Advanced Techniques and Tools for Software Development
Final Report - Museum Manager Application

Lorenzo Pisaneschi*,
Master's Degree in Computer Engineering, University of Studies of Florence, Florence, Italy
Email: *lorenzo.pisaneschi1@stud.unifi.it,

***Abstract*—In this report I present the project I developed for the "Advanced Techniques and Tools for Software Development" course at University of Florence, Italy: Museum Manager App, a booking management system desktop application designed for museums. However, the aim of this project is not to build up a complex application, but rather to exploit and show the powerful of frameworks which came in help in coding and testing. In fact, on one hand test techniques are the core of the work here presented (Unit testing, Integration testing and End to End testing); on the other, the loop is closed by tools which came in aid for maintain the code clean and net, modular and maintainable. Therefore, in this work the mainstream testing techniques, such as TDD, are shown, concepts like mocking, containerization, code quality and continuous integration are explained and tests against a real SQL database are addressed; all these aspects are treated with a single goal: demonstrate that it is possible to develop applications taking advantages from processes automation and advanced testing techniques.**

***Index Terms*—CI, Docker, Eclipse, Git, Java, Maven, Mocking, Modularity, SQL, Testing**

I. Introduction

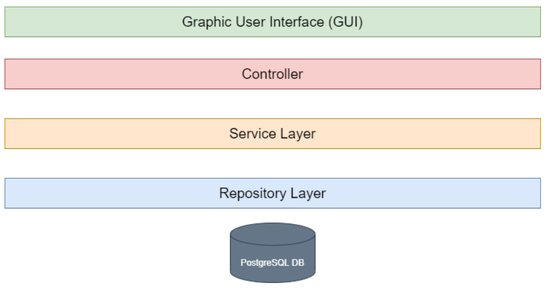


Figure 1. Application architecture overview.

Museum Manager App has been developed using Java 11 as programming language [1] and its main structure is shown in fig. 1. This architecture has been built splitting up the main project in different Maven [2] [3] sub-modules. This strategy makes the whole project modular, increasing its maintainability.

- **museum-manager-bom**: This module is the so called Bill of Material one: it collects all the Maven dependencies which will be used in the other modules. The BOM ensures that dependencies versions are in sync in the whole project.
- **museum-manager-parent**: The Parent POM describes the shared configurations among all the project parts: each sub-module inherits from it which, in turn, inherits from the BOM in order to make available all the synchronized dependencies.
- **museum-manager-repository**: The repository module implements the repository layer. Its responsibility is to communicate with the database, for basic CRUD operations. It incorporates a Transaction Manager system and the persistency level is structured by Hibernate [4] and JPA [5], as will be explained in detail in the follow.
- **museum-manager-core**: As the name says, this is the core module. In fact, it contains the service layer implementation, which is a mid-layer that allows the interconnection between the database and the controller, then the view. It also exposes controller and views interfaces.
- **museum-manager-presentation**: The presentation module implements views and controller interfaces belonging to the core module.
- **museum-manager-app**: In this module all is wired: in fact, it contains only the main class and configurations used to start the entire application.
- **museum-manager-report**: This project contains only its POM and its aim is simply to generate the JaCoCo report about Code Coverage (if we are interested) [6] [7]. Here repository, core and presentation modules are specified in order to obtain code coverage report about them.
- **museum-manager-aggregator**: This project is completely separated from the others and it is used simply to build all the modules with Maven or a Maven Wrapper [8]. It contains a jacoco Maven profile which, if activated, will add also the museum-manager-report module to the build, aggregating the generated jacoco reports.

The correct order chosen to build all the project modules is inferred by *the Maven Reactor* system, which is able to scan all the dependencies among sub-projects and decide how to connect all together. In this project also Maven profiles are used. When activated, Maven profiles allows to use specific plugins during the build: they are an option when we want to use some tools rather than other in determined situations. For this reason, in the Parent POM the following profiles are specified: the **pit** profile, which activates the PIT Mutation

Testing framework, the **jacoco** profile, explained above and included also in the museum-manager-aggregator module and the **javadoc** profile, which, if activated, generates the javadoc sources. Moreover, a **build-app** profile is also added to the museum-manager-app module and it is used to build the Docker image for the Museum Manager App during the package phase. Now that the entire architecture is summarized, it is possible to go deeper in project and used developing tools analysis.

II. Implementation

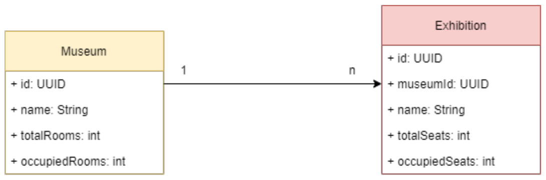


Figure 2. Museum and Exhibition relationship.

Implementation starts from the entities model, database configuration and mapping. For these purposes, Museum and Exhibition entities have been defined, considering that one Museum has many rooms, some available and some occupied by Exhibitions, where each Exhibition has a fixed number of seats, which can be booked. Naturally, a relation exists between Museum and Exhibition: a Museum can host various exhibition and this results in a "One to Many" relationship between the over mentioned entities, as shown in fig. 2. Regarding the database, the Museum Manager application is equipped with a PostgreSQL database on which entities are mapped by Hibernate ORM (Object Relation Mapping). Hibernate implements the JPA (Java Persistence API) and then it can be used in a natural way for entities manipulation. However, the relation between Museum and Exhibition is not implemented using Hibernate "@OneToMany" and "@ManyToOne" decorators: this is due to the willingness to have more control over entities and their manipulation. Finally, it is interesting to mention the Hikari use: Hikari is a very lightweight connection pool framework, which allows to consistently manage multiple requests for database connections [9].

A. Repository layer

Objects persistency is a Repository layer responsibility: all database communications methods are implemented there. It is important to stress that CRUD operations are defined in specific interfaces, in such a way that specific repositories (ie. classes that work with different databases) can be implemented referencing them. For this project two repositories interfaces (one for Museum and one for Exhibition entities) are defined and since PostgreSQL and Hibernate have been chosen, database configurations are set in a special file, the **persistence.xml**. In the persistence.xml a persistence unit is defined with all its properties, such as jdbc url, database user and password and Hibernate settings. The defined persistence unit will be handled by the JPA EntityManagerFactory instantiating an EntityManager, which really manages entities and communicates with the database. Regarding museum-manager-repository module persistence.xml file, it contains configuration to be used only during tests, since database to be used in the final application will be configured in the museum-manager-app module directly by the user.

1) Testing repositories: The above preface is fundamental, as it is fundamental to inspect how to unit test a repository. First of all a real database is needed and it is needed to start always from a clean situation, before each test, to avoid false negative or false positives results from unit tests. For this project Testcontainers library was the solution [10]. Testcontainer provides the possibility to create a real database (then a real PostgreSQL) inside a Docker container which will be active only for the duration of the tests. Before testing, it is also possible to run basic SQL script inside such a container, in order to start with desired situation (an empty database with all its schema). Then, before each single test, a clean routine is called to empty the database. This mechanism ensures to test the SUT (the repository) in isolation as much as possible, using an essential dependency (the database) getting closer to a unit test shape.

B. Transaction Management

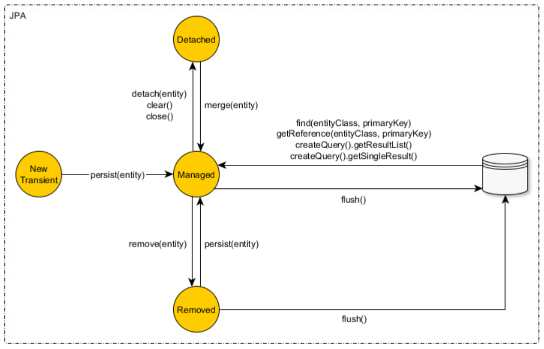


Figure 3. Entities states in a JPA persistence context.

Methods implemented in repositories have to essentially execute queries. During read operation (like *findAllExhibitions(), findallMuseums(), findMuseumByName(), findExhibitionById()* and similar) database state is not changed; things get more complicated when queries provide operations which change the database, mostly in concurrency situations, when more than one connection requests are send to database. The key concept is that the database has to be always consistent. For this aim transaction mechanism exists. A **transaction** is a unit of work, in which a sequence of operations is defined: if the transaction succeeds without anomalies, then its operations are made effective by committing them and changing the database state; otherwise, everything is canceled by a rollback. In other words, changes to database are effective only after the commit statement. Transactions expected properties can

be summarized in the **ACID** acronym: Atomicity, Consistency, Integrity, Durability.
In repositories tests, transactions are handled using directly the Entity Manager: in fact, transaction management is not a repository responsibility. To better handle transactions another component was defined, the **Transaction Manager**. Since transactions are managed differently by different databases, a Transaction Manager interface have been written as reference for classes responsible of this task dealing with different databases. Again, in this project a Transaction Manager class for PostgreSQL database has been implemented. *PostgresTransactionManager* works with the entity manager and the repositories defined for Museum and Exhibition entities and its responsibility is to mark a transaction as to be committed or to be aborted. To do this, the core idea is to expose a *doInTransaction* method which accepts as argument a lambda function, a Java Functional Interface, which contains the query code to be executed inside transaction, dealing with a *MuseumRepository*, an *ExhibitionRepository* or both. If an error occurs during a database operation an exception will be thrown from repository and catch by the transaction manager which will rollback the transaction; otherwise, if the transaction code returns successfully, the transaction manager will commit the updates to database. In this project context, the entity manager instance for repositories and transaction manager has to be the same.

1) Testing transaction management: As for repositories tests, *PostgresTransactionManager* unit tests are not real unit tests: again, a real PostgreSQL database is used as external dependency; moreover, the Transaction Manager uses also Museum and Exhibition repositories instances, adding other two dependencies. On one hand, repositories and database can't be mocked: they deal with third party libraries (for example, the Java persistency); on the other hand, the repositories have been fully tested before the *PostgresTransactionManager* testing and implementation. Finally, the use of a real PostgreSQL database make us confident for when services will interact with the tested repository layer.

C. Service Layer

Once repository module is fully tested and working, we can put our attention on **museum-manager-core module**. Controller, views and service interfaces belong to this module; in particular, in the core module service implementation is also provided, while views and the controller are implemented in the presentation module. Basically, the service wraps a *Transaction Manager* instance in order to execute CRUD database operations. The business logic is very simple and handles the relationship between the Exhibition and Museum entities: so, when a new exhibition is created or deleted for a Museum, its available rooms number is updated accordingly; the same happens for the available exhibition seats when an the exhibition is booked. Consequently, the *Transaction Manager* will use a *MuseumRepository*, an ExhibitionRepository or both depending on the kind of operation. It is important to underline that the Transaction Manager is able to accept any kind of the mentioned repositories implementations, depending on the application database. Regarding exceptions handling, database errors are managed in the Repository Layer, which throws a *RepositoryException*, catch in turn by the Service Layer which will communicate the error to the controller (and therefore to the user, updating the view with an error message) using a defined *MuseumManagerServiceException*, specific for this layer.

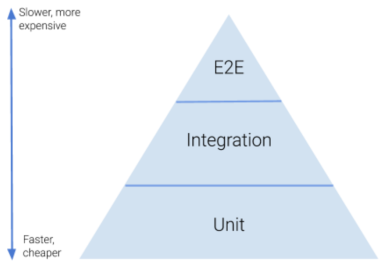
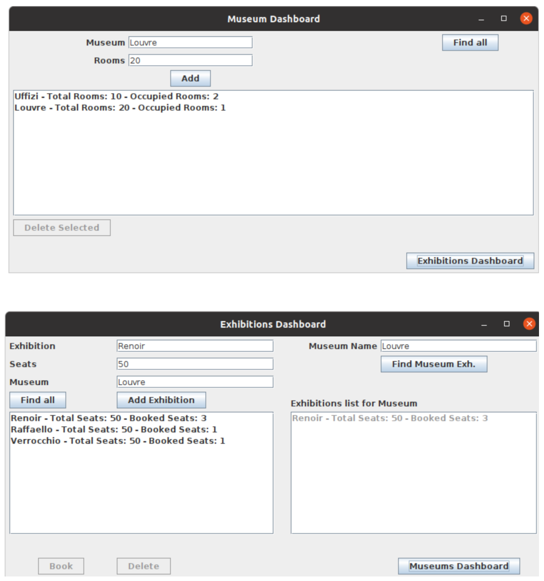


Figure 4. Test pyramid.

1) Testing service layer: As mentioned above, the *MuseumManagerService* uses a Transaction Manager dependency, which works with the repositories communicating with the PostgreSQL database. In this scenario, we must avoid direct database interaction for unit tests: the aim is to test our service component in isolation. To do that, we can deal with test doubles, that is, fake dependencies which simulate their expected behaviour. Mocks are an example of test doubles: in this project, *MuseumManagerService* is tested mocking its external *TransactionManager* dependency, the *MuseumRepository* and the *ExhibitionRepository* used to instantiate the *TransactionManager* itself; then in tests the mentioned components methods are stubbed to effectively use mocked functionalities. Such tasks are exploited in this project thanks to Mockito framework [11] [12], which enables mocking and stubbing operations; moreover, Mockito allows to "spy" entities instances (in our case, Museum and Exhibition) to verify the performed updates directly on their properties. In this way, for example, we can test a service method which creates an Exhibition using fakes *TransactionManager* and repositories instances and then verify if operations have been followed in the corect order, the effectiveness interaction between involved components and updates on tested entity-objects. Now that service and repositories are tested, they can be integration tested. Integration tests belong to the Maven verify phase: it means that they are executed after the test phase (where unit tests are performed) if there are no failure. *MuseumManagerServiceIT* tests repository and service layers integration. As done before, in order to use a real database, Testcontainer is used with the same clean-up policy, running each test using an empty database. Obviously, integration tests are meant to verify higher level behaviours compared to unit tests, which are more specific; unit tests are more and more faster compared to integration tests. This concept are well illustrated in the test pyramid, represented in fig. 4.

D. Presentation layer: controller and views



Museum Louvre
Rooms 20

Uffizi - Total Rooms: 10 - Occupied Rooms: 2
Louvre - Total Rooms: 20 - Occupied Rooms: 1

Exhibition Renoir
Seats 50
Museum Louvre
Museum Name Louvre

Renoir - Total Seats: 50 - Booked Seats: 3
Raffaello - Total Seats: 50 - Booked Seats: 1
Verrocchio - Total Seats: 50 - Booked Seats: 1

Exhibitions list for Museum
Renoir - Total Seats: 50 - Booked Seats: 3

Figure 5. Museum and Exhibition Dashboards views.

The **museum-manager-presentation** module contains views and controller implementations. The controller delegates user's requests, which come from the views, to the service and gives a feedback about such operations. In fact, the controller handles exceptions which could came from the service notifying the involved view. In other words, a *MuseumManagerController* implementation has three dependencies with which interacts: *MuseumView*, *ExhibitionView* and *MuseumManagerService*; it dispatches requests from the view handling them using the service and notifies the view itself if the request succeeds.
On the other hand, views (shown in fig. 5) are implemented exploiting the Swing framework and Window Builder. Window Builder tool by Eclipse is a GUI designer; it really simplifies the GUI assembly, making available standard GUI components such as labels, input and buttons. Once the GUI is assembled, relative code is generated by Window Builder itself and it is manually integrated in view interfaces implementations successively. Then, using AWT (Abstract Window Toolkit) and Swing [13], events which call controller methods (which in turn work with service and persistence layer) and GUI logic can be implemented. This is possible since the views have a *MuseumController* dependency. In other words, each button triggers a controller method call, defining an interaction with views underlying layers; these return results or throw exceptions and the controller updates the view accordingly. Regarding inputs validation, the buttons proposed to add new Museum and new Exhibition are enabled only if text fields are not blank and numeric input (used to set Museum total rooms and Exhibition total seats) contains only digits. Finally, retrieved museums and exhibitions are displayed as selectable strings; each string represents and object entity saved into database. Summarizing, Museum and Exhibition views implementations using Swing wrap GUI code generated thank to Window Builder tool and extend the Swing JFrame class; in wrapped code, it is then possible to apply some input validation and logic to trigger controller methods, using a controller instance injected inside view implementation. [14]

1) Testing presentation layer: As done for other modules, first of all, unit tests: since *MuseumSwingController* has service and views dependencies, them are mocked using Mockito in order to verify and test only the interactions between the SUT (ie. the Controller) and its dependencies. The same concept is applied to *ExhibitionSwingView* and *MuseumSwingView*, whose unit tests have a mocked Controller to verify the correctness of method calls triggered by actions on the GUI. Furthermore, such unit tests also contains UI tests which check GUI behaviour when the user interact with it. To simulate an user playing with the GUI **AssertJSwing** library is used.

```java
@RunWith(GUITestRunner.class)
public class ExhibitionSwingViewTest extends
    AssertJSwingJUnitTestCase {
//Class attributes
private FrameFixture window;
private ExhibitionSwingView
    exhibitionSwingView;
@Mock
private MuseumSwingController controller;

@Override
protected void onSetUp() {
  MockitoAnnotations.initMocks(this);
  GuiActionRunner.execute(() -> {
    exhibitionSwingView = new
        ExhibitionSwingView();
    exhibitionSwingView
        .setMuseumController(controller);
    return exhibitionSwingView;
  });
  window = new FrameFixture(robot(),
      exhibitionSwingView);
  window.show();
  }
}
```

In above code the UI test setup for the *ExhibitionSwingView* component is reported. Beyond the use of Mockito, we can see how the view is instantiated for user's interaction simulation. The window *FrameFixture* object represents our Exhibition view; the Robot simulates user-view interaction. In tests, the window object and *AssertJSwing* API can be used to match any component in the JFrame and to make assertions about their contents or state. The **GuiActionRunner.execute** utility method is used to execute operations on GUI component in the **EDT (Event Dispatch Thread)** where *AssertJSwing* do its checks. This utility method has to be used to populate the view with data, which in the final application will be retrieved from the database, and then test relative GUI controls: for example, the "Delete Button" should be enabled only when an object is selected by the user; for this aim we should use the GuiActionRunner.execute to populate a list, simulate an item selection and then verify if the "Delete Button" is enabled. Same considerations are valid for *MuseumSwingView* test. Finally, *MuseumSwingView* and *ExhibitionSwingView* are tested in integration with *MuseumController*, *MuseumManagerService* and *Transaction Manager*, using a real database with Testcontainer and simulating GUI operations with AssertJSwing.

E. Museum Manager App

All the layers are implemented, with their responsibilities. The **museum-manager-app** module put all together. It consists in only one class, *MuseumSwingApp*, which implements a main method where all the components described above are wired. Regarding database connection, the persistence.xml file does not contain information about the PostgreSQL database name, user and password, resulting in a minimal configuration file: in fact, database credentials are passed through Command Line launching the application. This feature is implemented using *picocli* [15], a Java command line parser, which allows to initialize main class Java properties with defined CL options. Obviously, we have to be sure that a PostgreSQL database configured with such credentials is up and running. Finally, also the whole application can be tested, with **End to End tests**. End to End tests differ from integration test because they verify the correctness of the whole application running, while integration tests use a test fixture where components are manually wired together. As done before, for E2E tests Testcontainer is exploited again to obtain a fresh PostgreSQL database (consequently, there is another persistence.xml file which configures such test containerized database), while the application is started using AssertJSwing; AssertJSwing allows to start application with command line arguments (consumed by picocli) and to interact with the view using a robot, as already seen for UI tests. Before each test, database is emptied and re-populated.
As graphically described in fig. 4, E2E tests are slower and more expensive than integration and unit tests; they are meant to be less then other kind of tests and to verify only very high level functionalities.

III. Development

Code quality, software life-cycle and development tools are important topics in software engineering, also for small projects. TDD is a technique which forces us to write clean and modular code, and it can be accompanied with other frameworks which help us to be even more effective. Furthermore, VCS (Version Control System) is fundamental to organize work and to be consistent developing new features; finally, automatize all the build process makes us confident and more productive.

A. Code Coverage

Code coverage (more precisely, test coverage) measures how many lines of source code are tested. An high code coverage indicates that code is less likely to contain bugs, though this indicator is not a sufficient condition to assert this. JaCoCo is a specialized Maven plugin that executes test coverage task; when used, it provides a report with coverage information. In museum-manager-parent POM, JaCoCo Maven plugin is configured and a specific jacoco profile is defined to use it; moreover, museum-manager-report project module only goal is to aggregate all JaCoCo reports for all project modules in one place. JaCoCo report goal is bound to the verify phase; the same thing is worth to the report-aggregate goal specified in museum-manager-report module. Finally, we can also decide whether to exclude a class, for example a model class, where there is no logic to be tested.

B. Mutation Testing

During the development of Museum Manager App, beside code coverage with JaCoCo, another useful framework has been used: PIT [16] [17]. PIT is a Java **mutation testing framework**. When activated, PIT performs some mutations (accordingly to specified rules) to our source code: if at least a test fail, such a mutant is marked as killed, survived otherwise. It is evident that all the generated mutants have to be killed: if it happens, signifies that our tests are quite strong to detect such mutations in source code. As for JaCoCo, a PIT plugin exists for Maven. In this project, PIT plugin is configured in the museum-manager-parent POM, where there is also a pit profile. In this case, pit profile is very useful: mutation tests require additional build time to be executed, then it is up to us decide when perform these tests simply activating the pit profile. As done in JaCoCo, we can decide to exclude code from mutations.

C. Docker

In section II, we saw how Docker [18] [19], with Testcontainer library, came in help testing a real containerized PostgreSQL database, across all project modules. This is not the only Docker feature exploited developing Museum Manager App. In fact, Docker has been used also to **dockerize** the entire application, that is to create a Docker image for the Museum Manager Application, which potentially can be distributed or released on the DockerHub or simply ran with **docker-compose**. In the follow, all these steps are analyzed.

1) Dockerize Museum Manager Java application: Dockerize a Java application signifies that we can build our Docker image to be ran inside a Docker container, with all the benefits that containers provide. The image build can be automatically performed during the Maven build of the application, using the *io.fabric8:docker-maven-plugin*, bounded to the package Maven phase. The plugin is explicitly defined in a Maven profile, build-app, declared in museum-manager-app pom; in plugin configuration all docker build directives are configured. Two properties are fundamental: **dockerFileDir** and **jarFile**. DockerFileDir contains path to the DockerFile. The DockerFile contains the instructions that the plugin will execute to effectively generate the Docker image. jarFile contains the jar name to use during image build and it is referenced to inside the Dockerfile.

2) Fatjar: Contextually, before Docker build defined in Dockerfile is ran activating the build-app profile, a Fatjar is created. A Fatjar is a self-contained jar for a specific application. It differs from a normal jar because of its integrated dependencies. Museum Manager App has essential dependencies, such PostgreSQL driver, so a Fatjar has to be generated during the Maven package phase in order to be consumed by the Docker build later. To do this, it is enough to configure the *maven-assembly-plugin* to generate during the package phase a **jar-with-dependencies**, making this explicit in *descriptionRef* tag and specifying the main class inside the *manifest* tag. Then, the Dockerfile is updated accordingly, referencing the Fatjar for its build.

3) Docker-compose: In Dockerfile, the following command is explicated:

```
CMD ["sh", "-c" , "java -jar /app/app.jar
    --database-url=$JDBC_URL
    --database-user=$DB_USER
    --database-password=$DB_PASSWORD"]
```

This is the command which will be executed to run a Museum Manager App Docker container. A container could also be simply instantiated from command line, passing database credentials (picocli options defined in main application class). This signifies that a running database must be provided in order to run Museum Manager App image. An option is to use **docker-compose** [20], which provides containers orchestration. This mean that two container, one running PostgreSQL image for database, one running Museum Manager App, can be instantiated with only one command. For this purpose, a *docker-compose.yml* file is created; there, services, relying on specific Docker images, are defined, with their dependencies: thus, the docker-compose file for this project specifies a museum-manager-app service, its environment variables to be used, its volumes and networks, and a postgres service, on which museum-manager-app depends. The network is the museum-manager-network which uses the standard bridge. Volumes ensure that persisted data will not be lost. Then, simply running the command **docker-compose up**, containers orchestration happens: images available on DockerHub are downloaded if not present locally and environment variables are used to start defined services. Obviously, if an image is not available in DockerHub, is meant to be accessible locally; in our case image of Museum manager app will be available locally after a build ran activating the build-app profile.

D. SonarQube

Code quality is fundamental. Java compiler and IDE helps reaching this goal, but additional analysis can be performed over projects. Museum Manager App has been develop using SonarQube, a platform for continuous inspection of code quality [21] [22]. It defines three main measures: **Reliability, Security and Maintainability**, inspecting code to detect bugs, vulnerabilities and code smells; moreover, SonarQube checks presence of duplication of code and analyzes also code coverage; for the latter, SonarQube inspects provided JaCoCo reports. The SonarQube analysis is defined by rules which could also be excluded to avoid false positives; obviously, we have to be sure while excluding rules. Rules can be managed directly in the POM; in particular for this project, this is done in the museum-manager-aggregator POM.
SonarQube can be ran both locally or in the cloud with **SonarCloud**; Museum Manager App has been developed using the latter service. To do this, a Continuous Integration Server is needed.

E. GIT and CI with GitHub Actions

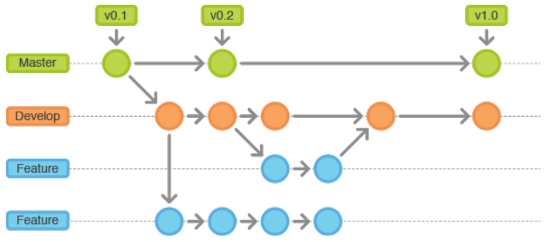


Figure 6. GitFlow paradigm.

In the previous, several tools used developing Museum Manager App were illustrated: code coverage using JaCoCo, mutation testing with PIT, containerization and container orchestration with Docker and in the end project code quality anaysis with SonarQube. As already has been stated and we can imagine, integration tests, e2e tests, mutation testing and dockerization processes can be slow and we may do not want to execute them always. The final step to build in automation is to use a Continuous Integration Server [23]. CI integrates all changes made to the code, executes project analysis and informs developers about build results. In other words, the whole Maven build project is delegated to a dedicated Continuous Integration Server. Museum Manager Application is hosted on GitHub, thus the GIT Version Control System is used developing the project, in particular the GitFlow schema, reported in fig. 6, and Pull Requests for new features implementation. A build on CI server is triggered every time a push to GIT is performed. Initially, TravisCI was used (the configuration was made in a travis.yml file); then, because of changes in Travis pricing policies, CI was migrated to **GitHub Actions** [24], which is perfectly integrated in GitHub. GitHub Actions configuration is made in yml files contained in *.github/worflows folder*, inside project root directory. Each yml file corresponds to a *workflow*, that is a build task definition. For Museum Manager App project, two workflows have been defined:

- **ci.yml**. It is the principal workflow. It is triggered always and executes PIT, code coverage and build running Maven verify phase.
- **sonar-cloud.yml**. This workflow is triggered only when push on develop or master and pull requests. It performs code quality inspection in SonarCloud running Maven verify phase.

Both workflows are configured with Docker and Java11 and use caching dependencies system for Maven, in order to speedup the build. *ci.yml* activates pit, jacoco, build-app profiles plus another: **coveralls**. Coveralls is a cloud service which keeps track of code coverage for GitHub projects analyzing JaCoCo reports; thus, after each build defined in *ci.yml* we can check code coverage results on Coveralls. Coveralls plugin is configured in museum-manager-report (bound to verify phase) and is used when coveralls profile is activated during a build. To make the flow work, a Coveralls token saved inside GitHub secrets has to be passed to GitHub Actions build command. The same mechanism is done inside sonar-cloud workflow, using SonarCloud arguments and token; for this workflow only the jacoco profile is activated to make available reports in SonarCloud. In conclusion, all developing tools used to increase code quality are integrated together on a CI server which works in symbiosis with the chosen VCS, GIT on GitHub; this technique makes us keep the work under control at each code change.

IV. Conclusion

In this project, a simple booking system application for Museums exhibitions, many developing tools and techniques have been used, in order to increase productivity and code quality. First of all, Eclipse IDE was chosen for its plugins and keyboard shortcuts in code writing. Then, Git was chosen as version control system, GitHub in particular, for its versatility and easy integration with the other tools used for this project such as Coveralls, for code coverage, SonarCloud, for code quality analysis and GitHub Actions, as CI server. Docker turned out as a powerful containerization technology, both for testing and application deployment; finally, the use of TDD alongside the mentioned techniques makes us more and more confident about our code correctness.

References

[1] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*. Addison Wesley Professional, 2005.

[2] Maven official website. [Online]. Available: https://maven.apache.org/

[3] L. Bettini, *Test-Driven Development, Build Automation, Continuous Integration*. LeanPub, 2020, ch. 7.

[4] Hibernate ORM. [Online]. Available: https://hibernate.org/

[5] JPA. [Online]. Available: https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html

[6] L. Bettini, *Test-Driven Development, Build Automation, Continuous Integration*. LeanPub, 2020, ch. 5.

[7] JaCoCo in EclEmma. [Online]. Available: https://www.jacoco.org/jacoco/index.html

[8] Maven Wrapper. [Online]. Available: https://github.com/takari/maven-wrapper

[9] HikariCP. [Online]. Available: https://github.com/brettwooldridge/HikariCP

[10] Testcontainers library. [Online]. Available: https://www.testcontainers.org/

[11] Mockito framework. [Online]. Available: https://site.mockito.org/

[12] L. Bettini, *Test-Driven Development, Build Automation, Continuous Integration*. LeanPub, 2020, ch. 8.

[13] Swing and AWT. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/swing/index.html

[14] L. Bettini, *Test-Driven Development, Build Automation, Continuous Integration*. LeanPub, 2020, ch. 13.

[15] picocli CLI. [Online]. Available: https://picocli.info/

[16] L. Bettini, *Test-Driven Development, Build Automation, Continuous Integration*. LeanPub, 2020, ch. 6.

[17] PIT framework. [Online]. Available: https://pitest.org/

[18] L. Bettini, *Test-Driven Development, Build Automation, Continuous Integration*. LeanPub, 2020, ch. 11.

[19] Docker. [Online]. Available: https://www.docker.com/

[20] Docker Compose. [Online]. Available: https://docs.docker.com/compose/

[21] L. Bettini, *Test-Driven Development, Build Automation, Continuous Integration*. LeanPub, 2020, ch. 15.

[22] SonarQube. [Online]. Available: https://www.sonarqube.org/

[23] L. Bettini, *Test-Driven Development, Build Automation, Continuous Integration*. LeanPub, 2020, ch. 10.

[24] GitHub Actions Docs. [Online]. Available: https://docs.github.com/en/free-pro-team@latest/actions