Tutorials — JWS

version #

Yet Another Kind of Assistants 2024 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.forge.epita.fr

This file contains details about the architecture you will be using for this project along with the annotations you may need while working on it.

1 The Layered Architecture

This is commonly called the "layered architecture", and is a staple of backend development. You may find implementations across a wide variety of tools that follow the pattern for each layer and associated object type. The lack of standardization in naming should not be an obstacle to your comprehension of the pattern and your ability to recognize it.

1.1 Controllers

Controllers are the doors for external applications to interact with you backend. Controllers's goals are the following ones:

- Expose methods used by Quarkus to create endpoints;
- Tell the client if something went wrong with the request;
- Get a result from a service and send a response to the user.

These responses and requests are objects called **Data Transfer Objects**.

Be careful!

Your Controllers should **never** be able to interact with Models.

1.2 Services

Services are the heart of your server, it is where all the logic happens. Services are not always bound to only one Repository since the logic and Entities may need information from different Models.

1.3 Repositories

Repositories of your server are where you will implement all database-related methods, such as retrieving data from the database for example.

Be careful!

You should not implement any logic in the Repositories. They only aim at interacting with the database and retrieving the needed data. Therefore you **must** have at most one Model per Repository.

1.4 Data Transfer Objects

DTOs are the objects managed by the Controllers.

There are two types of DTOs:

- Response DTOS;
- Request DTOS.

Data Transfer Objects (DTOs) should only be used in the Controllers. They are contracts that the client must follow in their requests and that the API must adhere to for its responses.

One of the main reasons DTOs exists is to separate presentation logic from business logic. This allows for business updates without the need to update the whole API for a simple service layer change, as it allows for decoupling. Of course, API changes for valid reasons. These changes should be made and communicated properly.

In our case, DTOs are automatically handled by Quarkus.

1.5 Entities

Entities are value objects managed by the Services. They are only used in the Services and Controllers and allow communication between the Controllers and Services. In many cases, especially in the early days of a project, Entities are similar to the underlying Models. However, some Services are aggregation services, and their responsibility is to compute aggregated data (statistics, processing, presentation, etc.). Additionally, even "mapping" Services will diverge from their Models over time to accommodate business needs, which justifies the separation of the layer.

Tips

Some Entities are similar to their mapping Models, but not all Services are mapping Services, and even mapping Services deviate from their Model in time.

1.6 Models

Models are objects managed by the Repositories. They should never be used outside of Services and Repositories. They allow communication between Services and Repositories. A Model is the object representation of a database entry.

2 Given annotations

2.1 Contexts and Dependency Injection

The instantiation of classes in order to use them can be repetitive. To simplify code, Quarkus, uses dependency injections.

Imagine that we have a class `Logger`, which allows us to log messages. We may want to inject it in other classes to use it. Then we can write this code :

```
@ApplicationScoped
class Logger(){
    public void log(String message){
        // Logging function
    }
}

class MyService() {
    @Inject Logger logger;
    public void doSomething() {
        logger.log("Hello World!");
    }
}
```

Here, the annotation `@ApplicationScoped` precise that we want to be able to instantiate a Logger whenever we want in any class.

The annotation `@Inject` allows us to get this instance. The default behavior of Quarkus is to generate only one instance of the class and then give the same instance to every classes that need it.

2.2 Database annotations

As said earlier, we will use Hibernate and JPA to handle data persistence for us. You **must** not write SQL queries by yourself in this project.

We will give you an example of how you should create your models. In our example we will create a student. Don't be afraid by the following code, we will explain it step by step.

```
@Entity @Table(name = "student")
class Student() {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY) public Long id;
    public String name;
    public String email;
    public @ElementCollection @CollectionTable(name = "student_roles") List<String> roles;
}
```

Here, we create a Model (or a Entity in JPA's documentation) for our student. The annotation `@Entity` allows us to define this class as a data model that the application will handle.

The annotation `@Table(name = "student")` defines the name of the table where the model will be stored later on.

The annotation `@Id` allows us to define the primary key of our model.

The previous annotation goes hand-to-hand with the annotation `@GeneratedValue(strategy = GenerationType.IDENTITY)` which defines the strategy to generate primary keys.

The annotation `@ElementCollection` tells that this field is a collection of elements, in our case it is a list of roles in a String format.

Some other annotations such as `OneToMany` or the `ManyToOne` can be useful. You may find more information about them in the following documentation : https://quarkus.io/guides/hibernate-orm

2.3 Repository and model management

To create your repositories, you should read the following documentation about `PanacheRepositoryBase` : https://quarkus.io/guides/hibernate-orm-panache

To update your database, your functions **must** be annotated with `@Transactional`. You will find more information about this annotation in the previous documentation link.

2.4 Rest API annotations

Here is a list of annotations that you will use to create your REST API.

- `@Path`: This annotation defines the URI for a method.
- `@GET`: This annotation specifies that the following method is a GET method.
- `@POST`: This annotation specifies that the following method is a POST method.
- `@PATCH`: This annotation specifies that the following method is a PATCH method.
- `@Produces` / `@Consumes`: Those annotations define the type of data, which correspond to the mediatype, that the method will respectively return or accept as input.
- `@PathParam`: This annotation allows us to retrieve an attribute of the request.

2.5 Miscellaneous annotations

- `@Value`: This annotation indicates that your class is value object, it generates automatically its setters, getters and constructors.
- `@With`: This annotation allows us to add methods like with*AttributeName* for each attribute of our class. We can then chain calls to theses methods to get a new instance of the class initialized with the values we passed to the *with* methods earlier.

```
@With
class FooBar() {
    public String foo;
    public String bar;
}

class Main() {
    public static void main(String[] args) {
        new FooBar().withFoo("foo").withBar("bar");
    }
}
```

- `@ConfigProperty`: This annotation allows us to retrieve the value of an environment variable.

```
@ConfigProperty(name="foobar") String foobar;
```

You mean it's working? For real this time?