Gym Administration System

Development of a Java SE GUI application using TDD, PIT, Coveralls, SonarCloud and GitHub Actions

Jason Ravagli
jason.ravagli@stud.unifi.it

Advanced Programming Techniques
November 2021

Università degli Studi di Firenze
Scuola di Ingegneria

Contents

1 Introduction

In this report, we present the development process of a Java SE application that interfaces with a database and interacts with the user through a graphical user interface (GUI). The development followed the Test Driven Development principles and was supported by the use of several tools and frameworks for the build automation, the quality of tests and application code, and the continuous integration.

As a case of study, we considered a simplified version of a gym administration system. Our system needs to keep track of gym members, courses offered by the gym, and member subscriptions to those courses. Furthermore, we wanted to develop two different versions of the application: a first version that interacts with a NoSQL database (namely, MongoDB) and a second version where the data are stored inside a relation (MySQL) database. The design and development process had to take into consideration code modularity and reuse across the two versions. Another important aspect to consider was the management of the transactions for the two types of databases to ensure data integrity when errors occur.

Sections 2 and 3 present the design of the application, the tools and frameworks that were used, and the implementation details of each part. Section 4 briefly describes the problems encountered during the development and how we dealt with them. Finally, in Section 5 we show how to build and use the application.

2 Design and Functionalities

During the design process, we followed the MVC pattern. We identified and separated parts of the projects with different responsibilities to ensure decoupling of components and facilitate code reuse across the two versions of the application.

Figure 1 contains the UML diagram of the final project. Some of the less important relationships (e.g. usage ad creation relationships) were omitted for the sake of readability, as well as class methods and attributes. Blue blocks correspond to part of the application (modules) with well-bounded responsibilities. Let us analyze them individually.

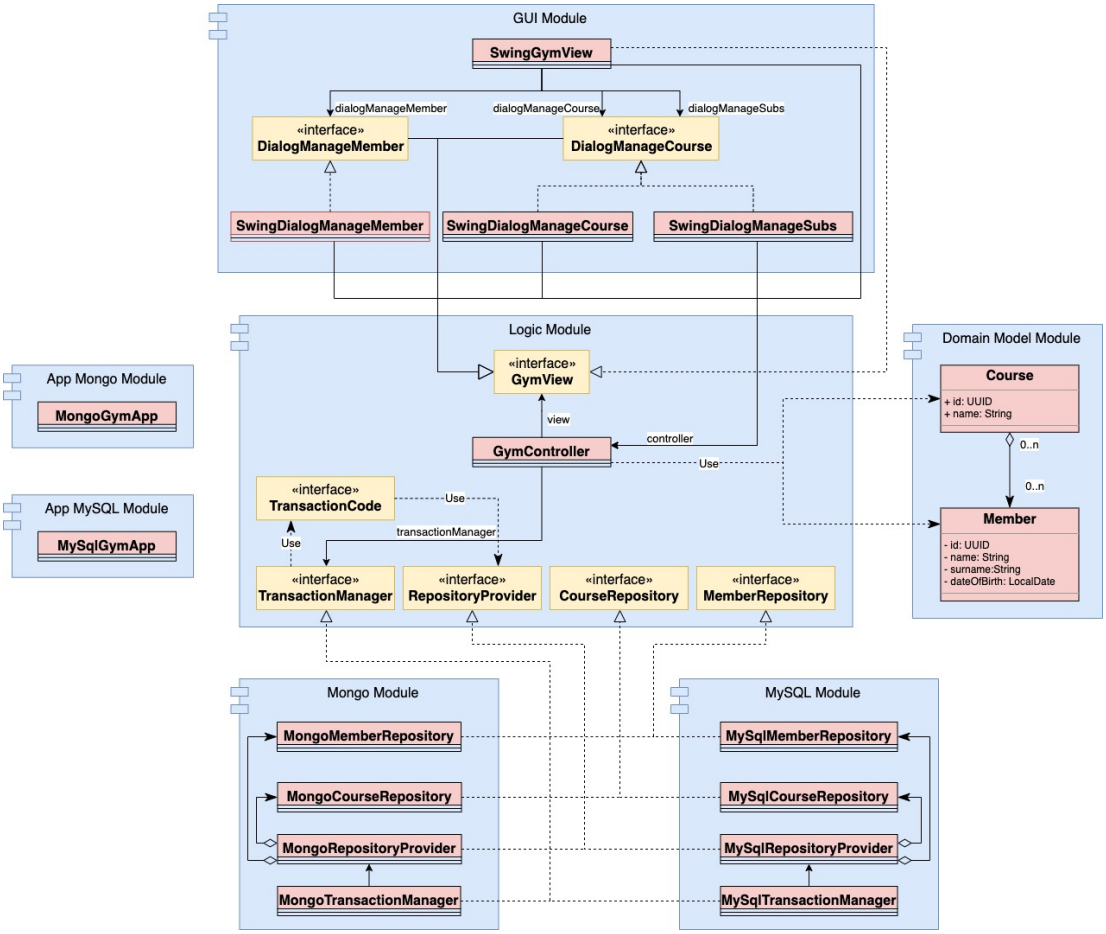

Figure 1: The UML diagram of the project. Blue boxes identifies part of the application with separate responsibilities. Minor relationships and class methods and attributes are omitted for the sake of clarity.

2.1 Domain Model Module

It contains the domain model of the application. Member and Course classes model the entities that should be mapped and stored in the database. To keep things simple, they define only a minimal set of personal data.

The two entities are bounded by a many-to-many relationship (the same course can have multiple subscribed members, and the same member can subscribe to multiple courses). We decided to handle it as a monodirectional relationship (the courses keep a list of the subscribed members). The reason is that, since we did not use an ORM framework, manually modeling many-to-many bidirectional relationships for different types of databases can be difficult, even in a simple scenario like ours.

2.2 Logic Module

It defines the GymController class containing the business logic of the application. It depends on the Model Module since the Controller uses and manipulates the domain model classes. The module also defines a set of interfaces that allows the Controller to communicate with the database and with a user interface transparently, decoupling the three components. This way, you can use the same logic with different databases and different UIs.

Let us briefly describe the responsibilities of these interfaces.
The Repository interfaces define methods to create, retrieve, update and delete entities on the database. TransactionCode is a functional interface that allows defining a sequence of operations on the database. This sequence will be handled as an atomic operation by the TransactionManager: it is devised for executing the TransactionCode inside a database transaction. RepositoryManager stores the Repository objects and makes them available when defining the code inside the TransactionCode. Finally, the View interface defines the methods to notify and provide data to the UI. Hence, a method of the Controller defines a sequence of database operations inside an implementation of the TransactionCode. The database operations take place through calls to Repository methods, provided by the RepositoryManager. Then, the Controller asks the TransactionManager to execute the TransactionCode atomically. Finally, with the results, it notifies the View.

2.3 Mongo Module and MySQL Module

These modules are the data layers of the application, that is, the part of the application that interacts with the database. They implement the TransactionManager, RepositoryProvider, and Repository interfaces for the respective database types.

2.4 GUI Module

It contains the classes that compose the GUI of the application. They are implemented using the Java Swing library. All of them have a reference to the

GymController to translate the user inputs into operations, and they implement the GymView to receive data and notifications from the controller.

The SwingGymView is the main window of the application and provides the user with the lists of the members and courses currently stored inside the database. It also contains all the necessary buttons to insert, update and delete members and courses and to manage subscriptions. Figure 2 shows the two screens of the SwingGymView. Insert and update operations happen through the SwingDialogManageMembers, SwingDialogManageCourses, and SwingDialogManageSubs, three dialogs whose role should be straightforward from their name.

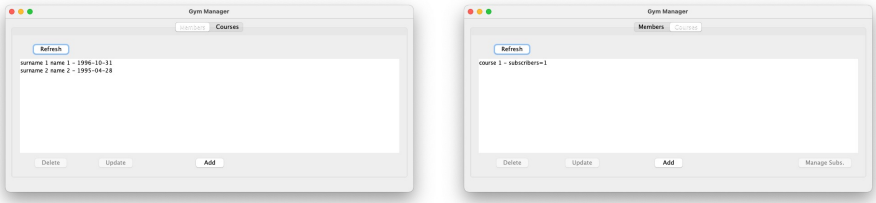

Figure 2: Screens of the main window of the GUI application

2.5 MongoGymApp and MySqlGymApp

They are the entry points of the two versions of the application. They only have the main method, in which they establish the connection to the competence database, instantiate the appropriate repositories, views, controllers, and other required classes, and manage their dependencies.

3 Implementation

For the implementation, we used **Maven** for build automation and dependency management. As anticipated in the introductory section, the classes shown in the previous section were implemented following the TDD principles. We developed single components in isolation through unit tests using **AssertJ** and **Mockito** frameworks. Also the GUI classes were developed using unit tests, exploiting the functionalities offered by **AssertJ Swing** for programmatically creating and manipulating Swing windows and components. Then implemented components were wired together and tested through integration tests. Finally, the two versions of the application were tested with end-to-end tests.

We used **JaCoCo** to calculate the percentage of code covered by tests and **PIT Mutation Testing** to validate test code against possible errors in the application code. We also used **SonarQube** to check the code quality of the project. All these tools are available as Maven plugins.

We used **Git** as our VCS, **GitHub** as the repository host, and **GitHub Actions** as the Continuous Integration provider. We configured GitHub Actions to connect to **Coveralls** and **SonarCloud**, two cloud services that host the code coverage and the code quality data of a project starting from the analysis results of JaCoCo and SonarQube.

Concerning the databases, we used **Docker** to set up and start database servers inside virtualized containers. Furthermore, the Docker plugin for Maven allowed us to manage database containers during the project build.

During the implementation, we exploited the build modularity features offered by the Maven modules. Each of the project modules presented in the previous section (the blue boxes in the UML diagram) corresponds to a Maven module in the final implementation. This way, we could implement separate aspects of the applications in separate projects and reuse common modules across different application versions. Moreover, with the modular build mechanism provided by Maven aggregators building a single version of the application is very easy: it is sufficient to include in the build process only the modules required by that specific version, without the need for building the entire project. The project is composed of **12 modules**:

- domain-model
- parent
- logic
- mongodb
- mysql
- gui
- app-mongo
- app-mysql

- report
- aggregator
- aggregator-mongo
- aggregator-mysql

In the following sections, we will analyze each of them, focusing on their contents and configurations. While reading, keep in mind that PIT, JaCoCo, Coveralls, and SonarQube Maven plugins are not activated by default in any of the modules: they are activated only through specific profiles.

3.1 Module domain-model

It corresponds to the Domain Model Module in the UML diagram.

Domain model classes Member and Course are simple data containers without logic: they only have getters, setters, toString, equals, and hashCode methods. The many-to-many relationship between Course and Member is implemented as a Set of Members inside the Course class: this way we ensure that a Member cannot be subscribed multiple times to the same Course. Since the classes have no logic, there are no tests inside the module.

The module pom has no parent and no particular configurations.

3.2 Module parent

Inside its pom file, it defines the common dependencies, properties, plugins, and configurations common to the other modules. Since it does not have Java code, its packaging type is set to pom.

Let us get deep into its pom file.

The dependencies are specified in the `<dependencyManagement>` section: if a module requires one of them has to include it inside its own `<dependency>` section.

The `<pluginManagement>` locks the version of the default Maven plugin to enforce reproducibility of the build and specifies common configurations for the JaCoCo, Coveralls, PIT, and Docker plugins.

The JaCoCo plugin is configured to keep track of covered lines (goal `prepare-agent`), generate the report files (goal `report`), and exclude classes without logic from the coverage.

```
1  <plugin>
2     <groupId>org.jacoco</groupId>
3     <artifactId>jacoco-maven-plugin</artifactId>
4     <version>0.8.7</version>
5     <executions>
6        <execution>
7           <goals>
8              <goal>prepare-agent</goal>
```

```
            <goal>report</goal>
         </goals>
      </execution>
   </executions>

   <!-- Exclude class without logic from the coverage -->
   <configuration>
      <excludes>
         <exclude>**/TransactionException.*</exclude>
         <exclude>**/MongoRepositoryProvider.*</exclude>
         <exclude>**/gui/**/SimpleDocumentListener.*</exclude>
         <exclude>**/app-mongo/**/MongoGymApp.*</exclude>
         <exclude>**/mysql/**/MySqlRepositoryProvider.*</exclude>
         <exclude>**/app-mysql/**/MySqlGymApp.*</exclude>
      </excludes>
   </configuration>
</plugin>
```

The PIT plugin only enables stronger mutators, letting the child modules bound the plugin goals to the Maven phases.

```
<plugin>
   <groupId>org.pitest</groupId>
   <artifactId>pitest-maven</artifactId>
   <version>1.7.2</version>
   <configuration>
      <mutators>
         <mutator>STRONGER</mutator>
      </mutators>
   </configuration>
</plugin>
```

Concerning the Docker plugin, we bound the start and stop goals to the `pre` and `post-integration-test` phases. The child modules will specify which image to use in the started container and its configuration.

```
<plugin>
   <groupId>io.fabric8</groupId>
   <artifactId>docker-maven-plugin</artifactId>
   <version>0.34.1</version>
   <executions>
      <execution>
         <id>docker-start</id>
         <phase>pre-integration-test</phase>
         <goals>
            <goal>start</goal>
         </goals>
      </execution>
      <execution>
         <id>docker-stop</id>
```

```
15          <phase>post-integration-test</phase>
16          <goals>
17             <goal>stop</goal>
18          </goals>
19       </execution>
20    </executions>
21 </plugin>
```

Finally, the pom file defines **four profiles**:

- `jacoco`: used for running the code coverage locally
- `coveralls`: used for running the code coverage and sending the results to Coveralls
- `sonar`: used for running code quality analysis
- `pit`: used for mutation testing

The first three enable the JaCoCo plugin since it is required when running the relative tasks (SonarQube also keeps track of code coverage thanks to the JaCoCo analysis). The pit profile enables the PIT plugin.

3.3 Module logic

It corresponds to the Logic Module in the UML diagram and it is a child of the parent module.

It contains **22 unit tests** used for implementing the GymController class.

Inside the pom file, there are specified additional configurations for the PIT plugin. Its `mutationCoverage` goal is bound to the test phase and the `mutationThreshold` is set to 100% (we want the build to fail if there are survived mutants).

3.4 Module mongodb

It corresponds to the Mongo Module in the UML diagram and it is a child of the parent module.

Its classes require a connection to a MongoDB database in order to function and be tested. Following the TDD, we could have used an in-memory database to write unit tests and implement the classes, and then write integration tests with a real database. However, we decided to directly write **integration tests to implement the components**, considering them as unit tests. Actually, they still are integration tests and they have to be executed in the proper Maven phase to correctly start and shut down the real database server, but they have to be considered as unit tests due to their number and granularity. They are **30** in total and they test MongoMemberRepository, MongoCourseRepository, and MongoTransactionManager. MongoRepositoryProvider is a wrapper for the two repositories classes without logic, hence it is excluded from testing.

It is worth analyzing the pom of this Maven module. Since we used integration tests as unit tests we had to properly configure some plugins, as shown in the following snippet.

```
<build>
   <pluginManagement>
      <plugins>
         <!-- Bind the PIT goal to the integration-test phase since they
         ↪  have been used to implement the module classes -->
         <plugin>
            <groupId>org.pitest</groupId>
            <artifactId>pitest-maven</artifactId>
            <executions>
               <execution>
                  <id>default-mutation</id>
                  <phase>integration-test</phase>
                  <goals>
                     <goal>mutationCoverage</goal>
                  </goals>
               </execution>
            </executions>
            <configuration>
               <!-- Exclude MongoRepositoryProvider (it contains no logic
               ↪  and there are no tests for it) -->
               <excludedClasses>
                  <param>it.jasonravagli.gym.mongodb.
                  ↪  MongoRepositoryProvider</param>
               </excludedClasses>
               <!-- Disable the threshold since the module contains only
               ↪  integration tests and database would not be shut down
               ↪  in case of failures -->
               <mutationThreshold>0</mutationThreshold>
            </configuration>
         </plugin>
         <!-- Configure JaCoCo to consider lines covered by integration
         ↪  tests -->
         <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <executions>
               <execution>
                  <id>prepare-it</id>
                  <goals>
                     <goal>prepare-agent-integration</goal>
                  </goals>
               </execution>
               <execution>
                  <id>report-it</id>
                  <goals>
                     <goal>report-integration</goal>
```

```
                </goals>
            </execution>
        </executions>
    </plugin>
  </plugins>
</pluginManagement>
<plugins>
  <plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <configuration>
      <images>
        <image>
          <name>${docker.image.mongo}</name>
          <run>
            <ports>
              <port>27017:27017</port>
            </ports>
            <wait>
              <time>60000</time>
            </wait>
          </run>
        </image>
      </images>
    </configuration>
  </plugin>
</plugins>
</build>
```

First of all, the `mutationCoverage` goal of the PIT plugin is bound to the integration-test phase instead of the default test phase. Moreover, the `mutationThreshold` is set to 0 to disable it: if the `mutationCoverage` falls below the threshold the Maven build would fail immediately, the post-integration-test phase would not be executed and the database server container would not be stopped.

The JaCoCo plugin is configured to include in the coverage data and the reports the lines covered by our integration tests.

Finally, we tell the Docker plugin to use an image containing a MongoDB server, expose the default port 27017 to the host and wait 60 seconds for the container to be completely ready. Remember that the start and stop goals were already bound to the pre and post-integration-test phases inside the parent pom. The Docker image used is `candis/mongo-replica-set:0.0.2`, not the official Mongo image. The reason is that if we want to support transactions we have to set up a MongoDB replica set, which basically is a cluster of MongoDB servers with at least 3 nodes properly configured to take care of backups and data replication. Since the configuration of such a cluster was beyond the purposes of this work, we search on Docker Hub for an already-built image. According to the notes on the Docker Hub page, a container of this image take about 40

seconds to be completely up. It does not produce any logs to check for, so we decided to wait 60 seconds before connecting to it with our tests.

3.5 Module mysql

It corresponds to the MySQL Module in the UML diagram and it is a child of the parent module.

Considerations about the development of the components through integration tests and the configurations of the plugins are the same as for the mongodb module. Of course, instead of the MongoDB image, we configured the Docker plugin to use a MySQL image. We used the official `mysql/mysql-server:8.0.27` image, which starts a MySQL 8 server. Below is shown its configuration:

```
<plugin>
   <groupId>io.fabric8</groupId>
   <artifactId>docker-maven-plugin</artifactId>
   <configuration>
      <images>
         <image>
            <name>${docker.image.mysql}</name>
            <run>
               <ports>
                  <port>3306:3306</port>
               </ports>
               <env>
                  <MYSQL_ROOT_PASSWORD>password</MYSQL_ROOT_PASSWORD>
                  <MYSQL_ROOT_HOST>%</MYSQL_ROOT_HOST>
                  <MYSQL_DATABASE>test</MYSQL_DATABASE>
               </env>
               <wait>
                  <log>.*mysqld: ready for connections.*port:
                  ↪  3306.*</log>
                  <time>30000</time>
               </wait>
               <volumes>
                  <bind>
                     <volume>${path.mysql-scripts.src}:
                     ↪  /docker-entrypoint-initdb.d</volume>
                  </bind>
               </volumes>
            </run>
         </image>
      </images>
   </configuration>
</plugin>
```

The default MySQL port 3306 of the container is exposed to the host. We specified a set of environment variables (section `<env>`) to change the password of the root user (`MYSQL_ROOT_PASSWORD`), allow external connections to use that

user (`MYSQL_ROOT_HOST`), and create a new test database (`MYSQL_DATABASE`). The Docker image on startup will use these environment variables to configure the container.

In the `<volume>` section we tell Docker to copy the specified SQL script file (`$path.mysql-scripts.src`) into the container folder `docker-entrypoint-initdb.d`. This script contains the DDL statements to create the tables required by the application. On startup, the MySQL server container reads and executes scripts contained in that folder, and automatically generates the database schema.

Finally, in the `<wait>` section we pause the build until the MySQL server logs on the console that is ready to accept connections, with a timeout of 30 seconds.

3.6 Module gui

It corresponds to the GUI Module in the UML diagram and it is a child of the parent module.

The SwingGymView is a JFrame, while the other 3 classes are JDialog. For their implementation, we wrote **100 unit tests** using the AssertJ Swing library. We encountered some problems and compatibility issues with this library, described in Section 4.

After implementing the GUI classes in isolation, we wrote **11 integration tests** to test their functioning when wired together and with the GymController.

The PIT plugin is not activated for this module since PIT seems to not work properly with AssertJ Swing tests. Further details can be found in Section 4.

3.7 Module app-mongo

It corresponds to the App Mongo Module in the UML diagram and it is a child of the parent module.

It has a single class, MongoGymApp, containing the main method. Here we connect to a MongoDB database, we create all the required components (GUI, controller, database classes) from the other modules and we manually manage their dependencies.

Using the picocli library we provide a set of command-line arguments to specify the address and the port of the database server and the database name to connect to. The main method was not developed using unit tests (for this reason is excluded from the code coverage and mutation testing).

The module, since it uses all the other modules (except the mysql one, obviously) and it is the entry-point of the MongoDB version of the application, lends itself to hosting integration tests and end-to-end tests. We wrote **10 integration tests** to test the correct collaboration of GUI, controller, and data layer classes, that is the correct functioning of the MVC components. Then we implemented **9 end-to-end tests** to verify the correct behavior of the application from the user point of view.

Inside the pom, the Docker plugin is configured to start the MongoDB image as shown for the mongodb module. We also use the maven-assembly-plugin to generate an executable Fat JAR during the package phase:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-assembly-plugin</artifactId>
   <executions>
      <execution>
         <phase>package</phase>
         <goals>
            <goal>single</goal>
         </goals>
         <configuration>
            <descriptorRefs>
               <descriptorRef>jar-with-dependencies</descriptorRef>
            </descriptorRefs>
            <archive>
               <manifest>
                  <mainClass>it.jasonravagli.gym.appmongo.
                  ↪  MongoGymApp</mainClass>
               </manifest>
            </archive>
         </configuration>
      </execution>
   </executions>
</plugin>
```

3.8 Module app-mysql

It corresponds to the App MySQL Module in the UML diagram and it is a child of the parent module.

The same considerations made for the app-mongo module apply. It contains **10 integration tests** and **9 end-to-end tests**. As for the app-mongo module, inside the pom, we configured the Docker plugin to start a MySQL container and the maven-assembly-plugin to generate the Fat JAR.

3.9 Module report

This module is a child of the parent module and produces aggregated code coverage reports of the whole project.

For this purpose, we configured the JaCoCo and Coveralls inside two profiles already defined in the parent pom: `jacoco` and `coveralls`. `jacoco` is intended to be used to produce a local aggregated report, while `coveralls` should be used during the build on the CI server to generate and send the report to the Coveralls platform.

```
<profiles>
   <profile>
```

```
3         <id>jacoco</id>
4         <build>
5            <plugins>
6               <plugin>
7                  <groupId>org.jacoco</groupId>
8                  <artifactId>jacoco-maven-plugin</artifactId>
9                  <executions>
10                    <execution>
11                       <phase>verify</phase>
12                       <goals>
13                          <goal>report-aggregate</goal>
14                       </goals>
15                    </execution>
16                 </executions>
17              </plugin>
18           </plugins>
19        </build>
20     </profile>
21     <profile>
22        <id>coveralls</id>
23        <build>
24           <plugins>
25              <plugin>
26                 <!-- JaCoCo report is required by coveralls-maven-plugin
                   ↪  -->
27                 <groupId>org.jacoco</groupId>
28                 <artifactId>jacoco-maven-plugin</artifactId>
29                 <executions>
30                    <execution>
31                       <phase>verify</phase>
32                       <goals>
33                          <goal>report-aggregate</goal>
34                       </goals>
35                    </execution>
36                 </executions>
37              </plugin>
38              <plugin>
39                 <groupId>org.eluder.coveralls</groupId>
40                 <artifactId>coveralls-maven-plugin</artifactId>
41                 <executions>
42                    <execution>
43                       <phase>verify</phase>
44                       <goals>
45                          <goal>report</goal>
46                       </goals>
47                    </execution>
48                 </executions>
49                 <!-- JaCoCo reports are generated from report-aggregate,
                   ↪  hence they are not in the default directory -->
50                 <configuration>
```

```
                    <jacocoReports>
                        <jacocoReport>${project.reporting.outputDirectory}/
                        ↪ jacoco-aggregate/jacoco.xml</jacocoReport>
                    </jacocoReports>
                    <!-- Source files of the modules are not automatically
                    ↪ detected -->
                    <sourceDirectories>
                        <sourceDirectory>${project.basedir}/..
                        ↪ /logic/src/main/java</sourceDirectory>
                        <sourceDirectory>${project.basedir}/..
                        ↪ /mongodb/src/main/java</sourceDirectory>
                        <sourceDirectory>${project.basedir}/..
                        ↪ /gui/src/main/java</sourceDirectory>
                        <sourceDirectory>${project.basedir}/..
                        ↪ /mysql/src/main/java</sourceDirectory>
                    </sourceDirectories>
                </configuration>
            </plugin>
        </plugins>
    </build>
</profile>
</profiles>
```

We bound the `report-aggregate` goal to the verify phase (that came after the test and integration-test phase where code coverage data are generated). This goal collects the JaCoCo execution data from the modules specified in the `<dependencies>` section to create the aggregated report files. In that section we specified only the modules involved in the code coverage, that is logic, mongodb, mysql, and gui.

Concerning the Coveralls plugin, we bound its `report` goal to the verify phase. The goal reads the XML aggregated report produced by JaCoCo (and specified in the `<jacocoReport>` section) and generates the Coveralls report that will be sent to the remote service platform. Information about Coveralls server address and security tokens are specified inside the GitHub actions workflows (see next sections). To conclude, Coveralls can automatically detect only source folders of the current module and those of modules that are subfolders of the current one. For this reason, we had to specify the path of the source folders of each module involved in the code coverage report.

3.10 Module aggregator

It is the module that builds the entire project. Its pom file does not inherit from any other pom.

Since it does not produce any artifact, we specified `pom` as the packaging type. We specified all the other modules except the report one in the `<modules>` section of the pom.

```
<modules>
    <module>../parent</module>
```

```
    <module>../domain-model</module>
    <module>../logic</module>
    <module>../mongodb</module>
    <module>../gui</module>
    <module>../app-mongo</module>
    <module>../mysql</module>
    <module>../app-mysql</module>
</modules>
```

This way, running a Maven build from the aggregator module will build and test the whole project and produce the Fat JARs of the two versions of our application.

The pom also specifies **three profiles**: `jacoco` and `coveralls`, already explained in the previous modules, and `sonar`.

```
<profiles>
   <profile>
      <id>jacoco</id>
      <modules>
         <module>../report</module>
      </modules>
   </profile>
   <profile>
      <id>coveralls</id>
      <modules>
         <module>../report</module>
      </modules>
   </profile>
   <profile>
      <id>sonar</id>
      <build>
         <plugins>
            <plugin>
               <groupId>org.sonarsource.scanner.maven</groupId>
               <artifactId>sonar-maven-plugin</artifactId>
               <executions>
                  <execution>
                     <phase>verify</phase>
                     <goals>
                        <goal>sonar</goal>
                     </goals>
                  </execution>
               </executions>
            </plugin>
         </plugins>
      </build>
   </profile>
</profiles>
```

When the first two are activated, the report module will be included inside the build, producing the aggregated JaCoCo and Coveralls reports. The sonar

profile, instead, activates the SonarQube Maven plugin that will analyze the code and send code quality reports to the SonarCloud server. As for Coveralls, the server address and security tokens are not configured in the pom file but in the GitHub Actions workflows. The only configurations specified for the SonarQube plugin are the classes to exclude from the code coverage (the same as JaCoCo, but we need to repeat them since SonarQube does not consider JaCoCo exclusions) and the code quality rules to ignore. This is done in the `<properties>` section. Details of rule exclusions and their explanation can be found directly in the pom file.

3.11 Modules aggregator-mongo and aggregator-mysql

They are aggregator projects to build a specific version of the application. Inside the `<modules>` section, they list only the modules required by the competence version. For example, for the aggregator-mongo we have:

```
<modules>
   <module>../parent</module>
   <module>../domain-model</module>
   <module>../logic</module>
   <module>../mongodb</module>
   <module>../gui</module>
   <module>../app-mongo</module>
</modules>
```

3.12 Continuous Integration with GitHub Actions

We created two workflow files for GitHub Actions. In the first one, we build the project on **Ubuntu 20.04 with JDK 11**. This is our reference environment, hence this workflow is activated on every push and pull request. The executed build command is the following:

```
xvfb-run mvn install -Pcoveralls,pit,sonar \
    -DrepoToken=$COVERALLS_TOKEN -DpullRequest=${{
    ↪  github.event.pull_request.number }} \
    -Dsonar.organization=jasonravagli-github
    ↪  -Dsonar.host.url=https://sonarcloud.io \
    -Dsonar.projectKey=jasonravagli_gym-manager
```

Since we have GUI tests, we need to execute the build within a graphical environment (through `xvfb-run`). We activate the coveralls, pit, and sonar plugins to perform mutation testing and generate code coverage and code quality reports. These reports are sent to Coveralls and SonarCloud servers. Security tokens to access the project profile on the two platforms are stored inside GitHub and passed to the build as environment variables:

```
env:
    COVERALLS_TOKEN: ${{ secrets.COVERALLS_TOKEN }}
```

```
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

After the build, we generate test reports and store the build artifacts (code coverage reports, mutation testing reports, test reports, JARs) in a zip folder that can be downloaded and inspected:

```
- name: Generate JUnit Report
  run: >
    mvn surefire-report:report-only surefire-report:failsafe-report-only
    site:site -DgenerateReports=false
  working-directory: ${{ env.workdir }}
  if: ${{ always() }}
- name: Archive Generated Reports
  uses: actions/upload-artifact@v2
  if: ${{ always() }}
  with:
    name: reports
    path: |
      **/target/site
      **/target/pit-reports
```

The second workflow file performs the build on **Ubuntu 20.04 with JDK 8** to ensure compatibility with the older version of Java. The command is the following:

```
xvfb-run mvn install
```

It does not make sense to repeat code coverage and code quality analysis also in this environment. The workflow is activated only on pull requests and when a push is performed on the master branch. The reason why we did not include macOS and Windows were not included in the GitHub Actions workflows is explained in the next section.

4 Problems Encountered

4.1 AssertJ Swing Errors and Compatibility Issues

During the development, we found some issues concerning the AssertJ Swing testing library. We developed the project with Eclipse and JDK 11 on macOS 11. Despite we gave to Eclipse and Java the proper permissions to control the computer (moving the mouse pointer, clicking, using the keyboard), the AssertJ Swing robot could not perform any input operations. This resulted in a systematic failure of GUI tests when launched from Eclipse. The only way to get around the problem was to perform Maven builds from the shell.

We could not perform mutation testing on GUI test classes. When using PIT with them, the build failed with the error "PIT requires a green suite". Apparently, during the preliminary test executions performed by PIT before mutating the code, some tests failed even if Maven surefire plugin could execute them correctly. We could not solve the problem and we decided to not activate the PIT plugin for the modules that use AssertJ Swing. Moreover, mutation testing on GUI classes, with a lot of autogenerated code concerning the UI, may not be appropriate.

AssertJ Swing gave problems also on the GitHub Actions builds. In Ubuntu environments, some tests encountered random failures due to ComponentLookup exceptions, that is when the library cannot find in the window the components to interact with. The library sometimes connected to windows with empty hierarchies (i.e. without components). This is a known and still open issue of AssertJ Swing (`https://github.com/assertj/assertj-swing/issues/157`). We found that the problem always presents itself when using classes that extend JDialog. We could not find any solution, and we decided to implement a JUnit rule that retries to execute the test in case of failures until a maximum number of attempts is reached. This way, we could get around the problem by exploiting its non-deterministic nature. The JUnit rule is activated only in Ubuntu environments.

4.2 Identity of Database Entities

To keep a single domain model module for both versions of the application, we had to use an identity management mechanism for database entities compatible with MongoDB and MySQL. MySQL provides autoincrement mechanisms for integer ids, but MongoDB, given its distributed nature, does not. MongoDB instead provides autogenerated UUIDs.

Application side, we decided to model the ID field of domain model classes as a UUID. Then we gave our application the responsibility to generate and assign UUIDs to new entities using the proper Java library. Database side, the ID field is treated as a binary field in MySQL and as a UUID field in MongoDB (the latter natively supports UUIDs).

4.3 Compatibility Issues with Docker Images

As we already described in the previous section, MongoDB needs a replica set to support transactions. For this reason, we choose to use a Docker image that provides a ready-to-use MongoDB cluster with a replica set. Unfortunately, the image is not supported on Windows and we had to exclude the OS from the GitHub Actions workflows.

The MySQL Docker image instead gave errors on the macOS environments provided by GitHub Actions. For some reason, the containers shut down immediately after the startup. The MySQL image worked fine on our macOS development environment, hence we concluded that the problem was with the GitHub Actions virtual environments. As for Windows, we excluded macOS from the CI build.

5 Build Replication and Usage

The project can be tested and built locally by running the following Maven commands from the `aggregator` directory:

```
mvn clean verify -Pjacoco,pit
```

`jacoco` and `pit` profiles are optional. When activated will generate the code coverage and mutation testing reports inside each module folder. With the `jacoco` profile activated an aggregated report about code coverage will be generated inside the report module.

The above command will also generate the executable Fat JARs for the two versions of the application, respectively inside the `app-mysql/target` and `app-mongo/target` folders. In the following subsections, we describe how to run these JARs.

5.1 MongoDB Version

The JAR file for the version that connects to a MongoDB database is called `app-mongo-1.0-jar-with-dependencies.jar` and can be found inside the `app-mongo/target` folder.

For a quick demo execution you can start an already-setup MongoDB Docker container by running the following command inside the `app-mongo` folder:

```
mvn docker:start
```

Then if you simply run the JAR file the application will start and connect to the MongoDB container.

If you already have your MongoDB database you can tell the application to connect to it by specifying additional arguments from the command line. The table below shows the **available arguments** (a description is provided also from the command line specifying `--help` at the moment of execution).

Argument	**Description**
`--mongo-host`	MongoDB server address
`--mongo-port`	MongoDB server port
`--db-name`	Name of the database to connect to

Keep in mind that the application uses Mongo transactions. When using custom databases, they must be configured with a Mongo cluster with replicas to support transactions. Otherwise, the application will stop with errors.

5.2 MySQL Version

The JAR file for the version that connects to a MySQL database is called `app-mysql-1.0-jar-with-dependencies.jar` and can be found inside the `app-mysql/target` folder.

For a quick demo execution you can start and set up the MySQL Docker container by running the following command inside the `app-mysql` folder:

```
mvn docker:start
```

Then if you simply run the jar the application will start and connect to the MySQL container.

If you already have your MySQL database you can tell the application to connect to it by specifying additional arguments from the command line. The table below shows the **available arguments** (a description is provided also from the command line specifying `--help` at the moment of execution).

Argument	**Description**
`--mysql-host`	MySQL server address
`--mysql-port`	MySQL server port
`--mysql-user`	Username used to access to the database
`--mysql-pwd`	Boolean flag. When true the user will be asked to input the db user password on startup, otherwise the default password *'password'* will be used (intended only for testing and demo purposes)
`--db-name`	Name of the database to connect to

The database to connect to must already have the schema required by the application. This schema can be found inside the file `docker/mysql-scripts/mysql-db-schema.sql`.