TQS: Quality Assurance Manual

Leonardo Flórido [103360]*, *Gabriel Hall [102851]*, *Tomás Almeida [103300]2023-06-02

1 Project management

1.1 Team and roles

Team Coordinator – Gabriel Hall

Responsible for:

1. Fairly coordinating the team and distributing tasks among members.
2. Taking initiative to solve problems that may arise.
3. Conducting regular checkups to ensure everyone and everything is on track.
4. Ensuring timely delivery of the project's expected outcomes.

Product Owner – Tomás Almeida

Responsible for:

1. Deep understanding of the presented problem and the ability to effectively represent stakeholder interests.
2. Capability to address team members' questions regarding the expected system features.
3. Involvement in accepting solution increments and ensuring the correct functionalities are delivered and functioning as intended.

DevOps – Leonardo Flórido

Responsible for:

1. Managing the development and production infrastructure, including required configurations, to ensure proper functioning.

2. Preparing deployment machines/containers for initial setup and readiness.
3. Creating and preparing the Git repository for team members' access and coding.
4. Setting up additional technologies like cloud infrastructures or database operations as necessary.

Developer – All
Every member contributes to the development of the project tasks.

1.2 Agile backlog management and work assignment

For backlog management and work assignment, we use GitHub's Project Management features.

The project contains its own tasks based on user stories, categorized into "To Do", "In Progress", and "Done" columns.

To begin work on a user story, a developer must create an issue, which represents the task being worked on. The issue includes a concise title, labels for categorization, assignment to a specific project, and the ability to assign it to one or more team members for efficient job assignment. Once all the work is completed, the issue should be marked as closed.

When a developer is working on an issue, they should indicate it as "In Progress" to inform other team members that it is being worked on. When an issue is closed, it is automatically moved to the "Done" column.

2 Code quality management

2.1 Guidelines for contributors (coding style)

For this project we followed the coding style AOSP Java Code Style and Python Code Style.

2.2 Code quality metrics

The main tool we used for static code analysis was Sonar Cloud. Each push to a branch and pull request to "main" triggered the analysis. This allowed developers to verify the quality of their code during development and one final time before merging their changes.

The quality gate used was the built-in quality gate recommended by Sonar Cloud, which focused on keeping new code clean to minimize the project's debt time and lower the effort required to improve old code.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

To facilitate the development of new features, we utilized the Git Feature Branch Workflow. We had a main branch that was updated whenever a release was ready.

A user story was considered complete when all necessary functionalities were developed, subjected to both static and system testing, and successfully passed. The code has been revised and the system with the new features is ready to be used by the target users.

3.2 CI/CD pipeline and tools

To implement Continuous Integration into our project, we used GitHub Actions, which provided an easy way to build pipelines and run them based on specific conditions.

We created a script that ran on every push and pull request, associated with the project. This script was responsible for building the project and running available tests. If these phases were successful, the project was analyzed, and the results were made available on the Sonar Cloud platform. The script ran on the latest Ubuntu image.

build.yml
This workflow will trigger 2 jobs:

- Run the projects tests
- Run the SonarCloud static code analysis for the the project and check if it passes the quality gate

For each project, we have a separate **docker-compose.yml** file. This file enables us to run the projects more smoothly and simplifies the deployment process when deploying our projects on Google Cloud.

4 Software testing

4.1 Overall strategy for testing

To test our backend functionalities, we planned to use TDD. We wrote tests for the features we were developing before implementing the actual methods and logic required. This approach required developers to consider exceptions and intended return data before implementing the methods themselves.

For testing our user interface, we chose to use BDD by integrating Cucumber with Selenium. This allowed our tests to emulate real scenarios.

All tests were developed using JUnit 5 as the base framework.

To isolate components, we used Mockito to test them individually, without interactions between different components. As mentioned earlier, we used Selenium to simulate user behavior for the user interface, and Cucumber to represent real usage scenarios.

4.2 Functional testing/acceptance

For functional testing, we chose to use Selenium, which can be used with Edge by utilizing the respective driver.

We also opted for using Cucumber, which simplifies writing tests from the end-user's perspective. To achieve this, the different web app features should be separated into individual feature files. These feature files should include all the necessary scenarios that users could encounter, including both valid and invalid scenarios, to ensure our system correctly validates and responds to user input.

4.3 Unit tests

Unit Testing was performed for all appropriate classes, repositories, services, controllers, and any other extra classes that were developed and used. All relevant methods were tested with at least one

valid and one invalid input to ensure there were no unexpected behaviors. When testing a component, all interactions with other components were mocked to successfully isolate it.

4.4 System and integration testing

Integration tests were developed for all controller endpoints to simulate real interactions with the REST API. These tests were like the unit tests developed for the controllers, but there was no mocking of the services, relying on the actual implementation.

These tests used a test database, which was created at the start of the tests and deleted at the end to ensure safe execution and prevent interference with the state of the actual database.