz8p2s37pol9psZhOrdDUzg
Content-Type: application/json
content-length: 94

{"email":"manager@vida-soft.com","firstName":"Manager","lastName":"User","userName":
"manager"}
```

And if we take that key and decode it, we are going to see the following json.

```
{
    "typ": "JWT",
    "alg": "RS256"
}{
    "iss": "https://localhost",
    "upn": "manager@vida-soft.com",
    "sub": "MANAGER",
    "iat": 1665070647,
    "exp": 1665072447,
    "groups": [
        "MANAGER"
    ],
    "auth_time": 1665070647,
    "jti": "a5b4043a-7b05-4a59-9ab1-7c341b8130d5"
}□K□)□T□□G0y□(V□*(□□)\w□□□D)02□□:!6-^+□O1    □□□βV"MqX!s□□d?□□7f
ar QMNXg□□
(.M□Ajik$l')Z□□[□bc□□<□pIM[
        6` z{Y□B%□x1<%VPkk7%□
L
```


The characters at the end resemble the signature, we did with our private key.

12.2. Using MP JWT API

Now since we are able to issue JWT tokens, we can move onto setting protection to our endpoints. The first annotation we are going to look at is `@Authenticated`.

12.2.1. The `@Authenticated` annotation

This annotation works both on class and on method level. By default adding the JWT extension does not protect your endpoints from access. You wouldn't have been able to use the login endpoint if they were. When you want to protect a single/all endpoints of a resource class, without limiting the users' access to it, you can just use the `@Authorized` annotation. This will make the access to an endpoint restricted only for the non-authorized users.

To test this, let's place the annotation on `ArticleResource` and try to get all articles.

```
curl -i --location --request GET 'http://localhost:8080/article/'
```

You will immediately get a response:

```
HTTP/1.1 401 Unauthorized
www-authenticate: Bearer
content-length: 0
```

The call won't even enter the method. Since you already know how interceptors work, you get the

idea how this worked.

Now let's try again, by getting our authentication token and build a query, that will let us view all articles

```
curl -i --location --request GET 'http://localhost:8080/article/' \
--header 'Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL2xvY2FsaG9zdCIsInVwbiI6Im1hb
mFnZXJAdmlkYS1zb2Z0LmNvbSIsInN1YiI6Ik1BTkFHRVIiLCJpYXQiOjE2NjUwNzA2NDcsImV4cCI6MTY2NTA
3MjQ0NywiZ3JvdXBzIjpbIk1BTkFHRVIiXSwiYXV0aF90aW1lIjoxNjY1MDcwNjQ3LCJqdGkiOiJhNWI0MDQzY
S03YjA1LTRhNTktOWFiMS03YzM0MWI4MTMwZDUifQ.NLGCkTVAWjoRGxR40whId55NMQKLmJVt3wvh-
dliooHxkpXHcGBxa260QpMDJ_Aok6uSE2qS2r-POXyl7lsSvx_YkdTzEJEKrJlN-
_9c6yqVYioa5N9vaa83FYpyGHgxtzf67UnWTiDsM_A-
_KhRrgN9f3uGaQCqhh4XIg3PbG2VGC2gVNTlhnFsyNBArC2igux00ZiXeJrUFqaWskbN4nKVrv_ADXBVsL0GJj
9A-ltqkbo8e-
gOgRgS084IcAkHCOSbChTYTE4u9bNr65hGAgentZvwIvr70SQvwlnY7JqXiYMTwln1acnamMUM5r4bz8p2s37p
ol9psZhOrdDUzg'
```

Using the `Bearer` word, before passing the token is mandatory. The authorization header can take different types of authorization tokens, and specifying the type of authentication before passing the token is needed. If you miss it, the authentication will not work.

12.2.2. The `@RolesAllowed` annotation

This annotation will limit the access to users with certain role. It could be placed on a method or a class level. Placing this on a method or a class, fully replaces the `@Authenticated` annotation. Just as the `@Authenticated` 's documentation states - *"Indicates that a resource can only be accessed by a logged in user. This is equivalent to the Jakarta EE8 RolesAllowed("**") construct."*

For our ArticleResource we want articles to only be created/edited by Authors, deleted by Authors and Managers, and the manager should be able to control the advertisers on an article. To achieve this behavior, we need to do it like this:

```
@Authenticated
@RequestScoped
@Path("/article")
public class ArticleResource {

    @Inject
    ArticleService articleService;

    @POST
    @Transactional
    @RolesAllowed({Author.ROLE_NAME})
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createArticle(@Valid @NotNull ArticleDTO articleDTO) {...}
```

```
    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getArticle(@Positive @PathParam("id") Long articleId) {...}

    @PUT
    @Transactional
    @Path("/{id}")
    @RolesAllowed({Author.ROLE_NAME})
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response editArticle(@Positive @PathParam("id") Long articleId, @Valid
@NotNull ArticleDTO articleDTO) {...}

    @DELETE
    @Transactional
    @Path("/{id}")
    @RolesAllowed({Author.ROLE_NAME, Manager.ROLE_NAME})
    public void deleteArticle(@Positive @PathParam("id") Long articleId) {...}

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getArticles(@QueryParam("page") @DefaultValue("1") @Positive int
page,
                                @QueryParam("size") @DefaultValue("10") @Positive int
size,
                                @QueryParam("author") @Positive Long authorId) {...}

    @PATCH
    @Transactional
    @RolesAllowed({Manager.ROLE_NAME})
    @Path("{id}/advertiser/{advertiserId}")
    public Response addAdvertiserToArticle(@Positive @PathParam("id") Long id,
@Positive @PathParam("advertiserId") Long advertiserId) {...}
}
```

We are going to leave the `@GET` methods alone, as they need to be accessed by all users. Now if we try to create an article with the manager or any other than author user, we are going to get:

```
HTTP/1.1 403 Forbidden
content-length: 0
```

If you use the right roles required for the annotations, you are going to get access to the endpoint. If not, you are never going to be able to access them.

12.3. Getting data from the JWT

Sometimes restricting the access by role is not enough. For example we do not want one author to

edit the article of another author. Also when an author creates an article, we want to identify them, without making them do it, by passing the `authorId` in the DTO. We can already get that from the token.

Let's make a `@PostConstruct` method which is going to help us identify the user, before going to the article methods.

```
public class ArticleResource {

    @Inject
    JsonWebToken jwt; ①

    User loggedUser;

    @PostConstruct
    void init() {
        Long userId = Long.parseLong(jwt.getClaim("upn")); ②
        loggedUser = User.findById(userId); ③
    }

    // sweet, sweet endpoints ahead
}
```

① Just like everything CDI-ish, we can inject an instance of the `JsonWebToken` class, which holds the user's token deconstructed.

② Then from that token, we can take the user's id from the `upn` claim. This is where we previously contained the user id, when we made the claim issuing code. The `upm` claim is the standard place, where you will place identification details about the user.

③ Since `upn` claim takes only string as a value, then we need to parse it back to long and use it in a query.

Next step is to update the methods which require to check the user that is performing the operation.

```
public class ArticleResource {

    public Response createArticle(@Valid @NotNull ArticleDTO articleDTO) {
        Author author = (Author) loggedUser;
        if (author == null) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        } else {
            Article article = articleService.createArticle(articleDTO.getTitle(),
articleDTO.getContent(), author);
            return Response.created(URI.create(String.format("article/%s", article.id
))).build();
        }
    }
```

```
    public Response editArticle(@Positive @PathParam("id") Long articleId, @Valid
@NotNull ArticleDTO articleDTO) {
        Article article = Article.findById(articleId);

        if (article == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        } else if (!loggedUser.equals(article.author)) { ①
            return Response.status(Response.Status.FORBIDDEN).build();
        } else {
            articleService.editArticle(article, articleDTO.getTitle(), articleDTO
.getContent());
            return Response.ok(new ArticleDTO(article)).build();
        }
    }

    @DELETE
    @Transactional
    @Path("/{id}")
    @RolesAllowed({Author.ROLE_NAME, Manager.ROLE_NAME})
    public void deleteArticle(@Positive @PathParam("id") Long articleId) {
        Article.delete(articleId, loggedUser.id); ②
    }
}
```

① Assuming AbstractEntity.equals is overridden

② Here we have added a named query into `Article` entity, to delete an article where the author's id matches the one of the logged user or if the user is a manager: `delete from Article a where a.id = :articleId and (a.author.id = :userId or (select count (m) from Manager m where m.id = :userId) > 0)`. Don't forget to surround the part starting from `author.id` with brackets, otherwise you are going to delete all articles when you do it with the manager.

You should also create methods to delete all related to the article comments, beforehand.

12.4. What can I do now?

Now you have all the basis of RBAC with MP JWT. Why don't you get a bit wild, and start protecting all of your endpoints in a way that it would make sense. You could also create a producer for the logged user, to avoid making the same `@PostConstruct` method everywhere when you need it.

Chapter 13. Reactive messaging with Kafka

At this moment we have learned how two applications can communicate with each-other using the REST interfaces offered by RestEasy. The HTTP REST protocol works perfectly when the client wants to pass a request and is waiting for a response immediately after the request was sent. In a small and controlled environment like our project structure here, this will happen lightning fast. You send a request and you get a response under a second.

In a real environments though, we don't usually tend to get things done so quickly. To make the user experience more seamless, we pretend that things have happened, before they do.

A.N.(AmazoN) example

Imagine you are browsing through a shopping catalog in Amazon and you see something that you want to buy. Sometimes Amazon will give you the option to purchase the item you're looking at with a single click, without having to verify your payment and shipping information. Other times it will ask you explicitly provide this data or at least confirm it. You click the check-out button, wait just 2-3 seconds, and BOOM! Order is completed.

> But has it really been completed?

The answer is no. To keep the user from waiting, Amazon will do some tricks under the hood. First it will show that the purchase has been completed, but it is not really completed it is just registered as pending. Then all the associated servers start doing their thing to check if the item is still available, if your shipping data is correct and contact your bank to withdraw the money from your account. And Amazon is doing this process fpr thousands of purchases simultaneously.

Depending on REST services in such a scenario is inefficient, as it requires for every customer to wait until their query has been processed by every system that is connected to your purchase. Imagine having to wait on the huge line just to buy a ticket to watch a movie, and the movie has already started. This would be very bad user experience. Nobody would visit that cinema again right?

To prevent such a scenario from happening Amazon is using some type of messaging services. Let's see how the messaging services could mitigate the process of purchasing an item:

1. The customer checks out their purchase and completes the order.
2. A REST call containing information about the order has been received inside an Amazon server.
3. The server sends out an OK response to the front-end system to tell the customer that the order was completed.
4. Asynchronously the server sends out a message to the various distributed services within Amazon about the purchase:
    a. One reads that message and saves the order in the customer's order history

b. Another message goes to the delivery department to notify the systems that an order has to be processed

c. A third system will take the customer's payment details and conduct a payment transaction with the payment provider of choice

d. Finally when all the processes are completed another system will send the user an email about the completion state of the order.

In this scenario if the payment and shipment information has been processed correctly, the user will receive an email, stating that the order is completed and the items are being processed. If something fails, the customer will be notified through email or through the app/website to re-submit their details. In the end this is creating a seamless flow, where the customer's experience, that seems smooth, they can continue shopping or do other stuff, while the order is being processed. In most scenarios there even shouldn't be a problem with the payment and the order for recurring customers, so they for example can benefit from purchasing items without having to wait for the system to process everything.

13.1. So what are messaging services?

Imagine a group chat with various participants but for web application services. Instead of each service having to call one-another to exchange information, they will write to the group chat. Let's use our aforementioned example and imagine it has three services:

- A "checkout" service that processes every order
- A "payments" service that will contact payment providers and charge the customer
- A "mailing" service that will collect various results from the services above and send emails to the customer

In a big platform like Amazon such a decentralization of services is inevitable, if they want to alleviate the high load of requests happening every second and better maintain the services' functionality. In most scenarios all the services will communicate using some kind of messaging system. Usually messaging systems like human-made ones have so-called channels, threads or topics, resembling groups, and participants, resembling the services themselves. In our example the participants are our services, and they can write and read messages from various channels or various actions upon reading a message.

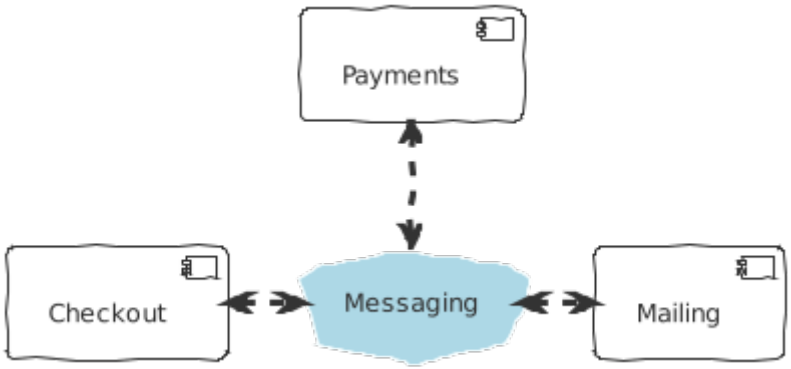


Figure 2. This image displays a very simplified version of how applications can communicate through a messaging service.

In a nutshell when an order is confirmed by the Checkout service, it will send a message to the Messaging service in a "group" called "orders". In that same group "Mailing" and "Payments" are also participants. They will receive the message and act accordingly. The payments service will call the payment provider using the customer's payment details and the Mailing service will assemble an email, confirming the customer's order and send it through an SMTP protocol.

Once the payment has been processed, if something fails, there might be another group chat (topic), called "Payment Failures" where Payments and Mailing participate.

To keep it simple, such a scenario is not found in the diagrams. The term for this is known as Dead Letter Queue.

If "Payments" writes to this topic, all other participants, such as Mailing, will read from it, and in the case of Mailing, it will assemble a an email stating what is wrong with the payments.

In all the stated scenarios, just like in real life, the service is not obligated to wait for the response of the other services. They just write the message to the topic and move on to the next action. The Checkout service does not need to know if the user's payment has gone through. The customer needs to know. Therefore, it is more convenient to notify them through email, instead of waiting for their details to be processed in a queue with dozen other order queries.

13.2. We can't send a message without a messenger

There are a lot of protocols and implementations of messaging services around the internet. All of them might work differently, but in their core, they serve the same purpose - an application writes to a topic, where other applications can read from and act upon receiving the message.

As mentioned, messages can be transmitted and received through various protocols and services, most of which are using TCP, AMQP and MQTT. Depending on the type and uses for the applications, we might want to implement different types of protocols for different scenarios. TCP and AMQP messaging services for example are more advanced than MQTT, whereas MQTT is lightweight and allows for quick communication between IoT devices, which are small servers on their own.

For the purpose of our course we are going to look through one of the most popular messaging service implementations, called Apache Kafka. Kafka is using the TCP protocol to allow

communication between its components and participants. In the following section you'll be able to learn more about Kafka in a summary.

What is Apache Kafka?

Apache Kafka (or Kafka for short) is an open-source distributed event streaming platform designed for handling real-time data feeds. It facilitates the seamless flow of data between different systems, enabling efficient communication in distributed architectures.

In other words, Kafka allows for two applications to communicate through so-called topics, where one application writes to the topic an the other reads from it.

Kafka works, through the following principles:

1. **Push-Subscribe Model:**
    - Kafka follows a publish-subscribe model, where data is published to topics by producers.
    - Consumers subscribe to specific topics to receive and process the published data.
2. **Topics and Partitions:**
    - Data is organized into topics, acting as channels for communication.
    - Topics are further divided into partitions to enable parallel processing and scalability.
3. **Producers and Consumers:**
    - **Producers:** Applications that send data to Kafka topics.
    - **Consumers:** Applications that subscribe to topics and process the data.
4. **Broker Architecture:**
    - Kafka operates with a distributed architecture consisting of multiple servers called brokers.
    - Brokers store and manage the data, ensuring fault tolerance and high availability.

To be able to work with Kafka, we nee to learn the names and purpose of each participant component:

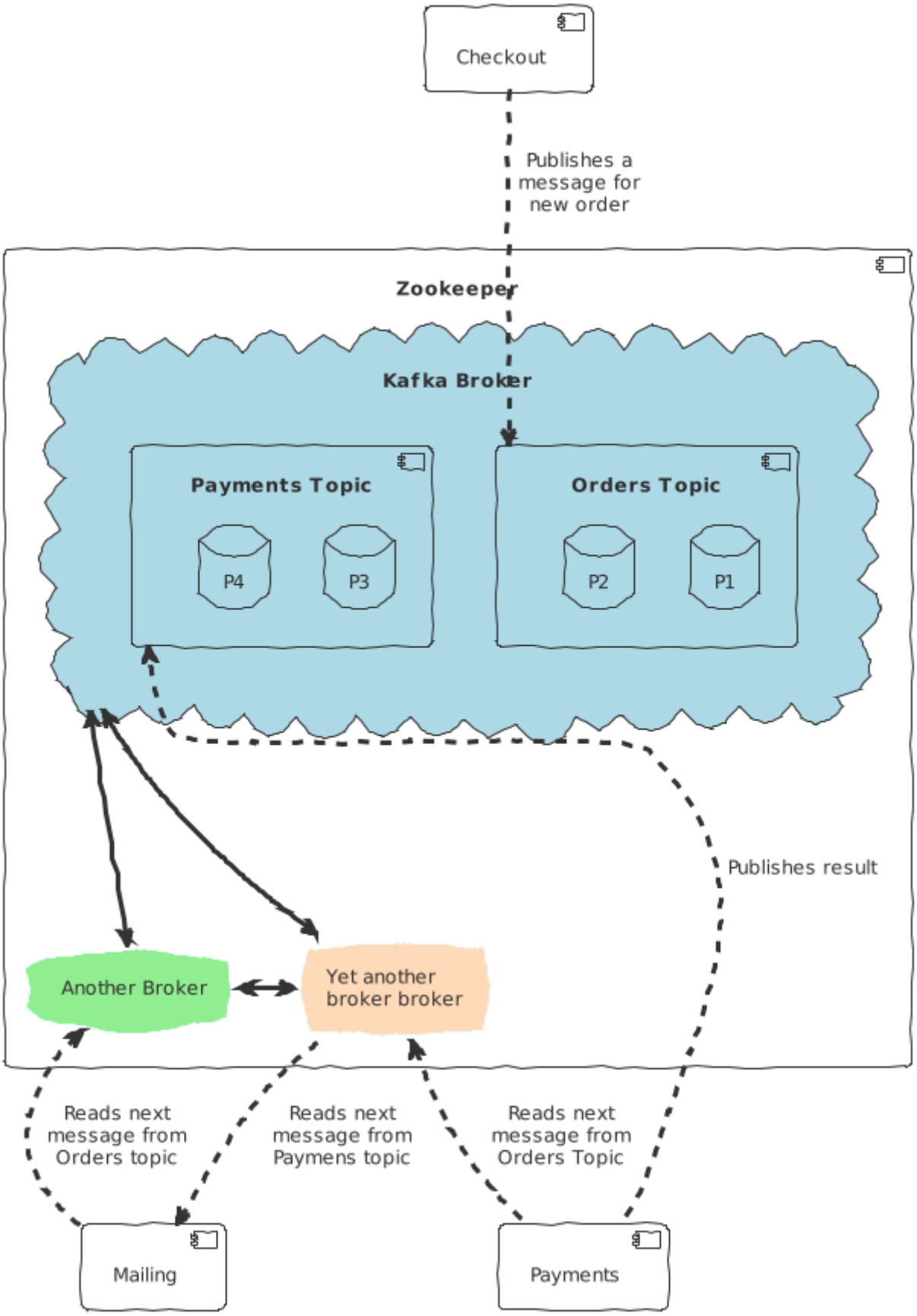
1. **Producer:**
    - Responsible for publishing data to Kafka topics.
    - Ensures the delivery of messages to the specified topics.
2. **Consumer:**
    - Subscribes to topics to receive and process data.
    - Can be part of a consumer group for load balancing and fault tolerance.
3. **Broker:**
    - Kafka servers that store and manage data.
    - Each broker in a cluster is aware of the data distribution and can serve as a leader or

follower for partitions.

4. **Topic:**
    - A logical channel for data streams.
    - Data is organized into topics, and each topic can have multiple partitions.
5. **Partition:**
    - Divides a topic into smaller, independently manageable segments.
    - Enables parallel processing and scalable data consumption.
6. **Zookeeper:**
    - Coordinates and manages the Kafka brokers in a distributed setup.
    - Maintains configuration information, leader election, and synchronization.

In summary, Apache Kafka simplifies real-time data streaming by providing a robust infrastructure for handling large-scale, distributed data flows among different components and systems.

After understanding what Kafka is, let’s see how our Amazon application participants would use it in a real-life scenario



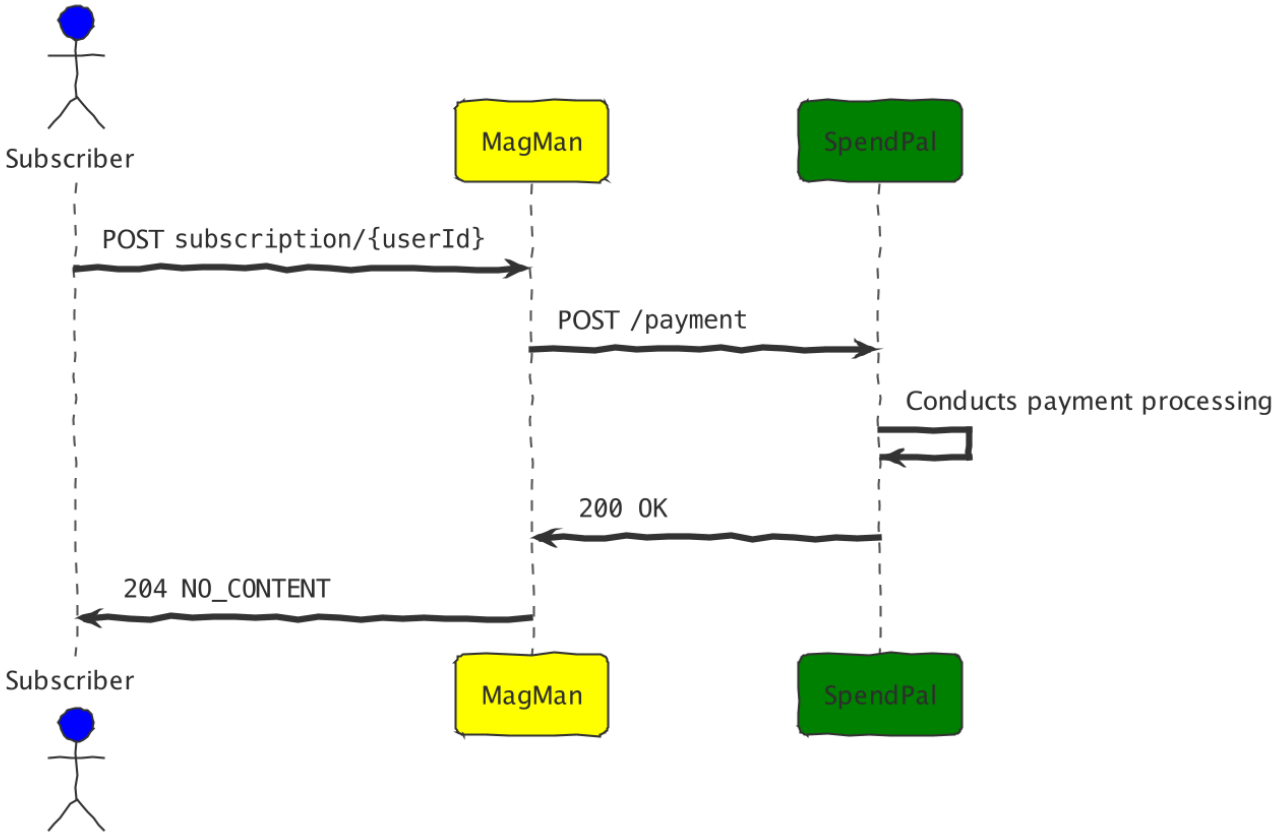
The image shows a hypothetical scenario how Kafka could work as a messaging service with Amazon's services. A zookeeper may contain more than one broker and each service would read from/write to the first available broker it gets connected to.

To get a better understanding of Apache Kafka, please refer to the official website: https://kafka.apache.org

13.3. Applying Kafka to our project

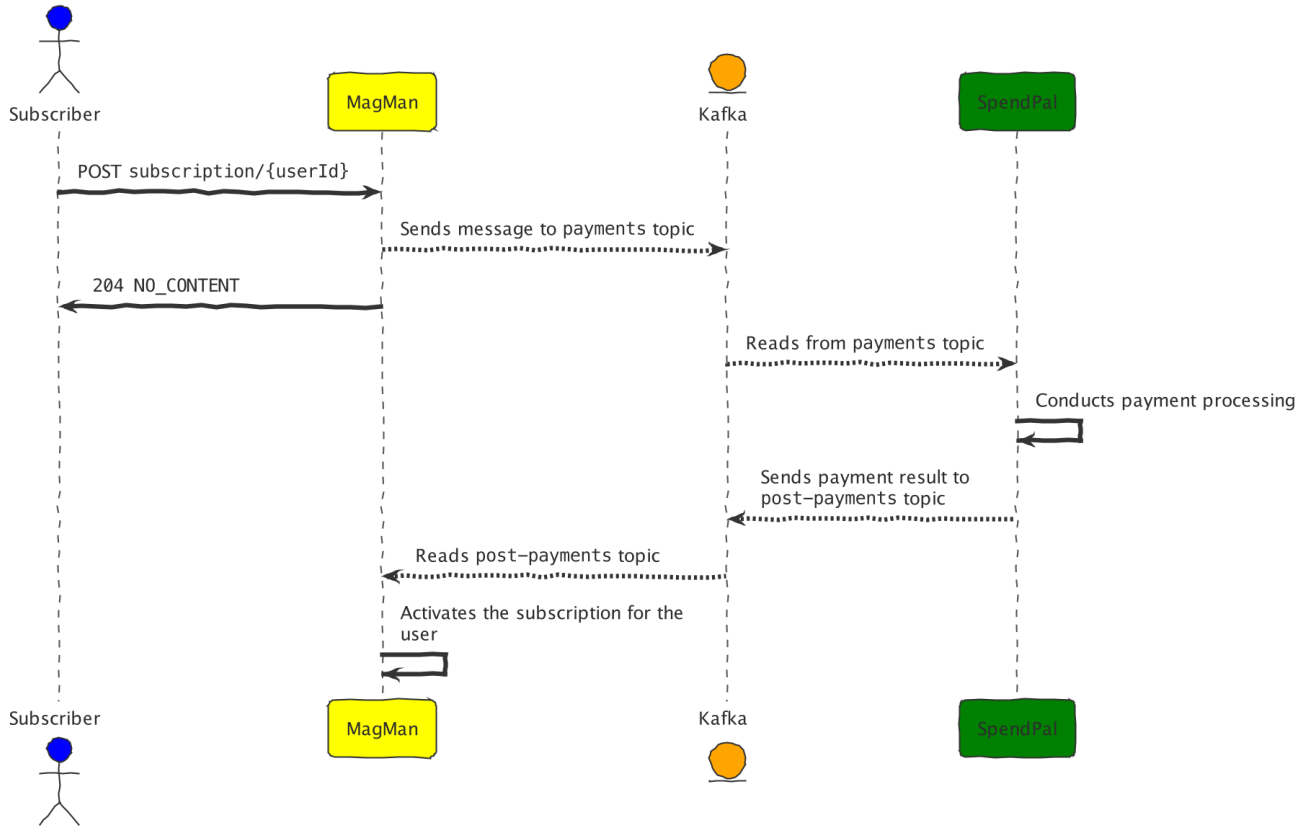
Now that we know what Kafka is and how it works, let's see how it could be applied to our project.

Currently, we have created our MagMan project for the Magazine Manager and SpendPal, that is responsible for charging our customers for the subscription services. The way it works is the MagMan performs a REST call to SpendPal every time we want to charge a customer. Let's visualize how this flow works right now.



As we can see from our sophisticated diagram, the process requires for the subscriber to wait for SpendPal to process the payment request, before getting a response. Just as mentioned earlier, this process wouldn't take that long if there were a couple of customers to pay for this feature, but we're using the power of distributed web applications for way bigger scenarios. Imagine if the customer was not just one, but hundreds or even thousands of them. This is the point where REST services would bottleneck the user experience, as each payment will have to wait in line to be processed and the customer will not be able to use the platform meanwhile.

Now let's see how our diagram would look if we add the Kafka infrastructure to our service…



Now regardless if the diagram looks a bit more intimidating, if we follow along we can see, that:

1. The user makes a payment providing their payment details
2. MagMan sends a message to a Kafka topic to notify all its listeners that a payment has occurred
3. The user gets the confirmation from MagMan and can move onto using the site
4. The rest of the process is managed in the background without the subscriber's knowledge, i.e. the payment gets processed and the SpendPal service returns a response message in a separate dedicated topic, which is read by MagMan
5. If the result of the payment was successful, we can allow the user to continue using the site, without any interventions, if not, we might trigger a mechanism to stop the user from using the website and ask them to provide payment information again.

With messaging systems we want to implement one way communication as each message payload is unique to the dedicated topic. We also do not want to double read a topic when we publish something in it. This is the reason behind having two topics to write and read from.

- The `payments` topic is designed to be read only by SpendPal service. It will contain data regarding payment information, such as credit card info, subscription type and so on.
- The `post-payments` topic is designed to be read by MagMan. It will contain, as the name suggests, post payment information, such as the payment status, timestamps, any error messages and so on.

Both topics are read and written to in a queue (FIFO) manner, meaning that we read the messages from top to bottom and every new message will be read and processed when the previous message has completed processing. Once the message was read, the Kafka broker will remember which messages were read and will provide the subscriber applications only with the unread messages.

13.4. Setting up Kafka for our services

Before we mess up with our project we will first need to set up our Kafka infrastructure. As mentioned in the previous chapters Kafka consists of many server components that need to be hosted somewhere in order for our applications to be able to connect, write and read messages.

Setting up Kafka manually is a bit complicated and requires some configurations, network adjustments and so on. Thankfully there is the thing called Docker. And this is the perfect time to learn what it is and how to install it on your machine. If you follow the link above, the process should be quite straight-forward.

Although we are going to use Docker for simplicity, there's nothing stopping you from going wild and trying to configure a Kafka broker yourself. Fell free to go through these steps in a way that is comfortable to you.

Installing Docker on some operating systems like Ubuntu, might force you to install a version from their official app store, known as Snapcraft. Installing Docker form there might not always give the desired results, when it comes to user permissions and access for the application to system resources. You might also not be gating the latest version of Docker, as these apps seem to not be maintained officially by the vendor. If you experience such issues, please make sure to delete all Snapcraft installations of Docker and install the official version from the Docker website.

Now that we have an idea how to set up the Kafka environment, the following steps will concentrate on using Docker for the set-up. The easiest way to get our Kafka docker container up and running is by creating a `docker-compose` script. If you have installed Docker properly, you should be able to invoke `docker-compose` from your terminal.

The next step is to create a docker-compose file. To do so, simply create a file, called `docker-compose.yml` into your project folder and place the following content inside:

```
version: '3'
services:

  zookeeper: ①
    image: confluentinc/cp-zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  kafka: ②
    image: confluentinc/cp-kafka
    depends_on:
      - zookeeper
    ports:
      - '9092:9092'
    environment:
      KAFKA_BROKER_ID: 1
```

```
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1

  create-topics: ③
    image: confluentinc/cp-kafka
    depends_on:
      - kafka
    entrypoint: [ '/bin/sh', '-c' ]
    command: |
      "
      # blocks until kafka is reachable
      kafka-topics --bootstrap-server kafka:29092 --list

      echo -e 'Creating kafka topics'
      kafka-topics --bootstrap-server kafka:29092 --create --if-not-exists --topic
payments --replication-factor 1 --partitions 1
      kafka-topics --bootstrap-server kafka:29092 --create --if-not-exists --topic
post-payments --replication-factor 1 --partitions 1

      echo -e 'Successfully created the following topics:'
      kafka-topics --bootstrap-server kafka:29092 --list
      "
```

① First we need a Zookeeper where our Kafka instance will live in

② Next is the Kafka server (the Broker), which needs to know where the Zookeeper is, in order for it to work

③ Finally this is a single time run script, that will create the topics where our publishers and subscribers will read and write to.

And now it's time to run a docker-compose script.

1. Open a terminal window inside the project folder or where you left the `docker-compose.yml` file.
2. Write the command `docker-compose up` and press enter
3. Some logs will appear. When the logs stop you should be able to see the following result within:

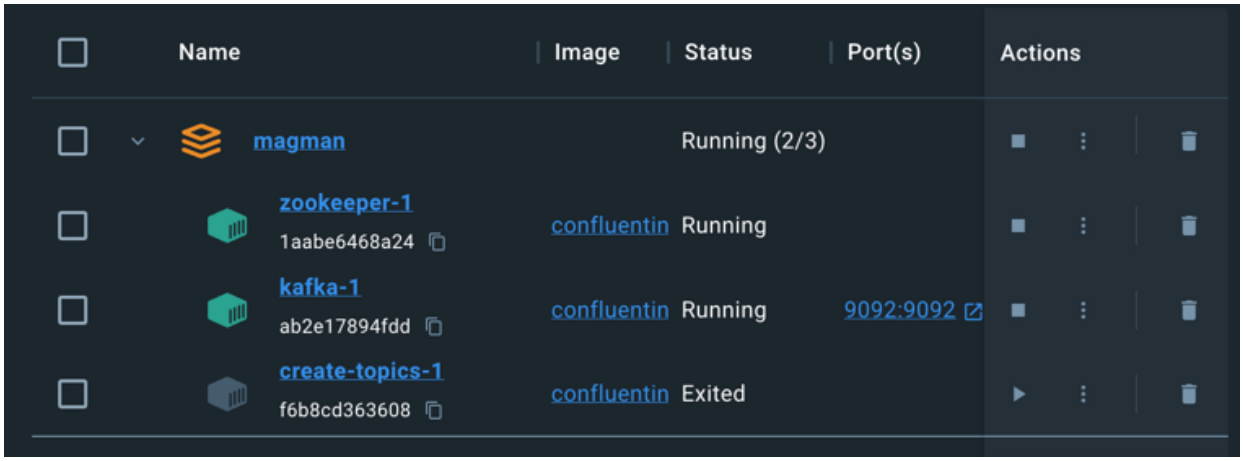
```
magman-create-topics-1  | Created topic post-payments.
magman-create-topics-1  | Successfully created the following topics:
magman-create-topics-1  | payments
magman-create-topics-1  | post-payments
magman-create-topics-1 exited with code 0
```

4. Now let's check that everything is fine and the containers are running. There are two ways to check that

a. Open another terminal window and write the command `docker container ls`. You should be seeing exactly two running containers

```
CONTAINER ID   IMAGE                          COMMAND                  CREATED          STATUS          PORTS                    NAMES
ab2e17894fdd   confluentinc/cp-kafka          "/etc/confluent/dock…"   6 minutes ago    Up 6 minutes    0.0.0.0:9092->9092/tcp   magman-kafka-1
1aabe6468a24   confluentinc/cp-zookeeper      "/etc/confluent/dock…"   15 minutes ago   Up 6 minutes    2181/tcp, 2888/tcp, 3888/tcp   magman-zookeeper-1
```

b. If you have installed Docker Desktop, you should see the two running containers in the containers tab as well



Don't worry that the `create-topics` container is down. Its purpose was to create the required topics and shut down. All we need to do now is implement the Producer and Subscriber logic in out applications.

Having our Kafka server set up, it is time to configure our applications to support Kafka.

1. Go to the `pom.xml` file of each application and add a new Quarkus extension:

```
<dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-smallrye-reactive-messaging-kafka</artifactId>
</dependency>
```

2. Go to each app's `application.properties` file and add the respective properties to enable the app to read and write messages:

   a. For MagMan

```
kafka.bootstrap.servers=localhost:9092

mp.messaging.outgoing.payments.connector=smallrye-kafka
mp.messaging.outgoing.payments.topic=payments

mp.messaging.incoming.post-payments.connector=smallrye-kafka
mp.messaging.incoming.post-payments.topic=post-payments
mp.messaging.incoming.post-payments.group.id=${quarkus.uuid}
```

   b. For SpendPal

```
kafka.bootstrap.servers=localhost:9092

mp.messaging.incoming.payments.connector=smallrye-kafka
mp.messaging.incoming.payments.topic=payments
mp.messaging.incoming.payments.group.id=${quarkus.uuid}

mp.messaging.outgoing.post-payments.connector=smallrye-kafka
mp.messaging.outgoing.post-payments.topic=post-payments
```

Notice that the configurations look the same, but the state of `incoming` /`outgoing` is inverted. This is reflecting our will to make MagMan only write to the `payments` topic and read from `post-payments` topic. The same goes for SpendPal inverse.

3. The final thing we need to do in order to have things up and running is to define a class that will handle messages.
    a. Create a new package and class in each respective project, called `kafka.KafkaMessageService` or something that feels closer to your heart. The point here is to have a dedicated class for this to keep our code structure clear
4. For MagMan in this class now you can define the following methods:

```
@ApplicationScoped
public class KafkaMessageService {

    @Inject
    @Channel("payments")
    Emitter<String> paymentsEmitter;

    @Incoming("post-payments")
    public void consumePostPaymentMessage(String message) {

    }

    public void sendPaymentsMessage(PaymentPayload payload) {

    }

}

//Where Payments payload can be a record or POJO in a separate class object

public record PaymentPayload(String username, CreditCardDTO creditCardDTO) {
}
```

5. Do the same shenanigan in SpendPal, but inversed.
6. Now when you try to run the projects you should be able to see the following logs:

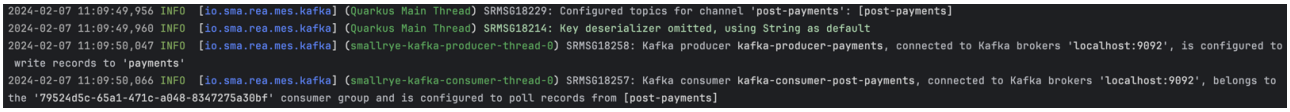


Figure 3. The log messages here signify that our applications are talking to the configured topics and are ready to consume and produce messages.

13.5. Implementing the messaging service in our project

Now that we have set up communication between our two applications through Kafka, we need to refactor the code so that they can actively use channels of communication we established.

For now, we won't get rid of the REST communication between the two. We would want a fallback mechanism in case the communication with Kafka seizes to work. Let's first implement our producer and consumer logic in each respective project…

In MagMan we agreed that we want to publish messages through the `payments` channel and consume messages from SpendPal. To do so, we will need to convert the `PaymentsPayload` record into a JSON string, which is going to be transmitted as our message to Kafka and then we want to use our injected `paymentsEmitter` object to send the message.

```
public void sendPaymentsMessage(PaymentPayload payload) {
    String payloadString = JsonbBuilder.create().toJson(payload); ①
    paymentsEmitter.send(payloadString).toCompletableFuture().join(); ②
    LOGGER.info("Successfully emitted message to payments topic: %s".formatted
(payloadString)); ③
}
```

① We use Jsonb's serialization capabilities to convert our Java object into the expected output format which is of type `String`.

② For now we need to make the emitter synchronous as it may cause the `@Transactional` scope of invoking methods to leak in the asynchronous thread, which will throw an exception for establishing connection without transaction.

③ We're placing this log, as this operation will happen behind the scenes and we want to confirm that what we sent is what we expected. **Please do not log credit card information in your real-world applications!**

The next step will be to implement the consumer logic within SpendPal.

```
@Incoming("payments")
public void consumePostPaymentMessage(String message) {
    PaymentPayload payload = JsonbBuilder.create().fromJson(message, PaymentPayload
.class);
    LOGGER.info("Received message with payload: %s".formatted(message));
}
```

To keep things simple, we're not going to do anything with the payload. We're just going to track the logs and see that messages are sent and received correctly.

Now that we have built the handling of the `payments` topic, we can sneak in the message sending method into our Payment service.

```
@ApplicationScoped
public class PaymentService {

    ...

    @Inject
    KafkaMessageService kafkaMessageService;

    boolean chargeSubscriber(Subscriber subscriber) throws SpendPalException {
        if (subscriber.creditCard != null) {
            CreditCardDTO creditCardDTO = new CreditCardDTO(subscriber.creditCard);
            try {
                kafkaMessageService.sendPaymentsMessage(new PaymentPayload(subscriber
.userName, creditCardDTO)); ①
                return true;
            } catch (Exception e) { ②
                LOGGER.severe(e.getMessage());
                ConfirmationDTO paymentResult = spendPalClient.chargeCustomer(new
CreditCardDTO(subscriber.creditCard));
                LOGGER.log(Level.INFO, "Charging subscriber with id: {0}  and card
type {1} of number: {2}",
                        new Object[]{subscriber.id, subscriber.creditCard
.creditCardType, subscriber.creditCard.number});

                if (paymentResult.getSuccess()) {
                    LOGGER.log(Level.INFO, "Successfully charged customer with id: {0}
and card type {1} of number: {2}",
                            new Object[]{subscriber.id, subscriber.creditCard
.creditCardType, subscriber.creditCard.number});
                    onSubscriberCharged.fire(subscriber);
                    return true;

                } else {
                    LOGGER.log(Level.WARNING, "Unable to charge customer with id: {0}
and card type {1} of number: {2}",
                            new Object[]{subscriber.id, subscriber.creditCard
.creditCardType, subscriber.creditCard.number});

                    return false;
                }
            }
        }

        return false;
```

```
    }

}
```

① Since we are going to trust on both systems doing their thing in the background, we consider that the payment information has been sent once the message is emitted through Kafka.

② As we mentioned, we are not going to get rid of the REST call SpendPal, we are just going to use it as a fallback mechanism.

After you have refactored the code, it is time to test it. Make sure you have configured a user correctly and send a request to charge subscriber.

```
curl --location --request POST 'http://localhost:8080/subscription' \
--header 'Authorization: Bearer bearing'
```

What you are expected to see as a result is log messages stating the successfulness of the message transmission in both services.

```
In Magman
2024-02-07 13:55:02,285 INFO  [com.vid.mag.mes.KafkaMessageService] (executor-thread-
1) Successfully emitted message to payments topic:
{"creditCardDTO":{"number":"123456778893233242","type":"VISA"},"username":"cave123"}

And in SpendPal
2024-02-07 13:55:02,285 INFO  [com.vid.mag.mes.KafkaMessageService] (executor-thread-
1) Successfully emitted message to payments topic:
{"creditCardDTO":{"number":"123456778893233242","type":"VISA"},"username":"cave123"}
```

Now that we know it works, we can do the opposite thing for the `post-payments` topic.

In SpendPal

```
@Incoming("payments")
public void consumePostPaymentMessage(String message) {
    PaymentPayload payload = JsonbBuilder.create().fromJson(message, PaymentPayload
.class);
    LOGGER.info("Received message with payload: %s".formatted(message));

    PaymentConfirmation paymentConfirmation = new PaymentConfirmation(payload.
username(), new ConfirmationDTO(true, LocalDateTime.now()));
    sendPaymentsMessage(paymentConfirmation);
}

public void sendPaymentsMessage(PaymentConfirmation confirmation) {
    String payload = JsonbBuilder.create().toJson(confirmation);
    postPaymentsEmitter.send(payload);
    LOGGER.info("Successfully sent payment confirmations with payload: %s".formatted
```

```
(payload));
}
```

And in MagMan

```
@Incoming("post-payments")
public void consumePostPaymentMessage(String message) {
    PaymentConfirmation paymentConfirmation = JsonbBuilder.create().fromJson(message,
PaymentConfirmation.class);
    LOGGER.info("Received payment confirmation for username %s and status %s"
.formatted(paymentConfirmation.username(), paymentConfirmation.confirmationDTO()
.getSuccess()));
}
```

Now every time you send a new request to charge customer, you will see two additional logs in MagMan.

```
2024-02-07 14:38:45,689 INFO  [com.vid.mag.mes.KafkaMessageService] (executor-thread-
1) Successfully emitted message to payments topic:
{"creditCardDTO":{"number":"123456778893233242","type":"VISA"},"username":"cave123"}
2024-02-07 14:38:47,887 INFO  [com.vid.mag.mes.KafkaMessageService] (vert.x-eventloop-
thread-3) Received payment confirmation for username cave123 and status true
```

This signifies that the communication between the two services is working properly.

13.6. Let's put our subscriptions into use shall we?

Since our Kafka communication is working it is time to do something with those subscriptions, not just pass messages. Let's make it so that our system can keep track of the Subscriber's subscription status.

1. Create a new entity, called Subscription

```
@Entity
public class Subscription extends AbstractEntity {

    @ManyToOne
    public Subscriber subscriber;

    //We need to support three types of statuses here: PENDING, VALID and FAILED
    @Enumerated(EnumType.STRING)
    public SubscriptionStatus status = SubscriptionStatus.PENDING;

    public LocalDateTime initiated = LocalDateTime.now();

    public LocalDateTime completed;
```

```
    public Subscription() {

    }

    public Subscription(Subscriber subscriber) {
        this.subscriber = subscriber;
    }

    public static Optional<Subscription> findLastPendingSubscription(Subscriber
subscriber) {
        return find("subscriber=?1 and status='PENDING'", Sort.descending(
"initiated"), subscriber)
                .firstResultOptional();
    }
}
```

2. Optimise the event handling upon new subscription

```
public class SubscriptionExtensionHandler {
    ...
    @Transactional
    @ActivateRequestContext
    public void observeSubscriptionExtension(@Priority(Priorities.APPLICATION +
2000) @Observes @ChargedSubscriber SubscriberChargedPayload payload) { ①
        Subscriber subscriber = Subscriber.getEntityManager().merge(payload
.subscriber()); //making sure that the subscriber entity is attached

        Subscription subscription = Subscription.findLastPendingSubscription
(payload.subscriber())
                .orElse(new Subscription(payload.subscriber()));
        if (payload.confirmation().getSuccess()) {
            subscriber.subscribedUntil = subscriber.subscribedUntil.plusYears(1);
            subscription.status = SubscriptionStatus.VALID;
            LOGGER.log(Level.INFO, "Extended subscription for user {0}, till {1}",
                    List.of(subscriber.id, subscriber.subscribedUntil.toString())
.toArray());
        } else {
            subscription.status = SubscriptionStatus.FAILED;
        }

        subscription.completed = payload.confirmation().getTimestamp();
    }

    public void sendEmail(@Priority(Priorities.APPLICATION + 1000) @Observes
@ChargedSubscriber SubscriberChargedPayload payload) {
        LOGGER.log(Level.INFO, "Sent email to subscriber {0}, about their
subscription renewal.", payload.subscriber().id);
    }
}
```

① Here we used to pass just the `Subscriber` as a payload, but now as we are processing more information, regarding the subscription, we will need a more detailed payload, requiring us to change the event payload itself. Here is an example of how that payload should look:

```
public record SubscriberChargedPayload(Subscriber subscriber, ConfirmationDTO
confirmation) { }
```

3. Now let's go back to the Payment service and refactor some logic there

```
@Transactional
public boolean chargeSubscriber(Subscriber subscriber) throws SpendPalException {
        subscriber = Subscriber.getEntityManager().merge(subscriber); //We make
sure that the subscriber instance is attached to the entity manager.
    failPreviousSubscriptionAttempt(subscriber); ①

    Subscription subscription = new Subscription(subscriber);
    subscription.persist(); ②

    if (subscriber.creditCard != null) {
        CreditCardDTO creditCardDTO = new CreditCardDTO(subscriber.creditCard);
        try {
            kafkaMessageService.sendPaymentsMessage(new PaymentPayload(subscriber
.userName, creditCardDTO));
            return true;
        } catch (Exception e) {
            LOGGER.severe(e.getMessage());
            return chargeSubscriberThroughRest(subscriber); ③
        }
    } else {
        subscription.status = SubscriptionStatus.FAILED; ④
        subscription.completed = LocalDateTime.now();
        return false;
    }
}

private void failPreviousSubscriptionAttempt(Subscriber subscriber) {
    Subscription.findLastPendingSubscription(subscriber)
            .ifPresent(s -> {
                s.status = SubscriptionStatus.FAILED;
                s.completed = LocalDateTime.now();
            });
}
```

① This operation is performed just to make sure there are no subscriptions left in `PENDING`, because we are creating a new `PENDING` one.

② As defined in the `Subscription` class, by default every new subscription gets the status `PENDING` so we do not need to set it explicitly here.

③ To make this method short and more readable, the logic behind our fallback mechanism has been moved to a dedicated method.

④ Since we have persisted the subscription, it is managed by the Entity Manager and every other change within the `@Transactional` scope will commit to the transaction, without needing to call `persist()` on the method again.

4. If you have not done this yet, feel free to extend the `UserDTO` class to see more information upon user login. For example you can add `subscribedUntil` date to it, so wen the client gets a login response, they can immediately check if the user is subscribed or not.
5. The final step is to handle the `KafkaMessageService` class. Here we will need to inject the event for `@ChargedSubscriber` and invoke it, once we receive a message from Kafka.

```
    @Inject
    @ChargedSubscriber
    Event<SubscriberChargedPayload> subscriberChargedEvent;

    @Transactional
    @Incoming("post-payments")
    public void consumePostPaymentMessage(String message) {
        PaymentConfirmation paymentConfirmation = JsonbBuilder.create().fromJson
(message, PaymentConfirmation.class);
        Subscriber subscriber = Subscriber.find("userName", paymentConfirmation
.username()).firstResult();
        LOGGER.info("Received payment confirmation for username %s and status %s"
.formatted(paymentConfirmation.username(), paymentConfirmation.confirmationDTO()
.getSuccess()));

        if (subscriber == null) {
            LOGGER.warning("No subscriber with the user name of '%s' was found."
.formatted(paymentConfirmation.username()));
        }

        SubscriberChargedPayload eventPayload = new SubscriberChargedPayload
(subscriber, paymentConfirmation.confirmationDTO());
        subscriberChargedEvent.fire(eventPayload);
    }

    public void sendPaymentsMessage(PaymentPayload payload) {...}
}
```

These steps now should be sufficient to demonstrate how our application can act upon sending and receiving Kafka messages.

13.7. What next?

Now that you know how to configure messaging services for your application, you can try and move further. Here are some things you might want to try:

1. Check the full documentation of the SmallRye Kafka extension in Quarkus.
2. Try to set up some unit tests using the in-memory reactive messaging Quarkus plugin.
3. Try to experiment with different types of scenarios where the communication with Kafka might fail and think of ways those issues could be resolved.

Chapter 14. MicroProfile Reactive Messaging and Server-Sent Events (SSE)

In the previous chapter we learned what messaging between applications is and how it can be used to allow applications to communicate between each-other asynchronously, on order to achieve seamless user experience, where the actions of the user will be handled by our services without having the user wait for a response. To do so, we provided the `quarkus-smallrye-reactive-messaging-kafka` extension to our applications, which, with a couple of configurations, allowed them to pass messages to each-other asynchronously and act upon the events happening within.

In this chapter we are going to learn what the reactive part of our extension means and how we can use it for a real-life scenario. But first…

What is Reactive Programming?

Reactive programming is a programming paradigm that focuses on reacting to changes in data or events rather than explicitly defining step-by-step instructions. In simple terms, it's about setting up reactions to events, like data changes or user interactions, and letting the program automatically respond to those events.

In other the reactive programming is looking from a different aspect of treating data. It provides functional APIs that allow to directly act upon the events, regarding the stream of data, instead of having to pre-define those instructions in an imperative manner.

Real-life scenarios where reactive programming is commonly used include:

- **User interfaces:** Reacting to user inputs such as clicks, typing, or mouse movements.
- **Web applications:** Managing asynchronous data streams like HTTP requests, WebSocket connections, or data from databases.
- **IoT (Internet of Things):** Handling real-time data from sensors or devices.
- **Gaming:** Reacting to user actions and events in real-time.

All of these applications of reactive programming can be performed using imperative approaches, but in some user scenarios the implementation would be way more simple and minimal if we are using reactive APIs.

So what are the differences between imperative and reactive programming?

In **imperative programming**, you explicitly define the steps and order of execution to achieve a desired outcome. It's like following a recipe, where you tell the computer exactly what to do and when to do it. In **reactive programming**, however, you define how your program should react to changes or events. It's more like setting up triggers and letting the program handle the responses automatically.

In conclusion, we are better off using reactive programing over imperative in the following scenarios:

- **Asynchronous Operations:** Reactive programming shines when dealing with asynchronous operations and event-driven architectures.
- **Real-time Applications:** For applications requiring real-time responsiveness, such as games or interactive dashboards.
- **Complex Event Flows:** When dealing with complex event flows or data streams, reactive programming can provide better organization and handling.

Ultimately, the choice between reactive and imperative programming depends on factors like the nature of the application, team expertise, and performance considerations. It's essential to weigh the pros and cons of each paradigm based on your specific use case.

Reactive programming, as a concept is vast and has a lot of concepts that you might need to understand, before proceeding with this chapter. The chapter intents to show you how you can use this paradigm within Quarkus. If you need to get better understanding of how and why things work, please refer to the resources at the end of the chapter or look for more information in the documentations of the APIs we are going to use.

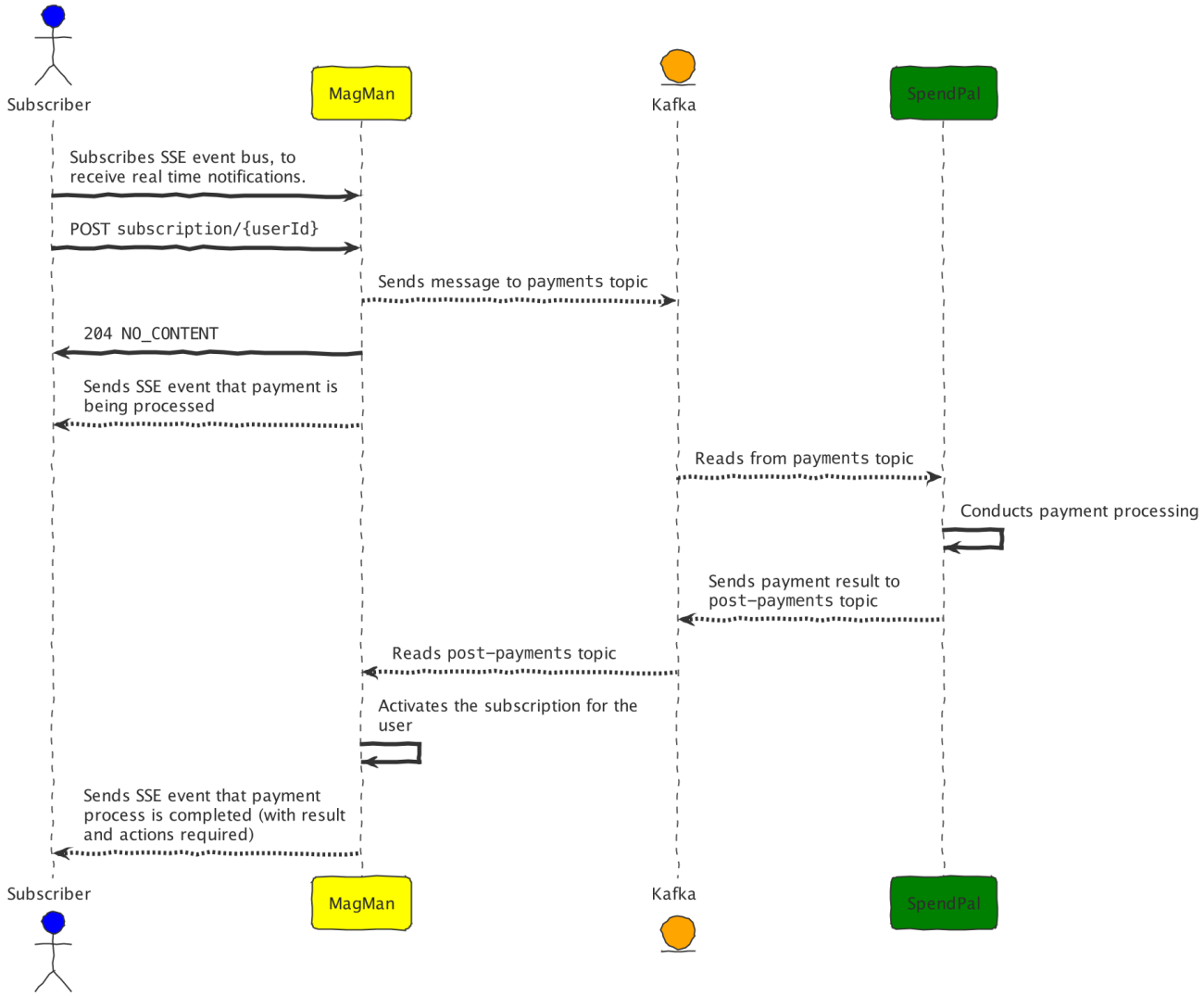
As you may have noticed our application does not immediately require the use of reactive programming, as it is small in scale, doesn't have a lot of asynchronous operations, does not need to react to a constant real-time event flows. In spite of that, this does not stop us to use whatever is available by the APIs provided or whatever suits the project best.

For example we're kind of forced to implement some reactive programming with the SmallRye Kafka plugin, as it uses reactive APIs. Knowing this, we may be able to use the event stream and send those events to the client, to create a front-end notification system for example.

Imagine a scenario where the user purchases a subscription, and a couple of minutes later gets a notification on their profile, that the purchase was completed successfully, or has failed, and they need to take action. This is what we are going to aim at in this chapter.

14.1. Server-Sent Events

Server-Sent Events or SSE for short are streams of events that are constantly sent from the server to the client. They use to replace the former WebSocket technology with a simpler and more flexible solution to notifying a web/application based interface of occurring events regarding a process, engaged with the user's session. As we mentioned earlier we want the customer to receive events upon changes happening regarding the status of the payment. Let's see how this would look in our updated sequence diagram:



Once the customer is logged in, they will subscribe to an SSE channel, that opens an endless stream to provide the client application with data on demand. Once the communication is established, the web application will start publishing events to the stream as they occur.

14.2. Adopting reactive

The `quarkus-smallrye-reactive-messaging-kafka` extension on its own does not allow us to invoke the reactive APIs for other purposes outside the boundaries of the Kafka implementation, although we can directly stream the received messages through a Server-Sent Event, we want to be smart about it, and maybe have a bit more control.

But first we need to import and alter some maven dependencies to allow our application to use reactive code.

1. Replace `quarkus-resteasy` and quarkus-resteasy-jsonb with the following dependencies:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-resteasy-reactive-jsonb</artifactId>
</dependency>
<dependency>
```

```
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-resteasy-reactive</artifactId>
</dependency>
```

Depending on your needs, you might need to use even more reactive APIs requiring you to call reactive Panache or reactive Hibernate for example. This chapter only affects the RestEasy APIs.

Failing to replace those dependencies or leaving both reactive and non-reactive maven dependencies inside the project will cause a run-time conflict and the project won't start. Make sure that you have only the reactive libraries imported to your project.

2. Add the following dependencies to your project:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-vertx</artifactId> ①
</dependency>
<dependency>
  <groupId>io.smallrye.reactive</groupId>
  <artifactId>smallrye-mutiny-vertx-web-client</artifactId> ②
</dependency>
```

① Eclipse Vert.x is a reactive library that will allow you to pass through system-wide events and act upon their arrival to the subscribed listeners. Unlike the Jakarta CDI events, Vert.x events function differently, as they can be handled from anywhere inside the code and work asynchronously.

② Mutiny is yet another reactive API that serves as the back-bone of RestEasy reactive implementations. We want to use Vert.x's EventBus capabilities alongside the APIs of Mutiny, so we need this library that is going to merge both APIs together.

Adding and replacing just those dependencies will not require you to change anything inside the code. Everything should work as expected.

Once we have this configured it is time to work with the code.

14.3. Implementing Server-Sent event logic using Mutiny and Vert.x

Implementing Server-Sent Events logic is quite straight-forward. The most important thing here is to be able to consume those type of events with your web client, if you want to be sure that what you're doing is working. The easiest way to consume server-sent events is by using a software that supports this type of REST response. A good recommendation for that will be Postman.

If you are going to listen for SSEs using postman, please make sure that you are using the latest version from the official website. Some app stores tend to publish older versions, which might not support SSE and you won't be able to receive those events.

To start off, we are going to need an endpoint that our customers will call every time they log into their account to start listening for events. In MagMan, create a new package and a class, called `sse.SseResouce`:

```
@RequestScoped
@Authenticated
@Path("sse")
public class SseResource {

    @POST
    @Produces(MediaType.SERVER_SENT_EVENTS)
    @SseElementType(MediaType.TEXT_PLAIN)
    public Multi<String> stream() {
        return Multi.createFrom().item(() -> "Hello Mutiny!");
    }
}
```

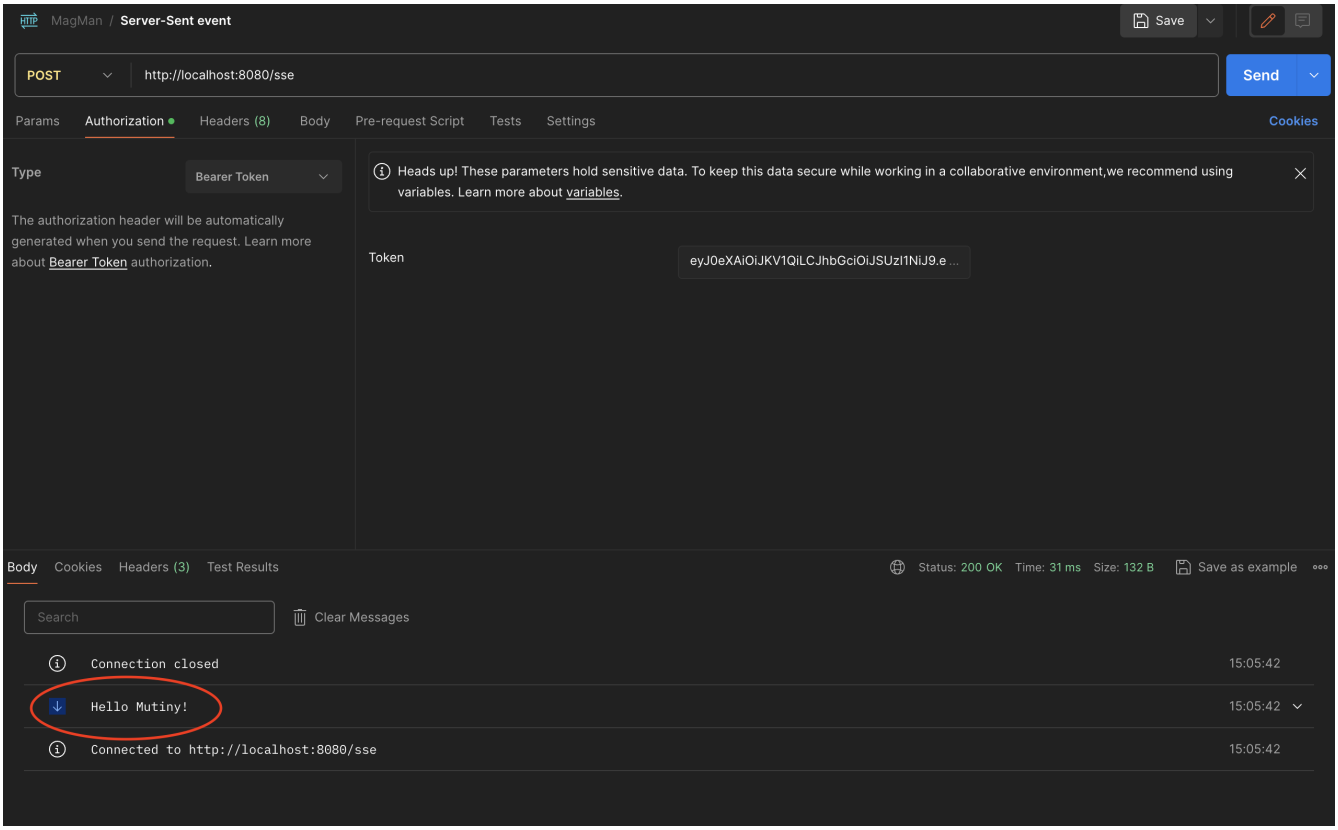
Now let's make sure that our imports and implementation is working properly. To do so, we are going to first implement a `"Hello Mutiny"` response here, that we are to expect when we test the endpoint.

```
public Multi<String> stream() {
    return Multi.createFrom().item(() -> "Hello Mutiny!");
}
```

Time to test our endpoint. Use your SSE supported client and call the endpoint.

Don't forget to provide Authorization header.

If you are using Postman, you should be able to see a screen like this:



If your result is the same, we can proceed with integrating Vert.x's EventBus:

```
@Inject
EventBus bus; ①

String userId; ②

public SseResource(JsonWebToken jwt) {
    userId = jwt.getClaim("upn");
}

@POST
@Produces(MediaType.SERVER_SENT_EVENTS)
@SseElementType(MediaType.TEXT_PLAIN)
public Multi<String> stream() {
    return bus.<String>consumer(userId) ③
            .bodyStream().toMulti();
}
```

① Make sure you are importing `io.vertx.mutiny.core.eventbus.EventBus` or the methods `bodyStream().toMulti()` will not be available.

② With the current state of the dependencies we have, we may not be able to use the non-reactive version of EntityManager within the scope of a reactive resource, so to make things easy we are going to use the user's id as the event identifier.

If you still want to inject the user here, you'll have to explicitly tell the resource to write in the working thread. This can be done by adding the `@Blocking` annotation on class level. If you want to work with the database on a non-

> blocking thread however, you will have to go through migrating hibernate and panache to their reactive counterparts, which will require to restructure the whole project.

③ We pass the userId to the `consummer()` method to denote what the name/address of the event is going to be.

Now every time somebody uses the event bus to send event to that address (the user's id), this consumer will pass the message down to the SSE stream.

The next step is to implement triggers, where the event will occur. As stated in our diagram, we want to get notification whenever something has happened with our transaction. The most convenient place to snag in such type of events would be inside the `KafkaMessagingService`. So let's go and make some changes to our produces and consumers.

1. Let's define a format in which notifications will be received by the customer. Inside the `sse` package, create a record for SSE payload.

```
public record SsePayload(Type type, String message) {

    @Override
    public String toString() {
        return JsonbBuilder.create().toJson(this);
    }

    enum Type {
        PAYMENTS, POST_PAYMENTS
    }
}
```

2. Go to KafkaMessagingService and alter the consumer and producer to uae the EventBus:

```
@ApplicationScoped
public class KafkaMessageService {

    //Loggers, Emitters, CDI events

    @Inject
    EventBus eventBus;

    @Transactional
    @Incoming("post-payments")
    public void consumePostPaymentMessage(String message) {
        PaymentConfirmation paymentConfirmation = JsonbBuilder.create().fromJson
(message, PaymentConfirmation.class);
        Subscriber subscriber = Subscriber.find("userName", paymentConfirmation
.username()).firstResult();
        LOGGER.info("Received payment confirmation for username %s and status %s"
.formatted(paymentConfirmation.username(), paymentConfirmation.confirmationDTO()
```

```
.getSuccess()));

        if (subscriber == null) {
            LOGGER.warning("No subscriber with the user name of '%s' was found."
.formatted(paymentConfirmation.username()));
        }

        SubscriberChargedPayload eventPayload = new SubscriberChargedPayload
(subscriber, paymentConfirmation.confirmationDTO());
        eventBus.send(subscriber.id + "", new SsePayload(SsePayload.Type
.POST_PAYMENTS, message).toString()); ①
        subscriberChargedEvent.fire(eventPayload);
    }

 public void sendPaymentsMessage(Long userId, PaymentPayload payload) {
        String payloadString = JsonbBuilder.create().toJson(payload);
        paymentsEmitter.send(payloadString)
                .thenRun(() ->
                        eventBus.send(userId.toString(), new SsePayload(SsePayload
.Type.PAYMENTS, "Payment information sent!").toString())) ②
                .exceptionally(throwable -> {
                    LOGGER.severe("Unable to send message through Kafka: %s"
.formatted(throwable.getMessage()));
                    eventBus.send(userId.toString(), new SsePayload(SsePayload.
Type.PAYMENTS, "Error sending Payment request").toString()); ③
                    return null;
                });
        LOGGER.info("Successfully emitted message to payments topic: %s".formatted
(payloadString));
    }
}
```

① Here we can simply pass the result of the event to the event bus, and reflect it down to the customer

② Here we wait asynchronously for the message to finish being processed and send an event through the EventBus.

③ In case of an error in the asynchronous process, we added a reaction how to handle the error.

If you've followed through, everything should be ready for testing. Fire up Docker, Kafka, SpendPal, and MagMan, to start testing the implementations in the following order.

1. Create your user or use an exiting one
2. Log in to the account
3. Subscribe to the SSE event from the endpoint we created in this chapter.
4. Register a credit card for the user.
5. Invoke the charge customer operation a couple of times.

If everything works properly, you should expect a couple of messages in the SSE event stream.

The stream will remain open wile the server is running and the client is connected. Until then every event that is passed through the event bus for that user id, will be sent to their respective SSE instance.

14.4. Some useful links

This chapter does not show it all, but if you find the topic interesting and want to learn more, then, here are some useful links to get you started.

- The guide to using Apache Kafka with Quarkus - https://quarkus.io/guides/kafka
- Getting started with Quarkus and Reactive - https://quarkus.io/guides/getting-started-reactive
- Understanding Mutiny, asynchronous code and its relation to Vert.x
- Using Vert.x with Quarkus
  - https://quarkus.io/guides/vertx-reference
  - https://quarkus.pro/guides/vertx.html
- Using Vert.x event bus with Mutiny https://quarkus.io/guides/reactive-event-bus
- Smallrye Mutiny documentation - https://smallrye.io/smallrye-mutiny/latest/
- Eclipse Vert.x official documentation - https://vertx.io/docs/vertx-core/java/

Chapter 15. Fault tolerance with MicroProfile

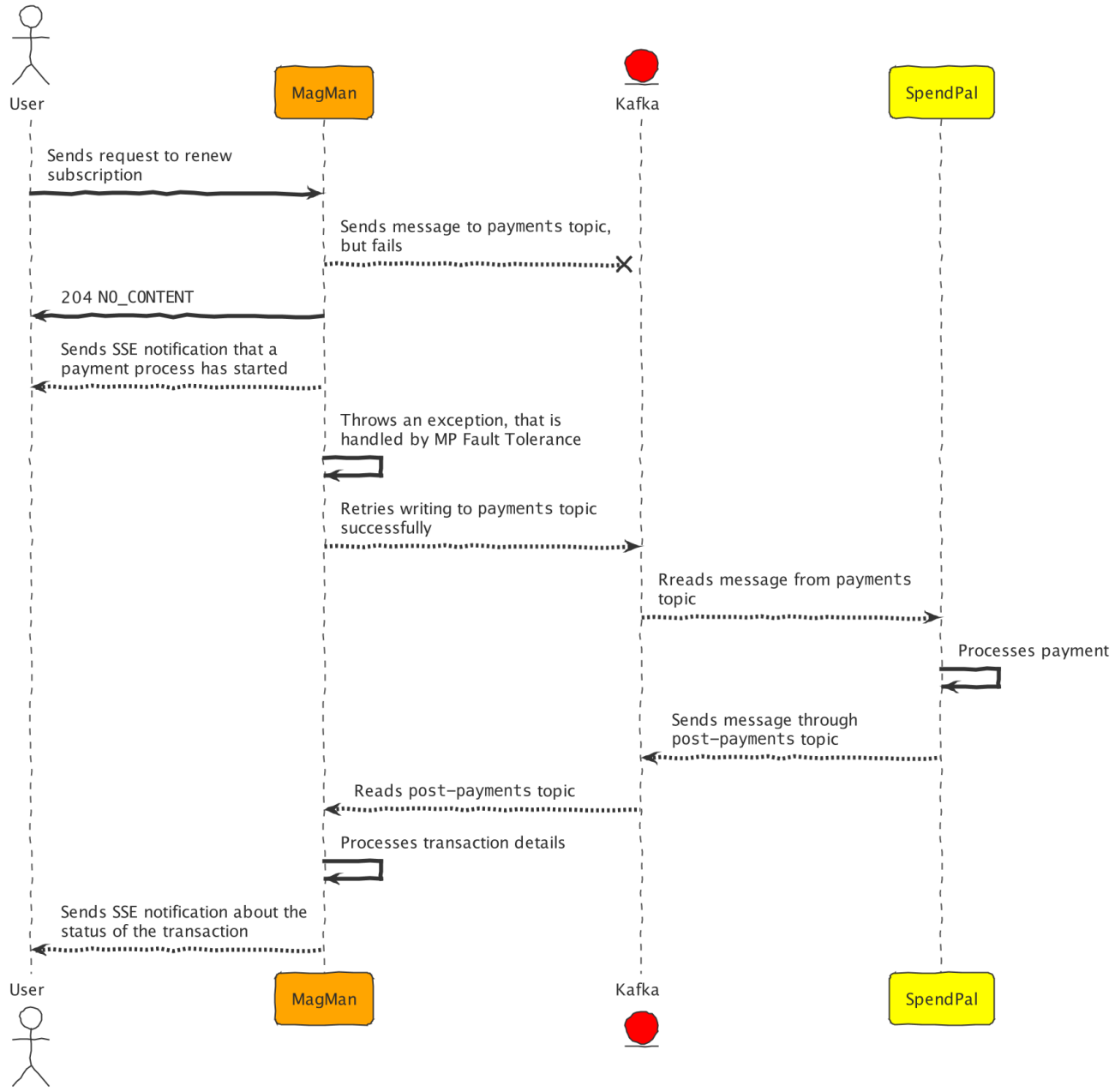
So far we have learned how to use MicroProfile specifications to add configurations, communicate through rest client and kafka and secure the access to our application. Our next endeavour will be the fault tolerance specifications of MicroProfile.

What is MP Fault tolerance?

MP Fault Tolerance provides a simple and flexible solution to build fault-tolerant microservices. Fault tolerance leverages different strategies to guide the execution and result of logic. As stated in the MicroProfile website, retry policies, bulkheads, and circuit breakers are popular concepts in this area. They dictate whether and when executions take place, and fallbacks offer an alternative result when an execution does not complete successfully.

Imagine that your application has to communicate to various other services internal or external and due to high traffic load or delay in communication the request fails to arrive, returns bad responses or something else fails. In this type of scenario, you would likely avoid telling the user that something went wrong until there is nothing you can do. Instead, what is a good practice to do is **retry** the failed call a couple of times, or maybe go through an alternative routine.
This is the type of problem that MP Fault tolerance is here to help you with. Instead of reinventing the wheel, you can use libraries specialized for this types of scenarios. Instead reinventing the wheel for every application you have, by implementing loops in a try-catch blocks or interceptors to handle exceptions and provide alternate flows, MP Fault Tolerance has you covered with very convenient APIs to handle such type of events.

Let’s hop into a scenario where fault tolerance would be useful:



This is a quite simple scenario, that clearly shows how we can use MP Fault Tolerance to tolerate the faults in the communications between our systems. Since Kafka is an external source that we pass data through, we can always expect hiccups to happen:

- The network connection might stop
- The zookeeper might be down
- Kafka may be too busy to acknowledge the message on time and the server times out
- The request might be malformed (although we are yet not sure about that)

Juts like quantum physics, the results and the underlying reasons behind them are unpredictable. Anything could happen, and we want to make sure that we have maximised our probability to process the customer's inquiry and minimise responses such as *"Something went wrong"*, or... *"Please try again later"*. To demonstrate how we can better tackle such scenarios, we are going to simulate the scenario in the diagram and see how MP Fault Tolerance helps us resolve the issues.

15.1. The `@Retry` annotation

`org.eclipse.microprofile.faulttolerance.Retry` is our first and most basic mechanism from the MP Fault Tolerance toolset. It simply does what it says. Once you place this annotation on a method it will invoke it in an interceptor, that is going to listen for exceptions. If an exception occurs, the interceptor will try to invoke the method again, as many times as the developer has defined. If the exception keeps being thrown, once the threshold has been surpassed, the interceptor will proceed by throwing the exception to the invocation class.

We can find usage of the `@Retry` annotation within the communication of MagMan and Kafka. This will allow us to tackle any unexpected problems that might occur and are not handled by our application, such as network failures or any other unexpected interruptions.

To start off, let's first introduce the latest plugin to our project - `smallrye-fault-tolerance`.

```
<dependency>
   <groupId>io.quarkus</groupId>
   <artifactId>quarkus-smallrye-fault-tolerance</artifactId>
</dependency>
```

Make sure to refresh your Maven dependencies, once you add the new plugin to be able to see the new APIs and compile the code from your IDE.

Our next step is to make the communication with Kafka more brittle. Currently, the smallrye kafka plugin is configured to infinitely retry to publish a message to a topic, meaning that if we break the connection with the publisher and the broker, the server will endlessly try to reconnect and halt any messages until the connection is reestablished. As good as it might be, we may want to try different ways to charge the customer if Kafka is not working. Although the process is asynchronous, the user will have to wait too much time to get their subscription renewed, which might be a problem on its own.

To make Kafka throw an exception when it's unable to publish a message to the topic, we are going to add a new property to our `application.properties` file:

```
kafka.retries=0
```

This will tell the Kafka publisher to not reattempt sending the message when it fails. Instead, it will throw an exception that will be handled by the code we already implemented for the purpose.

Although reattempts of publishing a message can be stopped, this does not stop Kafka from trying and waiting for a connection to happen. Keep in mind that this will add additional delay to the result we are going to expect.

Once we have that added, we will proceed with adding the `@Retry` annotation to `com.vidasoft.magman.messaging.KafkaMessageService#sendPaymentsMessage`

```
@Retry // org.eclipse.microprofile.faulttolerance.Retry ①
public void sendPaymentsMessage(Long userId, PaymentPayload payload) {
    String payloadString = JsonbBuilder.create().toJson(payload);
    paymentsEmitter.send(payloadString)
            .thenRun(() ->
                    eventBus.send(userId.toString(), new SsePayload(SsePayload.Type
.PAYMENTS, "Payment information sent!").toString()))
            .exceptionally(throwable -> {
                LOGGER.severe("Unable to send message through Kafka: %s".formatted
(throwable.getMessage()));
                eventBus.send(userId.toString(), new SsePayload(SsePayload.Type
.PAYMENTS, "Error sending Payment request").toString());
                return null;
            });
    LOGGER.info("Successfully emitted message to payments topic: %s".formatted
(payloadString));
}
```

① By default, the threshold of the `Retry` interceptor is 3. Meaning that this method will be called 3 times if an exception is thrown. If you want to increase/decrease the threshold, you can use one of `@Retry` 's attributes `maxRetries`.

With the addition of this annotation if `sendPaymentsMessage` throws an exception, we expect it to be handled by MP Fault Tolerance and the method to be called again. In the current state of the code, though, we are not going to be able to catch that exception, because we handle it within our Vert.x functions, which work asynchronously. To expose the exceptions that are thrown in `exceptioinally` we will have to synchronize emitter thread with the request's thread. But don't worry this will be temporarily.

```
public void sendPaymentsMessage(Long userId, PaymentPayload payload) {
    LOGGER.info("Attempting to send payment message"); ①
    String payloadString = JsonbBuilder.create().toJson(payload);
    paymentsEmitter.send(payloadString)
            .thenRun(() ->
                    eventBus.send(userId.toString(), new SsePayload(SsePayload.Type
.PAYMENTS, "Payment information sent!").toString()))
            .toCompletableFuture().join(); ②
    LOGGER.info("Successfully emitted message to payments topic: %s".formatted
(payloadString));
}
```

① Let's add a log to be able to count how many retrials were attempted.

② For now, as mentioned we will remove the exception handling here

Now let's attempt to break our connection.

1. Start all applications: MagMan, SpendPal, docker-compoe
2. Do the normal flow to make sure payments are still processed through kafka, without problems

3. Stop the docker containers by running `docker-compose down` in the root directory of the MagMan project
4. Reattempt sending request to Kafka
5. In a couple of minutes, you should be expecting a failure, with Transaction timeout exception. You should also see 3 new logs with the message `INFO [com.vid.mag.mes.KafkaMessageService] (executor-thread-1) Attempting to send payment message`.

We'll make a slight digression to clean up some debt...

Sometimes when dealing with asynchronous code or delays, that involve transactions, we might need to tackle some issues due to how scopes are transitioned between threads. In the current situation, since this is not an asynchronous operation anymore, we are going to have to wait a bit, before seeing any exception thrown. But there is one thing that doesn't like to wait - the transaction. Our `@Transactional` scope has a default timeout, that is very likely to be reached when all the retrials have passed. If we go back to our `PaymentService` class we are going to find the following piece of code:

```
try {
    kafkaMessageService.sendPaymentsMessage(subscriber.id, new PaymentPayload
(subscriber.userName, creditCardDTO));
    return true;
} catch (Exception e) {
    LOGGER.severe(e.getMessage());
    return chargeSubscriberThroughRest(subscriber);
}
```

If `sendPaymentsMessage` becomes synchronous when an exception is thrown and the fallback mechanism of `chargeSubscriberThroughRest` gets triggered. At this point though so much time has passed that the transaction has timed out. When we enter `chargeSubscriberThroughRest`, the moment we try to do something with the subscription object, an exception will be thrown and the changes will not take an effect.

To resolve the issue, we are going to try and limit the scope of the transaction to only where it is needed:

1. Remove the `@Transactional` annotation from `SubscriptionResource#chargeSubscriber`.
2. Remove the `@Transactional` annotation from `PaymentService#chargeSubscriber`.
3. Extract the code that creates subscriptions into a separate method and make it public and `@Transactional`.

```
    public boolean chargeSubscriber(Subscriber subscriber) throws
SpendPalException {
        Subscription subscription = createSubscription(subscriber); ①

        if (subscriber.creditCard != null) {
            CreditCardDTO creditCardDTO = new CreditCardDTO(subscriber
```

```
.creditCard);
            try {
                kafkaMessageService.sendPaymentsMessage(subscriber.id, new
PaymentPayload(subscriber.userName, creditCardDTO));
                return true;
            } catch (Exception e) {
                LOGGER.severe(e.getMessage());
                return chargeSubscriberThroughRest(subscriber);
            }
        } else {
            subscription.status = SubscriptionStatus.FAILED;
            subscription.completed = LocalDateTime.now();
            return false;
        }

    }

    @Transactional
    public Subscription createSubscription(Subscriber subscriber) {
        failPreviousSubscriptionAttempt(subscriber); ①
        Subscription subscription = new Subscription(subscriber);
        subscription.persist();
        return subscription;
    }
```

① We have moved `failPreviousSubscriptionAttempt` to `createSubscription` in order to keep it in the transaction scope.

4. Make `chargeSubscriberThroughRest` public and transactional

Now if you try and execute the `chargeSubscriber` method again, you will get 204 as a response and if we check the expiration date of the subscription, we will see that it has increased with an year. Our fallback REST method is being called.

15.2. The `@Asynchronous` annotation

Previously we have shown that you can use the reactive functions from Mutiny and Vert.x to achieve asynchronous code execution. But how to you turn an imperative piece of code into a reactive one? By using the `@org.eclipse.microprofile.faulttolerance.Asynchronous` annotation.

We will start off, by refactoring `KafkaMessageService#sendPaymentsMessage`.

```
@Retry
@Asynchronous ①
public CompletionStage<Void> sendPaymentsMessage(Long userId, PaymentPayload payload)
{ ②
    LOGGER.info("Attempting to send payment message");
    //same implementation ...
```

```
    return CompletableFuture.completedFuture(null); ③
}
```

① First we place the `@Asynchronous` annotation to the desired method.

Make sure that the method you are making asynchronous is `public`, otherwise MP Fault Tolerance will not be able to recognise this method as asynchronous, and it will run synchronised.

② All asynchronous methods should return an implementation of `java.util.concurrent.Future` or `java.util.concurrent.CompletionStage`.

Since `java.util.concurrent.Future` is expecting a generic type to serve as a result, we are going to provide the object `Void` which is used just as a placeholder in order to be able to satisfy the syntax of generic types.

③ Finally, we are using the builder of `CompletableFuture` to pass the return type. Since we cannot instantiate `Void`, the most relevant thing to do here is put `null` inside the parameter of `completedFuture()`.

Next, we are going to modify the `PaymentService` again by altering our `chargeCustomer` code.

```
    public boolean chargeSubscriber(Subscriber subscriber) {
        Subscription subscription = createSubscription(subscriber);

        if (subscriber.creditCard != null) {
            CreditCardDTO creditCardDTO = new CreditCardDTO(subscriber.creditCard);
            kafkaMessageService.sendPaymentsMessage(new PaymentPayload(subscriber
.userName, creditCardDTO)) ①
                    .thenRun(() -> eventBus.send(subscriber.id + "",
                            new SsePayload(SsePayload.Type.PAYMENTS, "Payment
information sent!").toString())) ②
                    .exceptionally(throwable -> { ③
                        LOGGER.severe(throwable.getMessage());
                        eventBus.send(subscriber.id + "", new SsePayload(SsePayload
.Type.PAYMENTS,
                                "Error sending message with Kafka. Will try REST: %s"
.formatted(throwable.getMessage())).toString());
                        try {
                            chargeSubscriberThroughRest(subscriber);
                        } catch (SpendPalException spe) {
                            try (ByteArrayInputStream bais = (ByteArrayInputStream)
spe.getBody()) {
                                String responseMessage = new String(bais.readAllBytes
());
                                eventBus.send(subscriber.id + "", new SsePayload
(SsePayload.Type.PAYMENTS, "Error making subscription through REST. " +
                                        "Please retry making a subscription: %s"
.formatted(responseMessage)));
                            } catch (IOException ioe) {
```

```
                                eventBus.send(subscriber.id + "", new SsePayload
(SsePayload.Type.PAYMENTS, "Error making subscription through REST. " +
                                        "Please retry making a subscription: %s"
.formatted(ioe.getMessage())));
                            }
                        }
                        return null;
                    });

            return true;
        } else {
            subscription.status = SubscriptionStatus.FAILED;
            subscription.completed = LocalDateTime.now();
            return false;
        }
    }
```

① Since we are returning a `CompletionStage` here, we can handle the result of this state in the future, without waiting for the Kafka message to fail.

② If the method has returned with no error, we are going to fall into the `thenRun()` function.

③ If an exception is thrown, however, we are going to fall into the `exceptionally()` method, where we have to run our fallback logic.

Remember that all of this is happening on a separate thread. Now when you run the code, unless you have not provided credit card for the customer, you will always get 204 as a response. Notice that we have moved part of the eventbus logic here. This will help us to track what happens in the future, by reading the events from the SSE endpoint we previously made. And if you have configured everything correctly at the end you should see the following event messages:

Figure 4. The first two messages bottom-to-top are when the flow ran correctly. Then after stopping the Docker containers we see that we had an error, and we reached into the fallback mechanism.

Although this works, it doesn't look as neat. Let's try and change that.

15.3. The `@Timeout` annotation

Another annotation in the MP Fault Tolerance suite is the `@Timeout` annotation. It simply does what it says. Adding this annotation to a method will cause it to break the execution if it takes over a certain amount of time. And guess what, we need to wait over three minutes to get an exception from Kafka. This is too long of a wait for the customer to get their subscription extended. To speed up the process we are going to stop tolerating the wait time to reconnect with Kafka to emmit the message.

```
@Retry
@Timeout ①
@Asynchronous
public CompletionStage<Void> sendPaymentsMessage(PaymentPayload payload) {
 //implementation
}
```

① By default `@Timeout` will wait 1000 milliseconds before it interrupts the execution of the method. You can increase/decrease that timeout, and also set the preferable unit, withing the annotation's parameters. Those parameters are optional and may be omitted if you are happy with the default options.

Now if we try to call `chargeCustomer` again, we should wait around 3 seconds to execute the fallback logic. Talking about fallback logic, it is time to finally clear out our fallback mechanism and make it a bit more simple.

15.4. The `@Fallback` annotation

`@Fallback`, just as the other annotations from this library, maintains interception that will invoke an alternate method in the case where the main method has failed. Instead of us having to make a try-catch logic where we have to manually invoke the alternate execution. We can let our code do it on its own, when it decides it is time to call it.

In our scenario, when Kafka fails, we want to automatically trigger the fallback method and pay the subscription through REST. To make this happen there are a couple of steps we need to perform.

1. Annotate the method we expect to fail with `@Fallback`
2. Provide information in the `@Fallback` parameters what should and when should happen upon the event of failure.
3. Implement the fallback code, that is going to handle the alternate implementation.

There are two ways to implement the fallback code:

1. By keeping the signature of the failed method to the fallback method and setting it in the `@Fallback` annotation parameter.
2. By creating a class, that implements the FallbackHandler class.

The first option is more limited and requires the fallback method to be in the same class, while the other needs to be implemented on a separate class and is more abstract to work with. The second option works similar to conventional CDI interceptors.

For ease, we are just going to adapt the `chargeSubscriberThroughRest` method to be invoked from a fallback trigger.

1. Move `chargeSubscriberThroughRest` from `PaymentService` to `KafkaMessageService`. As mentioned we need to have both methods in one place in order for the fallback to work.
2. Add/adapt the code to work with the new class it is in.

3. Make the signature of `chargeSubscriberThroughRest` the same as `sendPaymentsMessage`.

```
    @Retry
    @Timeout
    @Fallback(fallbackMethod = "chargeSubscriberThroughRest") ①
    @Asynchronous
    public CompletionStage<Void> sendPaymentsMessage(PaymentPayload payload) {
        // implementation
    }

    @Transactional
    public CompletionStage<Void> chargeSubscriberThroughRest(PaymentPayload
payload) throws SpendPalException {
        Subscriber subscriber = Subscriber.find("userName", payload.username())
.firstResult();

        // same implementation as before

        return CompletableFuture.completedFuture(null);
    }
```

① Notice that we added the `@Fallback` annotation along with the name of the method we are going to fallback on.

4. Place the `@Fallback` annotation on `sendPaymentsMessage` along with the name of the method we are going to call.
5. Refactor the `exceptionally` logic inside `PaymentService#chargeCustomer`. We no longer need to call `chargeSubscriberThroughRest` manually.

At the end it should look something like this:

```
public boolean chargeSubscriber(Subscriber subscriber) {
    Subscription subscription = createSubscription(subscriber);

    if (subscriber.creditCard != null) {
        CreditCardDTO creditCardDTO = new CreditCardDTO(subscriber.creditCard);
        kafkaMessageService.sendPaymentsMessage(new PaymentPayload(subscriber.
userName, creditCardDTO))
                .thenRun(() -> eventBus.send(subscriber.id + "",
                        new SsePayload(SsePayload.Type.PAYMENTS, "Payment information
sent!").toString()))
                .exceptionally(throwable -> {
                    LOGGER.severe(throwable.getMessage());
                    eventBus.send(subscriber.id + "", new SsePayload(SsePayload.Type
.PAYMENTS, "Error making subscription. " +
                            "Please retry making a subscription: %s".formatted
(throwable.getMessage())));
                    return null;
                });
```

```
            return true;
        } else {
            subscription.status = SubscriptionStatus.FAILED;
            subscription.completed = LocalDateTime.now();
            return false;
        }
    }
```

15.5. The `@CircuitBreaker`

The last MP Fault Tolerance feature we are going to look at is the circuit breaker. So far when we can't reach out to Kafka, we have to wait until the `@Timeout` decided that enough time has passed to trigger the `@Fallback` logic. In our current configuration this takes about 3 seconds. But if we add more users asking for the same resource, the computational time will start adding up.

What if instead of trying to execute `sendPaymentsMessage` every time a new payment request is opened, we directly called our fallback `chargeSubscriberThroughRest`. This would drastically speed up the amount of processed payments. What `@CircuitBreaker` does is help you define a certain threshold where if the method fails, it will stop being called for a certain amount of time, to avoid further damage by not executing functionality that is doomed to fail.

We can for example define our server to try calling `sendPaymentsMessage` and if it fails over 5 times, for the next 5 seconds to directly call the fallback. This is how we do it:

```
@Retry
@Timeout(5000) ①
@Fallback(fallbackMethod = "chargeSubscriberThroughRest") ②
@CircuitBreaker(requestVolumeThreshold = 3)
@Asynchronous
public CompletionStage<Void> sendPaymentsMessage(PaymentPayload payload) {
   //implementation unchanged
}
```

① We increased the timeout here in order to see the change more obviously, based on the time we wait

② By default `CircuitBreaker` will get triggered if the method fails 20 times, so we need to change it to something lower, to see any results faster. Other attributes are left with their defaults, meaning that the circuit will be opened for 5, bypassing `sendPaymentsMessage`, without having to retry and wait for something to happen second, and directly call `chargeSubscriberThroughRest`. After the cool-down has passed, the circuit will close, allowing for additional calls to `sendPaymentsMessage`, until the failure rate is met again.

You can find out more about CircuitBreaker's functionality in this beautiful article by Open Liberty here.

Let's try our application. If everything is configured properly, you should see that when Kafka is

down, the first request is processed slower, but the rest are directly processed by the fallback method, until 5 seconds have passed, and the circuit is closed again. Fell free to play with the thresholds if you want to observe a stronger effect of this or decrease it.

Chapter 16. Implementing OpenAPI documentation using MicroProfile OpenAPI

16.1. What is OpenAPI?

Our web applications do a lot of things. So much so, that we need to document every endpoint we implement to remember it is there and how to use it. Web applications don't work on their own. They need developers to implement web clients, such as front-end applications or other web applications, will communicate with our web servers. If we want a member of our development team to implement web page interface for our application, or allow other developers, outside our company to use our services APIs, we need to document our endpoints somewhere and share them.

Apparently the developer will use the documentation to better understand our APIs, how to call them and what to expect as a response. In ancient times everybody used to write their own documentation in their own style and format. This approach has a few drawbacks, however. You have to spend the time to write and style your documentation, update it every time you change something, and it is not interoperable, meaning that your documentation format is unique to your application. Also, when the application grows bigger you'll start putting less effort into updating that documentation, making it obsolete, and one day you'll have to spend a lot of time, checking that it is up-to-date.

Nowadays, there is a more modern solution to this concern. OpenAPI is here to do the job, by creating a standardized way to write API documentations, export them in yaml, and share them to other developers, who are going to implement your webhooks or endpoints either by hand or even automatically. There are plugins and tools that can generate endpoints based on an OpenAPI file. And that's the magic of it. Now when you want to give out documentation of your REST endpoints, you can simply pass them a yaml file, which they can auto-generate into a working code in their programming language.

With the power of MP OpenAPI, you get the ability to generate your openapi yaml and instantly keep it up to date, by using, you guessed it - annotations. You can also use the integrated Swagger UI to visualise and interact with your web application, without having to implement any front-end components for that purpose.

16.2. Integrating OpenAPI plugin into our application

To get started with our openapi documentation, we simply need to add yet another quarkus plugin in our `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-openapi</artifactId>
</dependency>
```

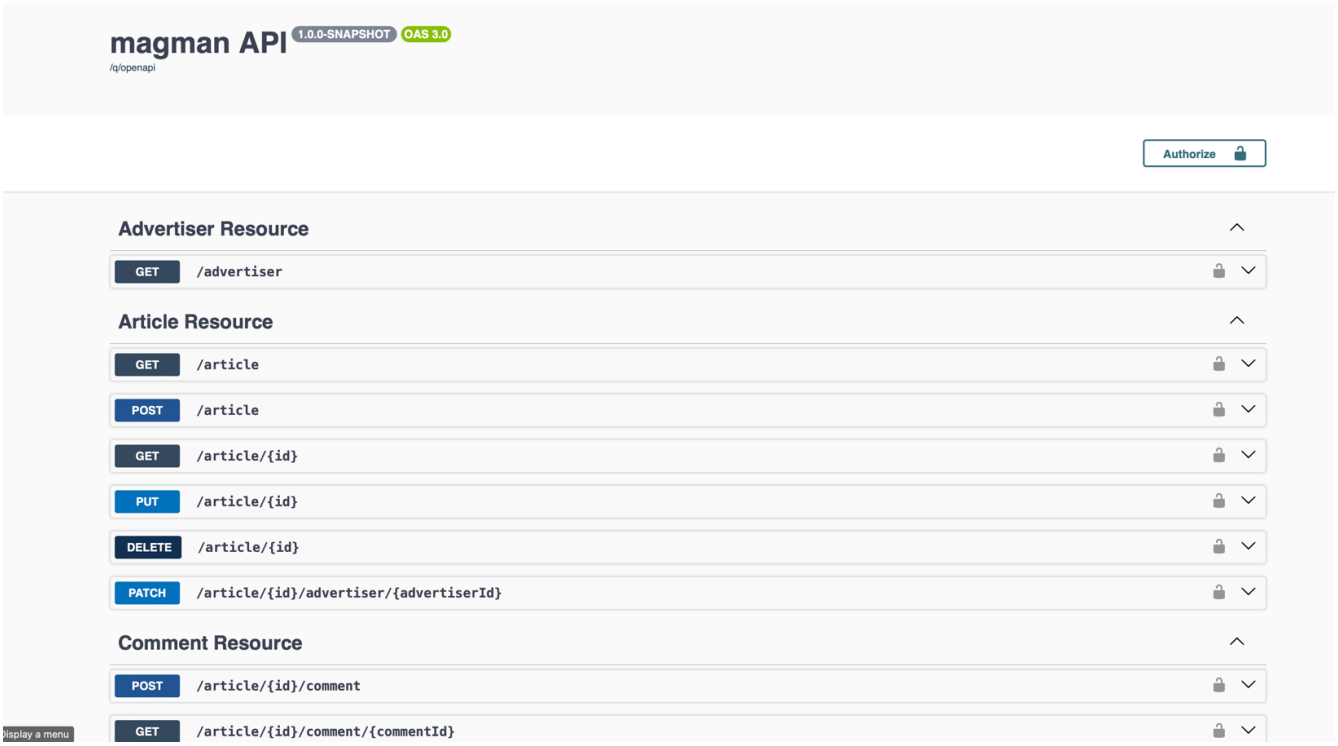
Don't be shy. You may add this plugin to all your Quarkus applications.

Now you can start your projects and go to `http://localhost:<prot>/q/openapi`. Your browser may ask you to download a yaml file, containing all the specified endpoints we created throughout this course. And the Swagger part - if you go to `/q/swagger-ui/`, you will see a handy user interface that you may start using directly to fiddle with your application. As previously mentioned this yaml file can be used by you and other developers to implement webhooks that can call your endpoints in their applications.

> But wait... we are not finished yet.

16.3. Using the MP OpenAPI annotations

Let's have a look at the Swagger UI page, that was generated by introducing our plugin:



It looks pretty hollow. We know there are different type of resources, that contain different type of endpoints, but we don't know what they are meant for and how to use them. Here comes the part with the annotations. We will use MP OpenAPI's annotations to better describe how our endpoints work. To start with the most obvious, let's first document our Article resource.

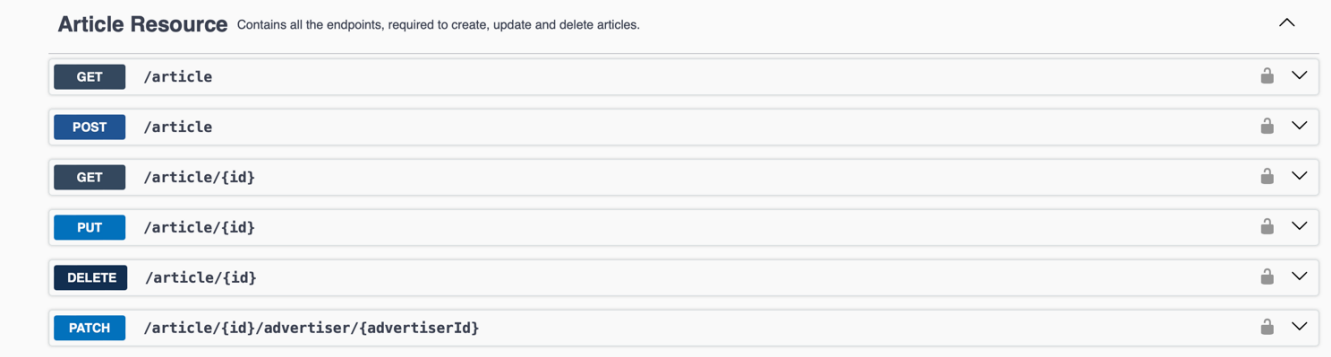
16.3.1. The `@Tag` annotation

We want to tell the readers of our documentation that Article resource is the section of endpoints responsible for maintaining articles on the platform. To do this, we need to annotate the whole class `ArticleResource` with `@Tag` and provide the needed attributes:

```
//imports
import org.eclipse.microprofile.openapi.annotations.tags.Tag;

@Authenticated
@RequestScoped
```

```
@Path("/article")
@Tag(name = "Article Resource", description = "Contains all the endpoints, required to
create, update and delete articles.")
public class ArticleResource {
    //implementation
}
```

Now if we reload the Swagger UI page, we are going to see that updated.



Great job! Now we have made the first step into making our documentation more understandable. Remember that this will also reflect into our openapi.yml file.

16.3.2. The `@Operation` annotation

Once we have the resource description out of the way, we might need to give a better description of our endpoints. This can simply be done by adding the `@Operation` annotation to each method, within the resource:

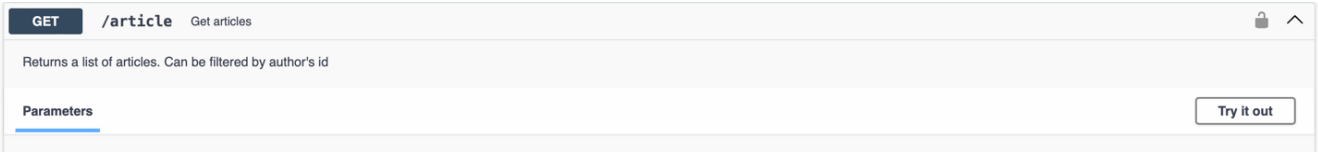
```
public class ArticleResource {

    @POST
    @Transactional
    @RolesAllowed({Author.ROLE_NAME})
    @Consumes(MediaType.APPLICATION_JSON)
    @Operation(
            operationId = "createArticle",
            summary = "Create article",
            description = "Creates an article"
    )
    public Response createArticle(@Valid @NotNull ArticleDTO articleDTO) {
        //code
    }

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(
            operationId = "getArticle",
            summary = "Get article",
            description = "Gets article by its id"
```

```
    )
    public Response getArticle(@Positive @PathParam("id") Long articleId) {
        //code
    }

    @PUT
    @Transactional
    @Path("/{id}")
    @RolesAllowed({Author.ROLE_NAME})
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(
            operationId = "editArticle",
            summary = "Edit article",
            description = "Updates an article by its id"
    )
    public Response editArticle(@Positive @PathParam("id") Long articleId, @Valid
@NotNull ArticleDTO articleDTO) {
        // code
    }

    @DELETE
    @Transactional
    @Path("/{id}")
    @RolesAllowed({Author.ROLE_NAME, Manager.ROLE_NAME})
    @Operation(
            operationId = "deleteArticle",
            summary = "Delete article",
            description = "Removes an article and its related comments by its id"
    )
    public void deleteArticle(@Positive @PathParam("id") Long articleId) {
        //code
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(
            operationId = "getArticles",
            summary = "Get articles",
            description = "Returns a list of articles. Can be filtered by author's id"
    )
    public Response getArticles(@QueryParam("page") @DefaultValue("1") @Positive int
page,
                                @QueryParam("size") @DefaultValue("10") @Positive int
size,
                                @QueryParam("author") @Positive Long authorId) {
        //code
    }

    @PATCH
    @Transactional
```

```
    @RolesAllowed({Manager.ROLE_NAME})
    @Path("{id}/advertiser/{advertiserId}")
    @Operation(
            operationId = "addAdvertiserToArticle",
            summary = "Promote article",
            description = "Adds advertiser to the article"
    )
    public Response addAdvertiserToArticle(@Positive @PathParam("id") Long id,
@Positive @PathParam("advertiserId") Long advertiserId) {
        //code
    }
}
```

Now if we refresh the page again, we should see our descriptions:

And if we open an entry, we will see our description:



16.3.3. The `@Parameter` annotation

So, you opened the "Get articles" tab, and you are met with a small form, allowing you to add in author, page and size.

Despite it being self-explanatory, one might want to read more information about these properties, get an example of what value they should contain or what are its limitations. From what our openapi plugin generated, we can see that that is not the case here. We will change that with the `@Parameter` annotation. You can place that either next to the parameter in the method's signature or on the level of the method.

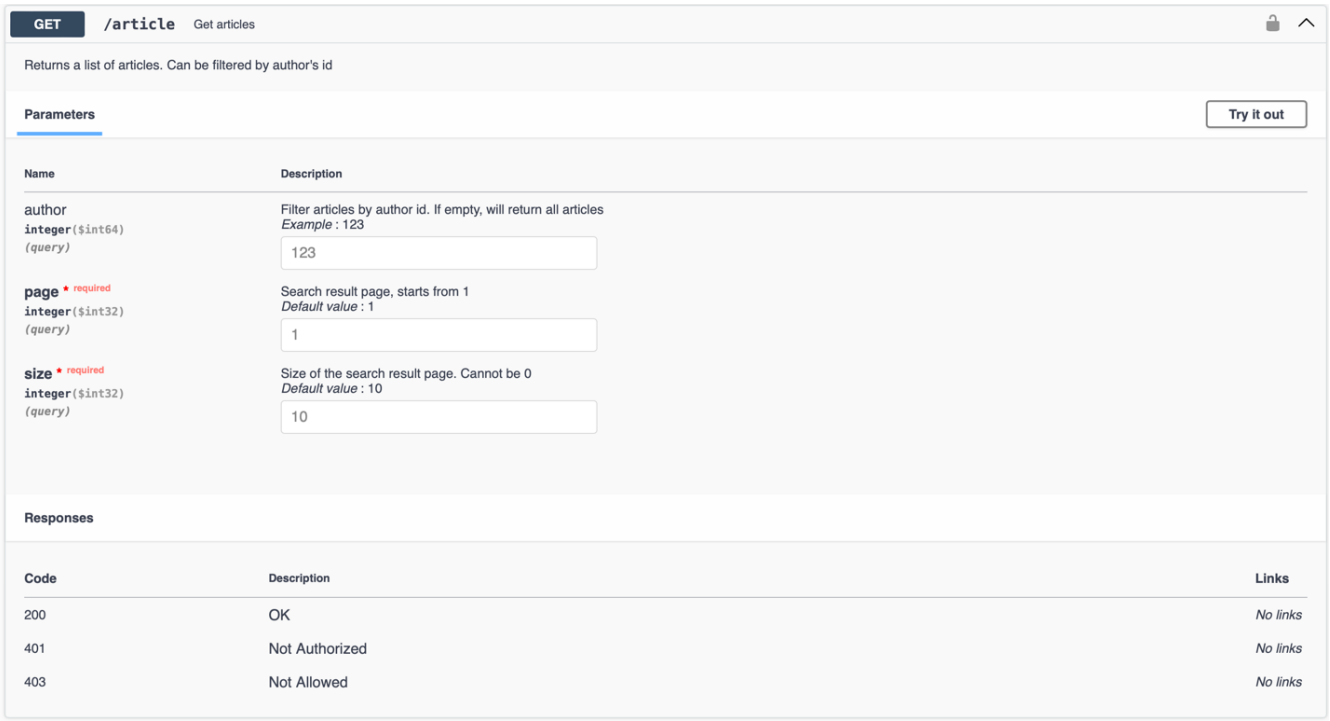
```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(
            operationId = "getArticles",
            summary = "Get articles",
            description = "Returns a list of articles. Can be filtered by author's id"
    )
    @Parameter(
            name = "author", ①
            in = ParameterIn.QUERY, ②
            description = "Filter articles by author id. If empty, will return all
articles",
            example = "123"
    )
    public Response getArticles(
            @Parameter(required = true, description = "Search result page, starts from
1") ③
@QueryParam("page") @DefaultValue("1") @Positive int page,
                                @Parameter(required = true, description = "Size of the
search result page. Cannot be 0") ③
@QueryParam("size") @DefaultValue("10") @Positive int size,
                                @QueryParam("author") @Positive Long authorId) {
    //code
                                }
```

① When using parameter annotation on method level, you must refer to the name matching that parameter, in order to help MP find the relation between your documented parameter and the parameter itself.

② It is also important to state where that parameter is, otherwise the documentation will not show up

③ When using the `@Parameter` annotation on attribute level, you don't need to state the name or the place of the parameter, unless you want to achieve some kind of custom result. You can simply add the description and other attributes that need to be explicitly added.

Now when we look at our Get Articles tab, we will see a more descriptive view:



16.3.4. The `@APIResponses` annotation

The next thing we can see incomplete in our "Get articles" endpoint are the responses. They don't tell the whole story. Let's go and fix that.

```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(
            //attributes
    )
    @APIResponses({
            @APIResponse(
                    responseCode = "200",
                    name = "List of articles",
                    description = "The articles that were found",
                    content = @Content( ①
                            schema = @Schema(name = "ArticleDTO", ②
                                    implementation = ArticleDTO[].class,
                                    description = "List of found articles, filtered by
author (if provided).",
                                    example = """
                                            [
                                               {
```

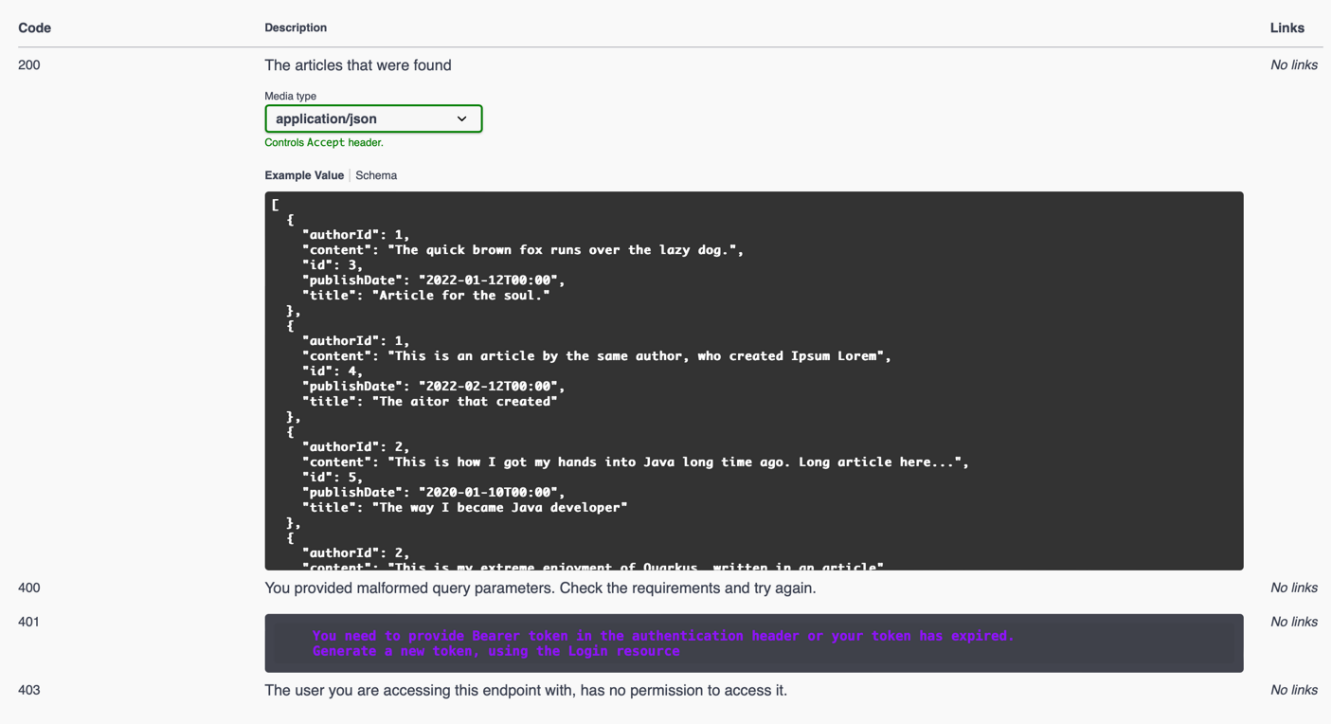
```
                                                    "authorId": 1,
                                                    "content": "The quick brown fox
runs over the lazy dog.",
                                                    "id": 3,
                                                    "publishDate": "2022-01-12T00:00",
                                                    "title": "Article for the soul."
                                                },
                                                {
                                                    "authorId": 1,
                                                    "content": "This is an article by
the same author, who created Ipsum Lorem",
                                                    "id": 4,
                                                    "publishDate": "2022-02-12T00:00",
                                                    "title": "The aitor that created"
                                                },
                                                {
                                                    "authorId": 2,
                                                    "content": "This is how I got my
hands into Java long time ago. Long article here...",
                                                    "id": 5,
                                                    "publishDate": "2020-01-10T00:00",
                                                    "title": "The way I became Java
developer"
                                                },
                                                {
                                                    "authorId": 2,
                                                    "content": "This is my extreme
enjoyment of Quarkus, written in an article",
                                                    "id": 6,
                                                    "publishDate": "2022-09-13T00:00",
                                                    "title": "I love Quarkus and
Quarkus loves me back"
                                                }
                                            ]
                                            """
                                    )
                            )
                    ),
                    @APIResponse(
                            responseCode = "400",
                            name = "Bad Request",
                            description = "You provided malformed query parameters. Check the
requirements and try again."
                    ),
                    @APIResponse(
                            responseCode = "401",
                            name = "Unauthorized",
                            description = """
                            You need to provide Bearer token in the authentication header or
your token has expired.
                            Generate a new token, using the Login resource
```

```
            """
            ),
            @APIResponse(
                    responseCode = "403",
                    name = "Not allowed",
                    description = """
                            The user you are accessing this endpoint with, has no
permission to access it.
                            """
            )
    })
    @Parameter(
            // attributes
    )
    public Response getArticles() {
    }
```

① With the `@Content` annotation we tell our generator what the response object will be, if there is any

② The `schema` attribute and annotation allow us to point to an object that we are going to return. If the object is POJO, the generator will try to generate an example, based on the expected JSON, but if that is not sufficient, we can fill in an `example` by providing value to the respective attribute.

Now we can refresh the page and see the results:

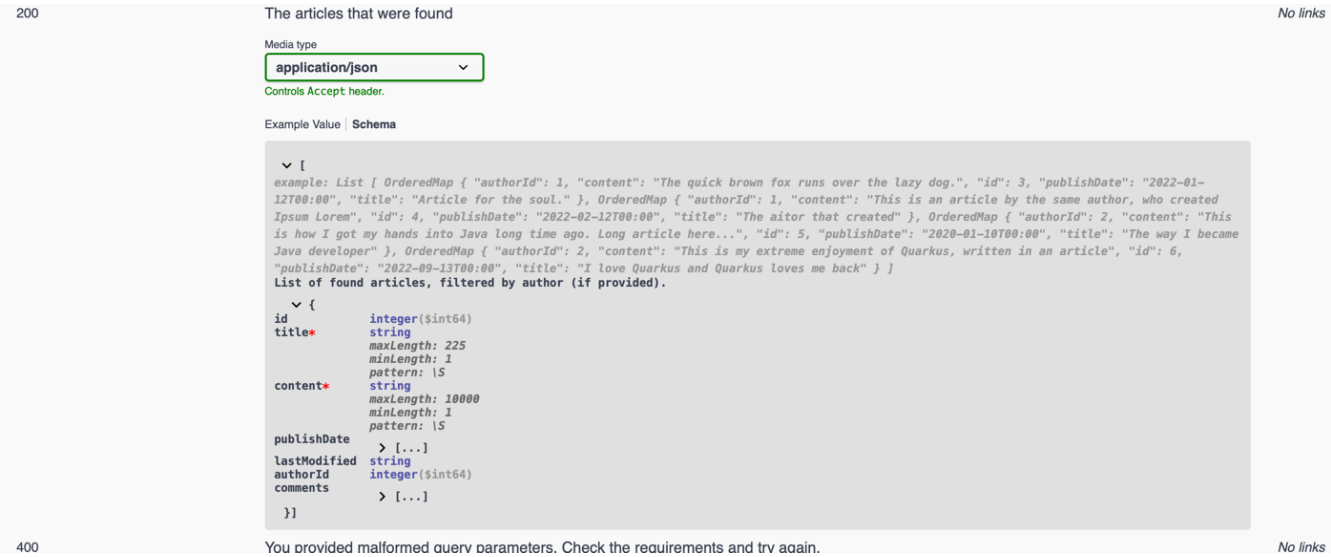


In the next step we are going to dwell into that `@Schema` annotation.

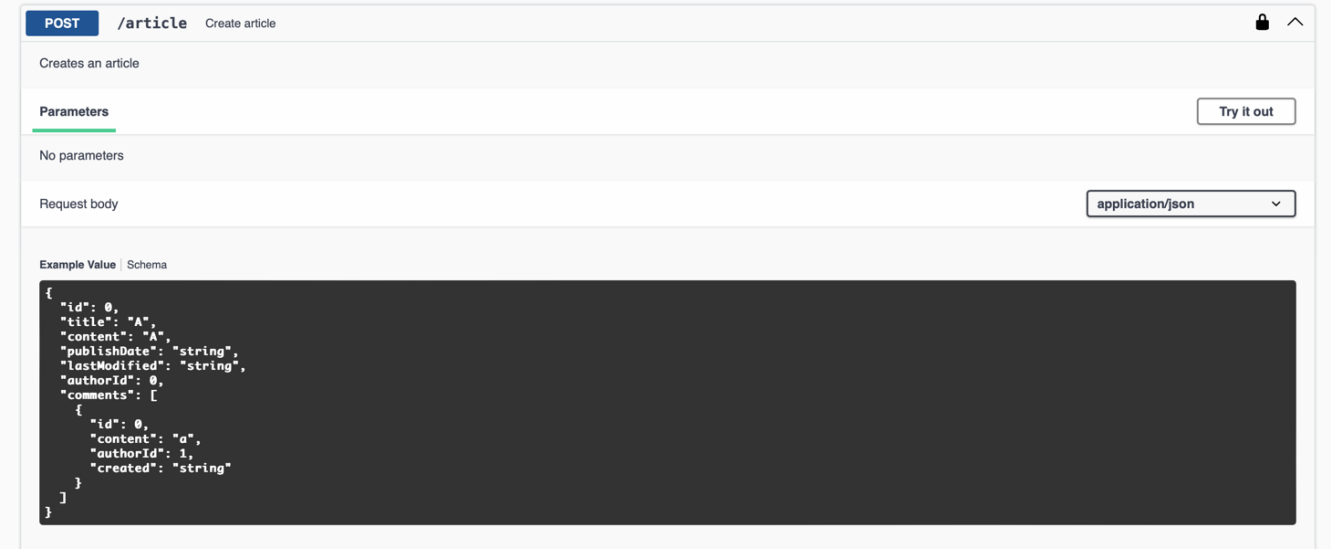
16.3.5. The `@Schema` annotation

This annotation has a bit more capabilities than just providing a reference to a class. We can use it

to describe the fields in that class. If you click on the "Schema" tab, next to "Example value", you will see description of each attribute of the response object is insufficient:



Furthermore, if we open the "Create article" tab, we will see a very bad example of what an article input should look like:



We can fix that, by using the `@Schema` annotation on each object and attribute we want to customize.

In `ArticleDTO`...

```
@Schema(description = "Contains data about the article")
public class ArticleDTO {

    @Schema(description = "The id of the article", example = "1234")
    private Long id;

    @NotBlank
    @Schema(description = "The title of the article", example = "The quick brown fox jumps over the lazy dog!")
    @Size(min = 1, max = 225, message = "The title of the article must be between {min} and {max} characters")
```

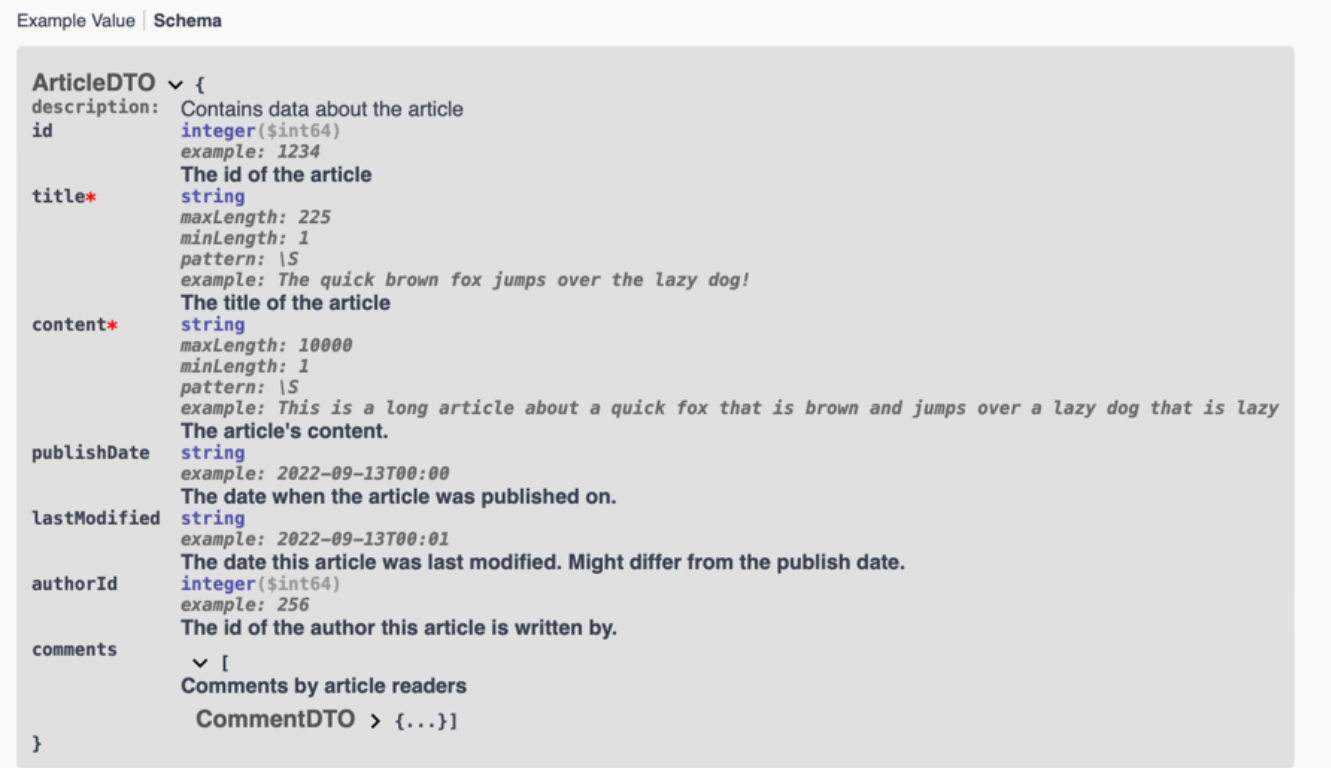
```
    private String title;

    @NotBlank
    @Schema(description = "The article's content.", example = "This is a long article
about a quick fox that is brown and jumps over a lazy dog that is lazy")
    @Size(min = 1, max = 10_000)
    private String content;

    @Schema(description = "The date when the article was published on.", example =
"2022-09-13T00:00")
    private String publishDate;

    @Schema(description = "The date this article was last modified. Might differ from
the publish date.", example = "2022-09-13T00:01")
    private String lastModified;

    @Schema(description = "The id of the author this article is written by.", example
= "256")
    private Long authorId;

    @Schema(description = "Comments by article readers")
    private List<CommentDTO> comments;

    //getters, setters, constructors
}
```

and in `CommentDTO`:

```
@Schema(description = "A comment by any user left on an article")
public class CommentDTO {

    @Schema(description = "The id of the comment", example = "124")
    private Long id;

    @NotBlank
    @Size(max = 255)
    @Schema(description = "The comment's content", example = "I loved this article. It
is true that dogs are lazy when foxes are quick.")
    private String content;

    @NotNull
    @Positive
    @Schema(description = "The id of the comment's author. Can be any type of user",
example = "235")
    private Long authorId;

    @Schema(description = "The date when the comment was published", example = "2022-
09-13T00:01")
    private String created;
```

```
    //getters, setters, constructors
}
```

Now if we refresh the page again, we will see those values updated.



And the automatic example in "Create article" should look a lot more clear:

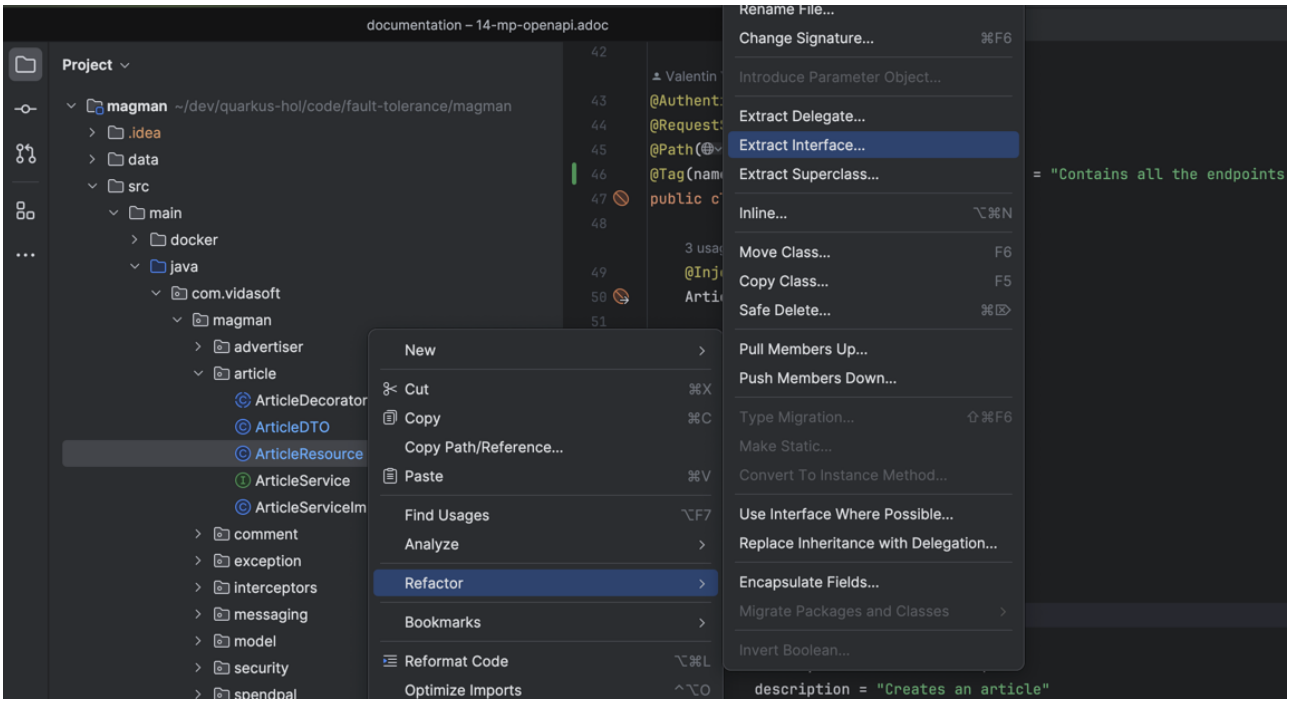
Notice that we are using the same POJOs for creating an article and getting an article. The auto generation of the example will use every value that it finds within the object, despite that we don't accept comments. You can always use the `example` attribute to provide a more accurate example.

16.4. Extracting our OpenAPI endpoints

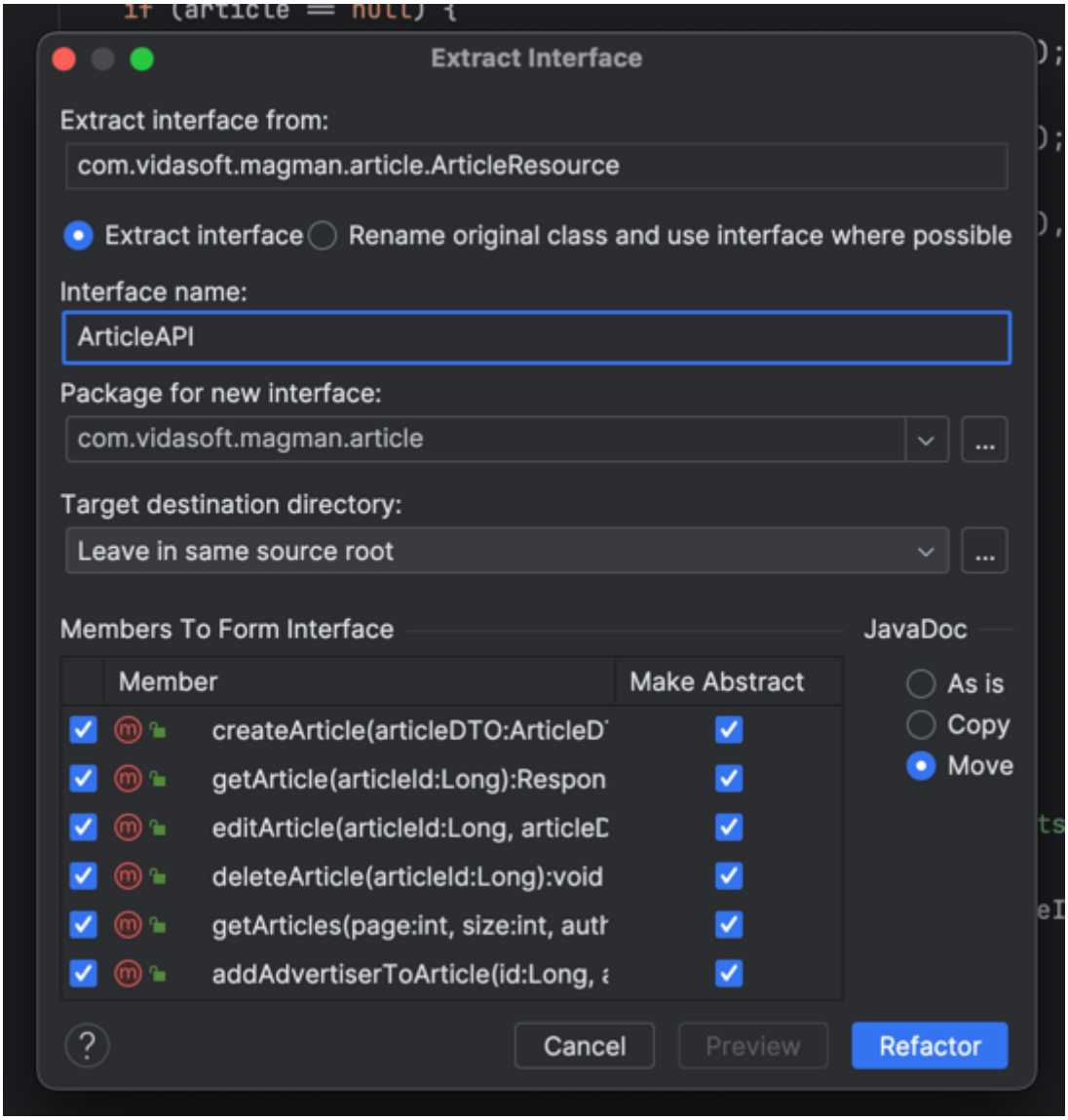
With all the new annotations, what you may have noticed is that our resources have less visible and readable code, but more annotations that cause misalignment of the code and obstruct its readability.

We can fix that by simply defining interfaces where we will place our OpenAPI annotations and constraint validations and leave the implementation in its own class. The simplest way to do this for every resource class you have it to use the power of your IDE. Here's how it's done in IntelliJ:

1. Right click on the resource class and select **Refactor → Extract Interface...**



2. A window will open, where you need to define your preferences



Make sure to select all the methods that are entry point of an endpoint and select **Move** on the JavaDoc option to ger all your OpenAPI annotations inside the interface.

3. When this is done, make sure that all the annotations are copied over to the interface.

At the end you should have an interface looking like this:

```
@Path("/article")
@Tag(name = "Article Resource", description = "Contains all the endpoints, required to
create, update and delete articles.")
public interface ArticleAPI {
    @POST
    @RolesAllowed({Author.ROLE_NAME})
    @Consumes(MediaType.APPLICATION_JSON)
    @Operation(
            operationId = "createArticle",
            summary = "Create article",
            description = "Creates an article"
    )
```

```
    Response createArticle(@Valid @NotNull ArticleDTO articleDTO);

    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(
            operationId = "getArticle",
            summary = "Get article",
            description = "Gets article by its id"
    )
    Response getArticle(@Positive @PathParam("id") Long articleId);

    @PUT
    @Path("/{id}")
    @RolesAllowed({Author.ROLE_NAME})
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(
            operationId = "editArticle",
            summary = "Edit article",
            description = "Updates an article by its id"
    )
    Response editArticle(@Positive @PathParam("id") Long articleId, @Valid @NotNull
ArticleDTO articleDTO);

    @DELETE
    @Path("/{id}")
    @RolesAllowed({Author.ROLE_NAME, Manager.ROLE_NAME})
    @Operation(
            operationId = "deleteArticle",
            summary = "Delete article",
            description = "Removes an article and its related comments by its id"
    )
    void deleteArticle(@Positive @PathParam("id") Long articleId);

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Operation(
            operationId = "getArticles",
            summary = "Get articles",
            description = "Returns a list of articles. Can be filtered by author's id"
    )
    @APIResponses({
            @APIResponse(
                    responseCode = "200",
                    name = "List of articles",
                    description = "The articles that were found",
                    content = @Content(
                            schema = @Schema(name = "ArticleDTO",
                                    implementation = ArticleDTO[].class,
                                    description = "List of found articles, filtered by
```

```
author (if provided).",
                                    example = """
                                            [
                                               {
                                                   "authorId": 1,
                                                   "content": "The quick brown fox
runs over the lazy dog.",
                                                   "id": 3,
                                                   "publishDate": "2022-01-12T00:00",
                                                   "title": "Article for the soul."
                                               },
                                               {
                                                   "authorId": 1,
                                                   "content": "This is an article by
the same author, who created Ipsum Lorem",
                                                   "id": 4,
                                                   "publishDate": "2022-02-12T00:00",
                                                   "title": "The aitor that created"
                                               },
                                               {
                                                   "authorId": 2,
                                                   "content": "This is how I got my
hands into Java long time ago. Long article here...",
                                                   "id": 5,
                                                   "publishDate": "2020-01-10T00:00",
                                                   "title": "The way I became Java
developer"
                                               },
                                               {
                                                   "authorId": 2,
                                                   "content": "This is my extreme
enjoyment of Quarkus, written in an article",
                                                   "id": 6,
                                                   "publishDate": "2022-09-13T00:00",
                                                   "title": "I love Quarkus and
Quarkus loves me back"
                                               }
                                            ]
                                            """
                            )
                    )
            ),
            @APIResponse(
                    responseCode = "400",
                    name = "Bad Request",
                    description = "You provided malformed query parameters. Check the
requirements and try again."
            ),
            @APIResponse(
                    responseCode = "401",
                    name = "Unauthorized",
```

```
                    description = """
                                    You need to provide Bearer token in the
authentication header or your token has expired.
                                    Generate a new token, using the Login resource
                            """
            ),
            @APIResponse(
                    responseCode = "403",
                    name = "Not allowed",
                    description = """
                            The user you are accessing this endpoint with, has no
permission to access it.
                            """
            )
    })
    @Parameter(
            name = "author",
            in = ParameterIn.QUERY,
            description = "Filter articles by author id. If empty, will return all
articles",
            example = "123"
    )
    Response getArticles(@Parameter(required = true, description = "Search result
page, starts from 1") @QueryParam("page") @DefaultValue("1") @Positive int page,
                         @Parameter(required = true, description = "Size of the search
result page. Cannot be 0") @QueryParam("size") @DefaultValue("10") @Positive int size,
                         @QueryParam("author") @Positive Long authorId);

    @PATCH
    @RolesAllowed({Manager.ROLE_NAME})
    @Path("{id}/advertiser/{advertiserId}")
    @Operation(
            operationId = "addAdvertiserToArticle",
            summary = "Promote article",
            description = "Adds advertiser to the article"
    )
    Response addAdvertiserToArticle(@Positive @PathParam("id") Long id, @Positive
@PathParam("advertiserId") Long advertiserId);
}
```

After you've done so, go over `ArticleResource` class and remove every annotation that is present in the newly created interface. At the end the class should look like this:

```
@RequestScoped
public class ArticleResource implements ArticleAPI {

    @Inject
    ArticleService articleService;

    @Inject
```

```
    @LoggedUser
    User loggedUser;


    @Override
    @Transactional
    public Response createArticle(ArticleDTO articleDTO) {
        //implementation
    }


    @Override
    public Response getArticle(Long articleId) {
        //implementation
    }


    @Override
    @Transactional
    public Response editArticle(Long articleId, ArticleDTO articleDTO) {
        //implementation
    }


    @Override
    @Transactional
    public void deleteArticle(Long articleId) {
        //implementation
    }


    @Override
    public Response getArticles(int page, int size, Long authorId) {
        //implementation
    }


    @Override
    @Transactional
    public Response addAdvertiserToArticle(Long id, Long advertiserId) {
        //implementation
    }
}
```

Pretty neat, huh?

16.5. Securing the OpenAPI documentation

OpenAPI can be very useful tool, but our current configuration has a bit of a security issue. If we don't want to share our APIs publicly, but just internally, we might not want users to be able to call

`/q/openapi` and download and exploit our APIs. This is why we can prevent this from happening.

By default, when you run quarkus in **prod** mode (directly run the application jar or set `quarkus.profile=prod`), quarkus will turn off the Swagger-ui page, but this will not disable the generation of the openapi file. To disable that, you can simply add `%prod.quarkus.smallrye-openapi.enable=false` into the application.properties file and recompile the application.

There is also another way to secure your OpenAPI endpoints, by using authentication methods or even configure the paths of the documents. You can read the full configuration reference in the official Quarkus article.

16.6. Conclusion

MP OpenAPI has a couple of more annotations which are not so mandatory, but can help you to achieve more capabilities with your yaml generation. You can always refer to the official documentation to find out how to do more stuff and achieve different results. Here are some useful links to look at:

- The official MP OpenAPI documentation - https://download.eclipse.org/microprofile/microprofile-open-api-1.0/microprofile-openapi-spec.html
- Another tutorial on documenting your applications using MP OpenAPI - https://openliberty.io/guides/microprofile-openapi.html

While at it, why don't you go through every endpoint in your applications and refactor them to look prettier in OpenAPI? Come on! It will be fun!

Chapter 17. Monitoring application health with Micrometer

In this final chapter of our hands-on we are going to look at how we can monitor the health and well-being of our application. Being able to know how our application performs is crucial for maintaining its resources and address any potential issues that our application might have, based on the performance.

With the Micrometer extension you are able to do exactly that, by enabling some default metrics such as CPU and RAM utilization and also being able to define your own monitoring trackers, such as timers, counters and gauges. Then you can export the data tools, such as Prometheus and use the export file to build dashboards, charts and alerts, based on the data extracted from our metrics preferences.

17.1. Configuring Micrometer with our Quarkus application

Jus as we did with every chapter that's ever existed in this hands-on, we are going to add yet two more extensions:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-micrometer</artifactId> ①
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-micrometer-registry-prometheus</artifactId> ②
</dependency>
```

① Adds the core features that enable Micrometer

② Adds support for Prometheus monitoring

This should be enough to get things up and running. Now if you start your project and connect to `http://localhost:8080/q/metrics`, you should be able to see all the preloaded metrics for the application. These are mainly JVM, system and HTTP related metric trackers, obtained by the JVM itself or the quarkus extensions we already use and support metrics right out of the box.

17.2. Creating custom metrics

This is the part where we are going to look into making our own metric properties. For example if we want to track how many times the createArticle method was called, we would create a metric of type **counter**, which is going to track every call to the `createArticle` method, or we want to track how much time it takes to create a new user, then we would use a **timed** metric, that will measure the time for the method to return its value. And lastly if we want to check how much active pending payments are waiting to be processed, we can create a **gauge** tracker.

Each of these monitoring metrics has several ways to be created within our application. Let's look at them one by one, and see what fits our needs.

17.3. Counters

Counters are the most basic type of metrics. These are simply doing what their names says. They count.

Defining a counter metric on a method invocation will start counting each time that method has been invoked. Such data may help to decide what is the importance of a functionality in your code or help you see if a functionality isn't called as much and encourage you to further investigate why.

In our MagMan application we can see fit for the counter function in many places, including:

- Checking how many users have called the login or register endpoint
- Checking how many articles or comments are created in real time
- Checking if a part of our code is called to many times - such as our fallback for payments

Based on this we can decide to increase our server resources to welcome more customers or check if something is wrong with our Kafka configuration and address it sooner rather than later.

There are three ways to create a counter:

1. With annotation - like every part of our Jakarta EE experience there's an annotation for it. To add a metric to a specific method to your code, you simply add the `@io.micrometer.core.annotation.Counted` annotation:

   ```
   @Override
   @Transactional
   @Counted
   public Response registerUser(NewUserDTO newUserDTO) {
       //implementation
   }
   ```

   Now if we go to `/q/metrics` and search for "register", we will find nothing. That is expected. The `/q/metrics` page shows data in real time and if we have never called the `registerUser` method at least once, there will be no data visible for it.

   The next step is to call `/user` endpoint and refresh the metrics page, and right away we are going to find this metric

   ```
   # TYPE method_counted_total counter
   method_counted_total{class="com.vidasoft.magman.user.UserResource",exception="none"
   ,method="registerUser",result="success",} 1.0
   ```

   You can personalize your `@Counted` annotation by giving a specific name to the metric for example `@Counted("register_user_endpoint")`. This will create a

separate dedicated counter in your Prometheus log:

```
# HELP register_user_endpoint_total
# TYPE register_user_endpoint_total counter
register_user_endpoint_total{class="com.vidasoft.magman.user.UserRes
ource",exception="none",method="registerUser",result="success",} 1.0
```

2. Use a convenience method on the `MeterRegistry` - this is a CDI-managed class that we can inject in our resource or service and directly add a counter to it

```
@RequestScoped
public class UserResource implements UserAPI {

    //other declarations

    @Inject
    MeterRegistry registry; ①

    Counter loginCounter; ②

    @PostConstruct
    void init() {
        String requestPath = request.uri();
        var originIp = request.remoteAddress().toString();
        logger.log(Level.INFO, "URL call attempt {0} from {1}", new String[]
{requestPath, originIp});
        loginCounter = registry.counter("login_user_endpoint"); ③
    }

    @Override
    @Transactional
    @Counted("register_user_endpoint")
    public Response registerUser(NewUserDTO newUserDTO) {
        //implementation
    }

    @Override
    public Response loginUser(LoginDTO login) {
        loginCounter.increment(); ④
        Optional<User> loggedUser = userService.loginUser(login.getUserName(),
login.getPassword());
        return loggedUser
                .map(u -> Response.ok(new UserDTO(u))
                        .header("Authorization", jwtService.generateJWT(u))
                        .build())
                .orElseGet(() -> Response.status(Response.Status.UNAUTHORIZED)
.build());
    }
```

```
}
```

① First we need to inject out metric registry. (You can also get the instance form `Metrics.globalRegistry`, if you're not in a CDI managed class.)

② The next step is to declare our counter. This strategy can be used for Timers and Gauges as well.

③ We have to register that counter with our registry to obtain its instance.

④ Finally when we need to increment it, we place the `.increment()` invocation to the point where we want to increment.

3. By calling the `.builder()` method - this method, gives you more customization abilities, but at the end serves the same function:

```
Counter loginCounter;
Counter successfulLoginCounter;

@PostConstruct
void init() {
    String requestPath = request.uri();
    var originIp = request.remoteAddress().toString();
    logger.log(Level.INFO, "URL call attempt {0} from {1}", new String[]
{requestPath, originIp});
    loginCounter = registry.counter("login_user_endpoint");
    successfulLoginCounter = Counter.builder("login_user_endpoint_success")
            .baseUnit("Schmeckle") ①
            .register(registry);
}

//more code

@Override
public Response loginUser(LoginDTO login) {
    loginCounter.increment();
    Optional<User> loggedUser = userService.loginUser(login.getUserName(), login
.getPassword());
    if (loggedUser.isPresent()) {
        successfulLoginCounter.increment();
        return loggedUser.map(u -> Response.ok(new UserDTO(u))
                .header("Authorization", jwtService.generateJWT(u))
                .build()).orElse(null);
    } else {
        return Response.status(Response.Status.UNAUTHORIZED).build();
    }
}
```

17.4. Timers

The next custom metering tool we are going to look at is timers. Timers are useful when you want to track how much time it takes for an operation to complete. This could come in handy once your services starts getting high loads of traffic. It could help you to see if there are any latencies more than the expected and act accordingly by optimizing the performance of your application or addressing issues with third parties.

Similarly to the counters, you have three ways to set a timer to your application.

1. Using the `@io.micrometer.core.annotation.Timed` annotation - simply add this annotation on the method you want to track the execution time of:

```
@Timed(value = "charge_subscriber_execution_time", description = "Tracks how much
time it takes to charge a subscriber")
public boolean chargeSubscriber(Subscriber subscriber) {
    Subscription subscription = createSubscription(subscriber);
    //implementation continues...
}
```

   Then on the metrics page you will be able to see some handy information about the endpoint's usage, such as the maximum time it took to perform the operation, the times this method was invoked and the total time this operation has been run.

```
# HELP charge_subscriber_execution_time_seconds_max Tracks how much time it takes
to charge a subscriber
# TYPE charge_subscriber_execution_time_seconds_max gauge
charge_subscriber_execution_time_seconds_max{class="com.vidasoft.magman.subscriptio
n.PaymentService",exception="none",method="chargeSubscriber",} 0.0061069
# HELP charge_subscriber_execution_time_seconds Tracks how much time it takes to
charge a subscriber
# TYPE charge_subscriber_execution_time_seconds summary
charge_subscriber_execution_time_seconds_count{class="com.vidasoft.magman.subscript
ion.PaymentService",exception="none",method="chargeSubscriber",} 6.0
charge_subscriber_execution_time_seconds_sum{class="com.vidasoft.magman.subscriptio
n.PaymentService",exception="none",method="chargeSubscriber",} 0.2530685
```

2. By invoking the registry - unlike the counters, this option is a bit different, as it requires to step things up a little, by first tracking the time and then reporting it to the registry:

```
private Timer createCommentTimer;

@PostConstruct
void init() {
    createCommentTimer = registry.timer("comments_create_comment");
}
```

```
@Override
@Transactional
public Response createComment(Long articleId, CommentDTO commentDTO) {
    Timer.Sample sample = Timer.start();
    //same old implementation...
    sample.stop(createCommentTimer);

    return Response.created(URI.create(String.format("/article/%d/comment/%d",
articleId, comment.id))).build();
}
```

It should be obvious that in order to track time in a more programmable way, we need to define where the timer starts and where it ends.

3. Finally, if we want more control on the timer definition, we can again rely on the builder methods the `Timer` class comes with:

```
@Inject
MeterRegistry registry;

private Timer createCommentTimer;
private Timer getCommentsTimer;

@PostConstruct
void init() {
    createCommentTimer = registry.timer("comments_create_comment");
    getCommentsTimer = Timer.builder("comments_get_timer")
            .tag("comments_resource", "get_comments")
            .register(registry);
}

@Override
@Transactional
public Response createComment(Long articleId, CommentDTO commentDTO) {
    Timer.Sample sample = Timer.start();
    //implementation...
    sample.stop(createCommentTimer);

    return Response.created(URI.create(String.format("/article/%d/comment/%d",
articleId, comment.id))).build();
}

@Override
public List<CommentDTO> getCommentsForArticle(Long articleId) {
    return getCommentsTimer.record(() -> Comment.findByArticleId(articleId)
            .stream().map(CommentDTO::new).toList());
}
```

Notice that there are different ways to start a timer. We can either get a class what

> has a `start()` and `stop()` method or wrap the content we want to track time of in a lambda function. The decision on which one to use, lies on how complicated the task is. You can find more about the different ways to run timers here.

17.5. Gauges

The final metering tool we are going to look at is the gauge. Gauges are convenient metering values that show us current state of something that could increase or decrease. For example the amount of RAM usage of your computer could be gauged or the usage of the CPU. Values like this help you make the decision whether to close an application to clear up more space or find what's bottlenecking the performance of your machine.

In our application a smart place to put a gauge would be to track the number of pending transactions. Then, if that number goes above a certain threshold, we can use it to trigger an alert that will tell us that there is something wrong with the payment processing services.

Unlike the counters and the timers, the gauge doesn't have its own annotation, as it doesn't need to be triggered upon a method call, because it has more specific uses that require different approach in creating and reporting information to them. There are two ways to use a gauge:

1. actively - this requires a convenient place to create your gauge and bind a function to it, which it will call every time the `/metrics` page is opened:

```
public class PaymentService {

    private static final Logger LOGGER = Logger.getLogger(PaymentService.class
.getName());

    @Inject
    EventBus eventBus;

    @Inject
    KafkaMessageService kafkaMessageService;

    @Inject
    MeterRegistry registry;

    public void buildPendingTransactionsGauge(@Observes StartupEvent startupEvent)
{
        Gauge.builder("subscriptions_pending", Subscription:
:countPendingSubscriptions).register(registry); ①
    }

    //further implementations
}
```

① We assume that we have already defined a method to obtain all pending subscriptions.

2. passively - instead of doing a call which might involve dedicating the time and resources of our

environment, the gauge can cling onto a method that is called often enough and steal the data from there:

```
@ApplicationScoped
public class KafkaMessageService {
    private static final Logger LOGGER = Logger.getLogger(KafkaMessageService.
class.getName());

    private Set<PaymentPayload> pendingMessages = ConcurrentHashMap.newKeySet();
//We are adding entries from different threads, so we need a set that could support
it
    //more imports and methods here...

    @Retry
    @Timeout(5000)
    @Fallback(fallbackMethod = "chargeSubscriberThroughRest")
    @CircuitBreaker(requestVolumeThreshold = 3)
    @Asynchronous
    public CompletionStage<Void> sendPaymentsMessage(PaymentPayload payload) {
        pendingMessages.add(payload);
        registry.gaugeCollectionSize("payments_to_send", List.of(Tag.of("Tag1",
"Tag1")), pendingMessages); ①
        LOGGER.info("Attempting to send payment message");
        String payloadString = JsonbBuilder.create().toJson(payload);
        paymentsEmitter.send(payloadString).toCompletableFuture().join();
        LOGGER.info("Successfully emitted message to payments topic: %s".formatted
(payloadString));

        registry.gaugeCollectionSize("payments_to_send", List.of(Tag.of("Tag1",
"Tag1")), pendingMessages); ②
        pendingMessages.remove(payload);
        return CompletableFuture.completedFuture(null);
    }
}
```

① Creating the gauge entry for the first time with the registry requires to add its initial or current value

② We can then update that entry as we need to.

In this scenario we are waiting for `paymentsEmitter.send` to complete its operation. Since `sendPaymentsMessage` is not synchronized, it can be called as many times as the resources of the application allow. This is a convenient point where we can keep track on the current amount of messages that are waiting to be sent. Now think of the real world scenario - being able to tell that something is wrong with the messaging service, before too many messages add up, and act upon it.

17.6. Compatibility with MP Metrics

Formally Quarkus used to work with the MicroProfile Metrics specification. If you choose to use the annotations from MP Metrics you are free to do so, as long as you configure the appropriate dependencies. You can find out more on how to use MP Metrics annotations with Micrometer on Quarkus here.