

TQS: Quality Assurance manual

Francisca Inês Marcos de Barros [93102], Gonalo Andr  Ferreira Matos [92972], Isadora Ferreira Lored  [91322], Margarida Silva Martins [93169]

v2021-06-15

1	Project management	1
1.1	Team and roles	1
1.2	Agile backlog management and work assignment	2
2	Code quality management	2
2.1	Guidelines for contributors (coding style)	2
2.2	Code quality metrics	2
3	Continuous delivery pipeline (CI/CD)	3
3.1	Development workflow	3
3.2	CI/CD pipeline and tools	4
4	Software testing	5
4.1	Overall strategy for testing	5
4.2	Functional testing/acceptance	5
4.3	Unit tests	5
4.4	System and integration testing	5

1 Project management

1.1 Team and roles

To better structure our team work, despite that everyone contributes as a developer, each person has a major role. This does not mean that it is the only person who plays that role, but the person that must have full ownership over it and so, should be referred to in case of doubts about that specific field. The roles assigned to each member are the following:

DevOps Master Gonalo

Product Owner Francisca

QA Engineer Isadora

Team Coordinator Margarida

1.2 Agile backlog management and work assignment

In order to completely implement agile techniques, the group opted to adopt the *ZenHub* platform¹ for backlog management, since it is one of the few project management software solutions that fully connects with *GitHub*. It provides a plugin for connecting to the selected repository.

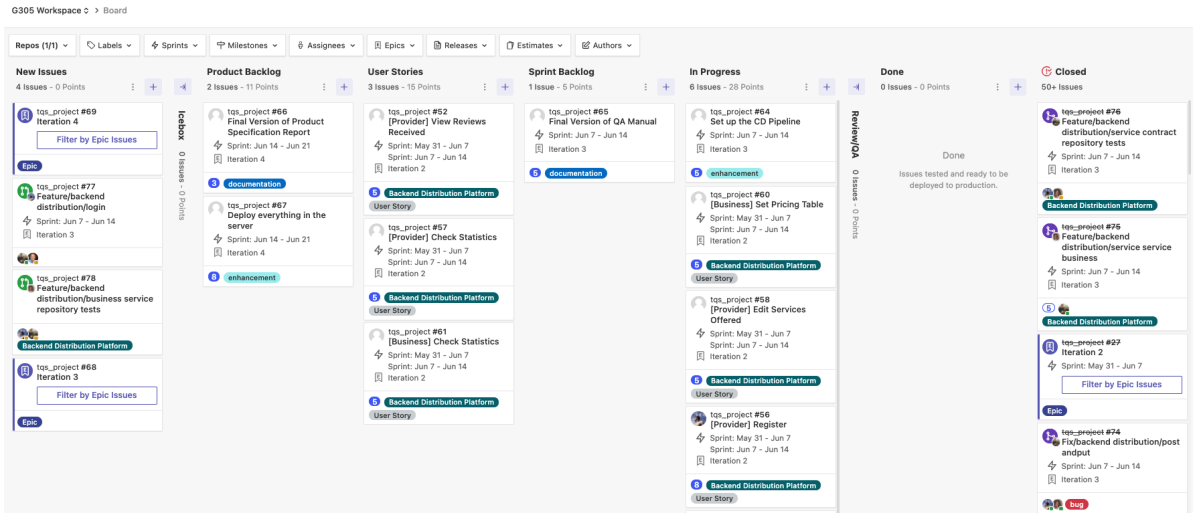


Figure 1 - Group's Workspace Board

The weekly sprints were established on the board, which made it easy to track the team's progress throughout the project. Different pipelines were established, as shown in the image above, in order to better organize what was expected to be our major emphasis during our week-long sprint.

Our board can be accessed in the following link - [ZenHub Workspace Board](#)

2 Code quality management

2.1 Guidelines for contributors (code style)

When it comes to coding style, consistency is everything. As a result, establishing criteria for contributors benefitted the team in developing a consistent coding language.

We opted to use the Android Open Source Project's Coding Guidelines for the majority of our project's code (the backend, in Java) since it was compatible with the team's regular code-style.

2.2 Code quality metrics

For static code analysis, SonarCloud was selected as the medium to facilitate this process because it was simple to use in practical classes.. Pipelines for every aspect of this project (being, each frontend platform and backend platform) were defined on GitHub Actions, making sure that the analysis was run individually, and published accordingly.

Regarding both backend platforms, the Quality Gate shown in the image below was defined for every code analysis, run on the develop branch when opening up Pull Requests and allowing merges.

¹ <https://www.zenhub.com/>

Conditions on New Code

Conditions on New Code apply to all branches and to Pull Requests.
















Metric	Operator	Value	Edit	Delete
Coverage	is less than	70.0%		
Duplicated Lines (%)	is greater than	5.0%		
Maintainability Rating	is worse than	A		
Blocker Issues	is greater than	0		
Critical Issues	is greater than	5		
Reliability Rating	is worse than	A		
Security Hotspots Reviewed	is less than	100%		
Security Rating	is worse than	A		

Figure 2 - Quality Gate defined for both backend platforms

In terms of coverage, we opted to drop the default value from 80 percent to 70 percent because there were many circumstances that were not covered by any tests, and therefore would be decreasing this value incorrectly.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

The method chosen to structure our GIT repository was GitFlow², the branching model focused on incremental releases. Essentially, it defines rules for branch names that allow the project to have a predictable structure so that every member can easily understand what kind of work is being done in every branch. In addition to these rules, there are also protection rules³ applied to the main branches, to make sure the rules are followed. The name patterns used in this project, are the following:

main

Main branch, used by production environment. Stores the official release history. One commit (through Pull Request) per release done by the DevOps Master and reviewed by a team member.

develop

Branch for development, the most used during each iteration, where the feature branches are created and merged to.

² <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

³

<https://docs.github.com/en/github/administering-a-repository/defining-the-mergeability-of-pull-requests/managing-a-branch-protection-rule>

feature/[component]/[featureName]

Branch for a new feature. Created (from the develop branch) by any developer to develop a feature on one of the project components. When finished, should be merged into develop with a Push Request, that must be reviewed by a team member that is not the author before merging (the reviewer is responsible for merging). The component for which the feature is being developed must be identified by one of the following names: frontend-distribution, frontend-end-user, backend-distribution, backend-end-user.

fix/[component]/[featureName]

Branch for corrections during interactions. It is created from develop to make a quick fix and is merged there, with a PR with peer-review.

release/[number]

Branch for release. It is created from develop at the end of each sprint, for the final adjustments. When it is created it is not possible to add new features, only to correct bugs and write documentation. When the release preprocessing is done and the product is ready for the official release, it must be merged into main, identifying the merge commit with the version tag.

hotfix/[hotfixName]

Branch for bug corrections on production. It is created from main to correct a problem with the software running in production. When the problem has been fixed, it is merged into main and develop.

doc/[component]/[docName]

Branch for documentation. The component name should be the same as the names defined for the feature branch. If the documentation applies to the project as a whole, the component name should be project.

3.2 CI/CD pipeline and tools

For CI and CD methodologies we have recurred to GitHub tools. For the Continuous Integration process we have set up GitHub actions, that runs tests every time a pull request is opened or a branch is merged into develop, to assure that this branch is always stable. Besides tests, it is also integrated with Sonar Cloud for static code analysis.

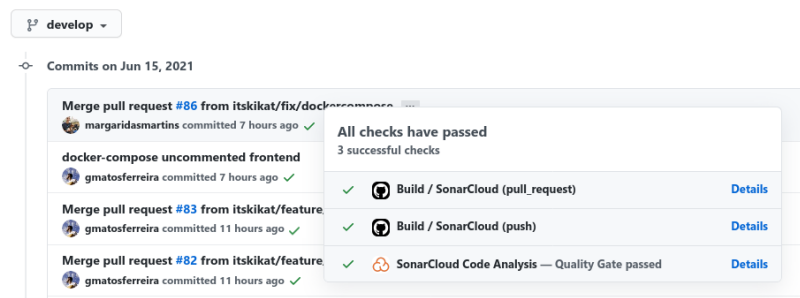


Figure 3 - Example of an action for testing triggered by a PR into branch develop

When it comes to the Continuous Deployment, we have also set up a GitHub actions pipeline that deploys our work to the Virtual Machine every time a commit is made to the main branch. To make this happen, we have a self-hosted runner in our VM, which is always working in the background.

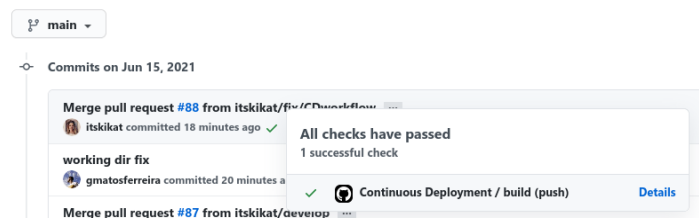


Figure 4 - Example of an action for deployment triggered by a push into branch main

This process is done with Docker and docker-compose, which creates a container for every component of our system (one for the database, two for each backend and other two for each frontend).

4 Software testing

4.1 Overall strategy for testing

This project development process follows a TDD approach.

To build the test battery, several testing tools are used, which allows it to cover all the scenarios: Mockito, JUnit, AssertJ, Cucumber, Selenium, SonarQube, Jacoco, SonarLint inspection tool plugin for IntelliJ.

The tests directory was structured by subfolders that allow for a better understanding.

4.2 Functional testing/acceptance

Selenium automation ecosystem is used for functional testing, which allows to replicate the user interaction with the web page and assure that it meets the expected requirements. To make the code more legible, avoid duplication and ease maintenance, the Page Object Model pattern should be followed, isolating the test implementation from its usage. To make sure the tests match the user scenarios, they must be written with Cucumber.

4.3 Unit tests

For every API endpoint that I created, the following unit tests must be implemented:

- Unit test for the controller with mocking of the service;
- Unit test for the service, with and without (2 for each service) mocking of the external API.

Every other class that is not a data class and implements some functionality (for example a cache service) must also have implemented unit tests to assure that it presents the expected behaviour in all scenarios.

4.4 System and integration testing

For every API endpoint created, integration tests must be implemented to validate the application API, with the full web context.

