TQS: Quality Assurance manual

André Miguel Antunes Cardoso [108269], Tiago Alexandre Abreu Figueiredo [107263], Alexandre Pedro Ribeiro [108122]

v2024-06-03

1 Project management

1.1 Team and roles

André Miguel Antunes Cardoso - *Team Leader & Product Owner*
Responsible for defining the product specification and user stories.

Tiago Alexandre Abreu Figueiredo - *QA Engineer*
Responsible for defining the quality assurance practices and development guidelines.

Alexandre Pedro Ribeiro - *DevOps master*
Responsible for creating the CI/CD pipelines.

All the above are also developers, responsible for developing the features according to the product owner's specification.

1.2 Agile backlog management and work assignment

User Stories should be written in a concise manner, in the form of "As a [user], I want [feature] so that [benefit]. These user stories are then organized in Jira and grouped as part of bigger feature.
The backlog must be updated at least once per weak to determine what user stories should be prioritised for the current iteration.
Due to the small nature of the team, the assignment of the work is done on a first come first served basis, with some exceptions to accommodate developer preferences on working on the frontend or the backend services.

1.3 Guidelines for contributors (coding style)

This project it is recommended to follow the Google Java Style Guide, with some exceptions:

- The section on JavaDoc is to be ignored as there is no requirement to use JavaDoc on this project.
- Wildcard imports are allowed for the Java standard library and testing libraries.
- The use of non-ascii characters is discouraged
-

Similarly, for the JavaScript project it is recommended to follow the Google JavaScript Style Guide, with, again, some exceptions:

- The sections on the usage of JSDoc are to be ignored, as this project uses TypeScript, which provides much of the same functionality, with, however, the disadvantage of not generating the reference documentation.
- The column limit should be set a 100 instead of 80, with an additional exception, of highly indented template code (JSX).
- The use of highly indented JSX is itself also not recommended, except where unavoidable.
- The use of the spread operator is discouraged due to its underlying implementation.

Whilst these guidelines should be followed, they are not enforced during CI.

1.4 Code quality metrics and dashboards

Static code analysis is done when a Pull Request is opened, and for each commit done on the branch being merged until the Pull Request is approved and merged. This analysis is done on SonarCloud and trigged via GitHub Actions.
There is only one quality gate defined that requires all code merged to have a greater than eighty percent code coverage, less than three percent duplicated lines, no new bugs or vulnerabilities, a complete review of all if any, and limited new “code smell” introduced.
This quality gate was chosen as it helps prioritize the quality of new code, whilst making sure that new code is maintainable.
Only Java code is counted towards these metrics, completely ignoring any UI code written on the frontend.

2 Continuous delivery pipeline (CI/CD)

2.1 Development workflow

When the user story is assigned to a developer, a new branch is created with the same name of the task in Jira. After the work is completed, a review is requested to another developer to ensure that quality targets are met and that the feature is complete.
In some cases, when additional work is required, or a new subtask is created, a new branch is created following the naming convention of “improv/*” or with the name of the project being improved.
When an aggregation of features deemed ready for release is done, a new pull request is created to merge develop into master.
A user story is considered done when the goal set in its title is accomplished and the quality gate is met. For example, a user story that has the following title “As a user, I want to sign-in and login into the service”, is considered complete when both the backend service supports the logging in of users, the frontend has an interface that allows this to be manually tested, and the quality gate is met, with all tests passing.

2.2 CI/CD pipeline and tools

For CI, we use GitHub actions, running workflows like the static code analysis.

For CD, due to time constraints and problems with UA's VPN, we didn't manage to implement a continuous delivery pipeline. Instead, we manually deployed the project to the VM.

2.3 System observability

While a GitHub action is running, a live log is displayed on the GitHub's repository page, and afterwards, a complete log is available for a set amount of time.

At the end of the Sonarcloud actions, which had 2 steps (Build&Analysis, Results), a link was provided for the analysis dashboard of the specific branch that was being worked on.

3 Software testing

3.1 Overall strategy for testing

For this project, we did Feature Branching, creating the branches through Jira and developing the selected feature until its implementation was successful. Because we are a smaller team and each one of us had other projects to work on, we decided that we weren't going to do TDD, but instead develop the code first and then write the necessary tests to save time.

For testing, we used AssertJ and Mockito for the lower layers and Rest-Assured to test the functionality of our API.

3.2 Functional testing/acceptance

When testing our API, tests should be written in the perspective of the user and simulate real data where we could, which can be seen in the Rest-Assured tests.

For the frontend manual testing is to be done due to time constraints but maintained the same policy.

3.3 Unit tests

When writing unit tests, we applied the knowledge we have of our code and developed tests that approved the expected results while focusing on our developer perspective, meaning the tests are harder for outsiders to understand but easier for us developers.

This is reflected in the tests using AssertJ and Mockito, most relevant in the Service layer.

3.4 System and integration testing

When testing the Repository layer, we used TestEntityManager to integrate a real database into our tests, which made it possible to detect schema conflicts and data insertion problems that otherwise would have gone unnoticed.