Exercise — JWS

version **#1.1**

Yet Another Kind of Assistants 2025 <assistants@tickets.assistants.epita.fr>

Copyright

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

Contents

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

File Tree

```
jws/
└─ src/
   └─ main/
      └─ java/
         └─ fr/
            └─ epita/
               └─ assistants/
                  └─ jws/
                     ├─ converter/
                     │  └─ **.java  (to submit)
                     ├─ data/
                     │  ├─ model/
                     │  │  └─ **.java  (to submit)
                     │  └─ repository/
                     │     └─ **.java  (to submit)
                     ├─ domain/
                     │  ├─ entity/
                     │  │  └─ **.java  (to submit)
                     │  └─ service/
                     │     └─ **.java  (to submit)
                     ├─ errors/
                     │  └─ **.java  (to submit)
                     ├─ presentation/
                     │  └─ rest/
                     │     ├─ *.java  (to submit)
                     │     ├─ request/
                     │     │  └─ **.java  (to submit)
                     │     └─ response/
                     │        └─ **.java  (to submit)
                     └─ utils/
                        └─ **.java  (to submit)
```

Obligations

*Obligations are **fundamental** rules shared by all subjects. They are non-negotiable and to not apply them means to face sanctions. Therefore, do not hesitate to ask for explanations if you do not understand one of these rules.*

Obligation #0: **Cheating**, as well as sharing source code, tests, test tools or coding-style correction tools is **strictly forbidden** and penalized by not being graded, being flagged as a cheater and reported to the academic staff.

Obligation #1: Your submission repository must be **clean**. Except for special cases, which (if any) are **explicitely** mentioned in this document, an *unclean* repository may contain:

- binary files;[1]
- files with inappropriate privileges;
- forbidden files: `*~`, `*.swp`, `*.o`, `*.a`, `*.so`, `*.class`, `*.log`, `*.core`, etc.;
- a file tree that does not follow our specifications.

[1] If an executable file is required, please provide its sources **only**. We will compile it ourselves.

1 Introduction

1.1 Bomberman

Bomberman is a video game where the player must defeat enemies by placing bombs that explode in cardinal directions after a delay to destroy obstacles and kill enemies. Players can also be killed if they are caught in a bomb explosion, including their own.

1.2 Goal

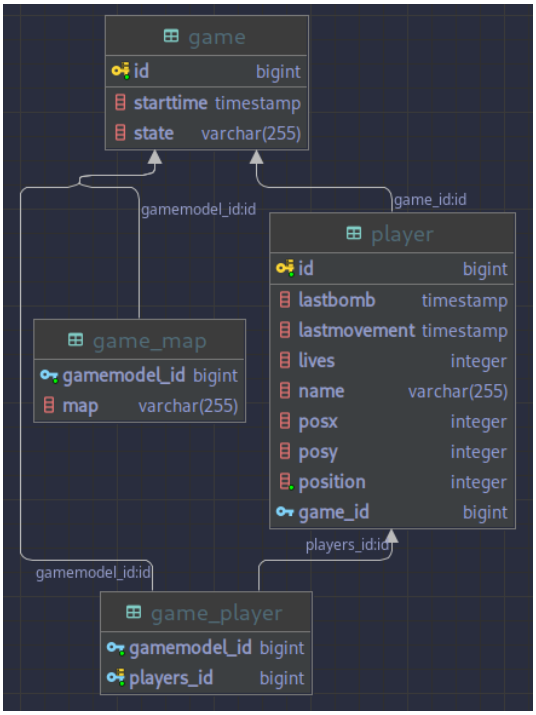
The goal of this project is to create the backend of an online Bomberman game. You **must** implement multiple REST endpoints for the client to communicate with the backend.

Those endpoints are all defined in a Swagger document given in YAML format on the intranet.

Your code will be tested in two different ways :

- **Architecture tests: the architecture of your project will be tested.**
  Therefore, it should comply with the *architecture* section of the subject.
- **API tests: your backend should send proper responses to requests.**
  Those requests and responses are all defined in the JSON, in the Swagger document, and explained in the *paths* section.

The following diagram is an example of the database that you can implement in this project.



In this project, you may encounter concepts that you may not have seen before, such as ORM and REST API. To fully understand these concepts, you should look at the corresponding courses available on Moodle.

2 Frameworks

For this project you will be using two different frameworks: Quarkus and Hibernates.

2.1 Quarkus

Quarkus was designed for ease of use right from the beginning. It has features that work well with minimal configuration and is considerably less opinionated and more lightweight than other frameworks. You can use an active record pattern and define your functionalities as entity methods, use a repository pattern, or use services to implement them easily with minimal setup. Quarkus is optimized for memory density and rapid startup time. Applications running on the JVM with Quarkus can deliver nearly twice as many instances in the same amount of RAM when compared to other cloud-native Java stacks, and up to 7 times more instances when packaged as a native binary.

2.2 Hibernates

Hibernate ORM is the de facto standard JPA implementation and provides a fully featured Object Relational Mapper (ORM). It works beautifully in Quarkus and is a powerful tool that allows you to define your entities and relationships in a declarative way.

3 Architecture

Adhering to the right architectural pattern is the main goal of this project. It will be a valuable knowledge if you have to work on the backend of an intranet or an application during your internship, for example.

Your project **should** respect the architecture provided in the given files.

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.

3.1 Controllers

Controllers are the doors for external applications to interact with your 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.

3.2 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.

3.3 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.

3.4 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.

3.5 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.

3.6 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.

4 Given annotations

4.1 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.

4.2 Miscellaneous annotations

- `@Value`: This annotation indicates that your class is a value object. It generates automatically its getters and constructors, as well as other useful methods.
- `@With`: This annotation allows us to add methods like with*AttributeName* for each attribute of our class. We can then chain calls to these 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;
```

5 How to start

5.1 Set-up the database

First, let us configure a PostgreSQL-specific environment variable:

```
42sh$ echo 'export PGDATA="$HOME/postgres_data"' >> ~/.bashrc
42sh$ echo 'export PGHOST="/tmp"' >> ~/.bashrc
42sh$ source ~/.bashrc
```

This first line adds a PGDATA environment variable to your .bashrc, containing the location of your choice to store PostgreSQL's data. The second specifies the host name of the machine on which the server will run. As the value begins with a slash, it will be used as the directory containing a socket on which postgres will listen. Then, let us initialize a new PostgreSQL database cluster. It will generate default databases and configurations in the $PGDATA directory.

```
42sh$ nix-shell -p postgresql
42sh$ initdb --locale "$LANG" -E UTF8
```

Be careful!

If you experience a permission denied error, you may need to restore your rights using `42sh$ chmod 755 ~`

5.2 Create a database

Your server is now up and running! You have to set the `DB_USERNAME` variable to your login. PostgreSQL offers an interactive shell that connects to this server and behaves as a front-end for your operations on databases:

```
42sh$ export DB_USERNAME=<login>
42sh$ postgres -k "$PGHOST"
42sh$ psql postgres
```

`psql` takes the name of the database you want to connect to. Here, we choose the default database, `postgres`. These commands will create a brand new database named `jws`.

```
-- Give yourself all the rights
-- If you are not on the PIE, <login> should be your username
postgres=# ALTER ROLE "<login>" SUPERUSER;
-- Create a database named jws
postgres=# CREATE DATABASE jws OWNER <login>;
-- Exit PostgreSQL
postgres=# \q
```

After that you have to create a schema named `jws`.

5.3 Given Files

In order to start the project in good condition, a tarball containing all the necessary files for proper project functioning is available on the intranet.

You **should** decompress this tarball and open the directory with IntelliJ before reading the upcoming sections of the subject.

Once your project is opened in your IDE, you can type the following command to start the server.

```
42sh$ mvn quarkus:dev
```

It will launch your API on port 8082. We have provided you with a Hello World endpoint, so the next command will return `"Hello World!"`.

```
42sh$ curl http://localhost:8082
```

To launch the viewer, you can type the following command from the root of your project:

```
42sh$ java -jar front-end.jar
```

You can now use your viewer through your browser at the following URI : `http://localhost:3000`

A Swagger of the project is given for you to see the different endpoints you have to implement, error codes, and the answers' format. You will find it in `src/main/resources/openapi.yaml`, you can use a website such as https://editor.swagger.io to see it in better conditions.

In order to pass our architecture tests, you **must** respect the following architecture :

- `data` : This directory **must** contain all the logic related to the model layer

  - `model`: This directory **must** contain all your models
  - `repository`: This directory **must** contain all you repositories
- `domain`: This directory **must** contain all the logic related to the service layer
  - `service`: This directory **must** contain all your services
  - `entity`: This directory **must** contain all your entities
- `presentation`: This directory **must** contain all the logic related to your controllers.
  - `rest`: This directory **must** contain all your REST controllers
    * `request`: This directory **must** contain your request DTO
    * `response`: This directory **must** contain your response DTO

In order to make your program work, you **must** handle the following environment variables:

- `JWS_MAP_PATH`: A path to a map in RLE format, you can find a default one at this path `src/test/resources/map1.rle`
- `JWS_TICK_DURATION`: The duration of a tick in ms
- `JWS_DELAY_MOVEMENT`: The delay in tick between two movements
- `JWS_DELAY_BOMB`: The delay in tick between two bombs set down, it is also the delay before the bomb explodes
- `JWS_DELAY_FREE`: The delay in tick before the map starts to shrink
- `JWS_DELAY_SHRINK`: The delay in tick between each map reduction

Be careful!

Be careful, your code will be tested with the following commands:

```
42sh$ mvn package -Dquarkus.package.type=uber-jar -DskipTests
42sh$ java -jar target/jws-1.0-runner.jar
```

They do not have the exact same behavior as

```
42sh$ mvn quarkus:dev
```

The first command will take longer to complete, hence, you should use the first batch of commands only to make sure your code will behave as expected on the moulinette, and the second to make quick checks.

6 Endpoints explanation

6.1 List games

This endpoint lists all the games registered in the database. The format to follow is in the provided Swagger.

6.2 Game creation

This endpoint creates a new game and a first player with the name provided in the request. The initial state of the game **must** be `STARTING`.

Players must spawn at the following points with 3 lives:

- Top left (x=1, y=1)
- Top right (x=15, y=1)
- Bottom right (x=15, y=13)
- Bottom left (x=1, y=13)

IDs of games and players must start at 1.

The map is passed in a simplified RLE format in the response, the length before each kind of block is less than 10.

The blocks should have this name in the RLE file :

- “M” -> Metal
- “W” -> Wood
- “G” -> Ground
- “B” -> Bomb

6.3 Get a specific game

This endpoint gathers informations about a specific game. All the format details of this endpoint are listed in the Swagger.

6.4 Join a game

All the needed information is in the Swagger. Players **must** spawn at the points defined in the *game_creation* part.

6.5 Start a game

You should update the state of your game to `RUNNING`. Once again, all the remaining information about this endpoint is in the Swagger.

6.6 Move

This action allows the player to move. The movement is done in the given direction.

> **Be careful!**
>
> The player can only move in a cardinal direction and cannot go through walls nor through bombs.

The remaining behavior of this endpoint is defined in the given Swagger.

6.7 Set down a bomb

This endpoint allows the player to place a bomb. The bomb can only be placed where the player currently is. If all players are dead, the state of the game should be updated to `FINISHED`.

The bomb explodes in the cardinal directions within a range of a single block. The explosion destroys only wooden blocks and takes a life from the players within its radius. You must ensure that the bomb explodes precisely JWS_DELAY_BOMB ticks after it has been planted.

The bomb's explosion will be tested separately from the process of placing them down.

7 Advanced Features

7.1 Shrinking

> **Be careful!**
>
> To implement this feature, we strongly recommend that you have completed all endpoints. It is essential that all endpoints, up to and including "move", function flawlessly.

This feature allows you to reduce the size of the map. If the game drags on, the map will start to shrink. Players are given a free period of time (c.f JWS_DELAY_FREE), after which the map starts to shrink every interval (c.f JWS_DELAY_SHRINK).

The map reduction method is as follows: The map's playing area's length and width get decremented by one, all the blocks around the boundary must be replaced by metal blocks. When a player is hit by the zone, their lives are reduced to zero.

The following images show the map before and after the first iteration of the shrinking.

Success is granted to those who seek the light and fight the shadows.