TQS: Quality Assurance manual

Igor Coelho [113532], João Capucho [113713], Zakhar Kruptsala [114478], Victor Milhomem [128660]
v2025-06-09

Contents

1 Project management

1.1 Assigned roles

Whilst all members of the team are developers, the following roles are distributed to the members. These members will be responsible for their assigned area, however this does not prevent others from also helping in these areas.

Name	Role
Igor Coelho	Team Coordinator/Team Leader
João Capucho	DevOps master
Zakhar Kruptsala	QA Engineer
Victor Milhomem	~~Product owner~~ Disappeared person

1.2 Backlog grooming and progress monitoring

Jira is used for all project management needs, the work is organized in user stories that each must be assigned to an epic. More complex user stories can be divided into subtasks, when those subtasks can be done independently of one another. Developers must be assigned the smallest work item, which usually corresponds to a user story, but can also be the subtask of a user story as previously mentioned. A developer, before assigning itself a work item, must first discuss with the remaining team, to ensure the full scope of the work is understood and conflicts with other work are properly addressed. Once the developer has gotten the permission of the team to assign a work item, it can either be done by self assignment or by the team member responsible for the part of the project that will be worked on. A developer should under normal circumstances only have one work item assigned to them.

All user stories must be defined in the classic template of “As a [persona], I [want to], [so that].”, have accompanying acceptance criteria, and a cucumber test for said criteria, before they can be assigned to any developer.

All work is organized in sprints, where only epics (and related user stories), bugs, and standalone tasks (i.e. that aren’t part of a user story) are eligible to be scheduled in a sprint. This helps ensure the delivered work is consistent from a utilization point of view. Sprint planning meetings are scheduled each week to discuss the work done in the last sprint, review the backlog for missing or underspecified work items, and to create the next sprint. It’s also at this time that it is recommended that developers discuss work items they will work on. During the sprint, more work can be assigned as developers finish their work items.

Team progress is tracked using burnup reports for per sprint progress and velocity reports for overall team productivity. The burnup report is consulted throughout the sprint to inform the team of the current and expected pace. The velocity report is used in the sprint planning to help the team manage the amount of work that is scheduled for the sprint.

Requirements level coverage is done using Xray, a test management tool integrated directly in Jira, which not only permits analyzing requirements test coverage, but also trace their coverage all the way from epic to test execution.

2 Code quality management

2.1 Team policy for the use of generative AI

While the use of such tools isn't forbidden by the team, it is strongly discouraged for production code and even for test code (except for the most mundane of boilerplate). This decision is made given the lack of reliability exhibited by these tools, the massive amount of data exposed to the tool operator, and the legal limbo that the code generated by them encounters itself in.

Whilst their use is not forbidden, all the code committed by team members is their sole responsibility, any errors committed by these tools will be your sole responsibility if you choose to use them, so always triple check the generated code.

If you do choose to use these tools, follow these best practices:

DOs	DON'Ts
• Prefer use of a local LLM to prevent leakage of data to third parties. • Always double-check the generated code. • Check the generated code for duplication, especially when using multiple sessions. • Ensure the generated code follows the coding style defined in the next section. • Use the tool to generate small parts of the code, instead of all the code at once, this allows you to more easily review the generated code and guide the tool with more control.	• Do not **ever** give production data to the tool, even if it's running locally. • Do not run commands provided by the tool directly without first examining it. • Do not try to integrate any MCP or similar tool with any production service.

2.2 Guidelines for contributors

Coding style
The coding style followed by the team is based on that of the Android Open Source Project (AOSP) coding style. This includes following the java language rules. However, we don't require adding a copyright statement at the top of each file, we don't impose rules on the order and grouping of imports, and we don't enforce field naming conventions.

Code reviewing
All changes before being accepted must be reviewed by at least one other team member (this is enforced in CI). Changes that are not yet part of a PR do not need a code review, however, one can be requested by the author of the changes, or proactively by a team member, if it is deemed relevant enough to provide early feedback on the changes.

When doing code reviews the team members should aim to catch bugs and code smells proactively, they should also suggest clarifications to complex parts of code such that when another team member later reads that code, they can clearly understand its function. Any discussions related to code style should immediately be deferred to the coding style.

Currently, the team doesn't intend to integrate AI code reviewing tools due to their high noise to signal ratio. Team members may choose to review their own code with one of these tools. However, when performing a code review for another team member, they must not use these tools.

2.3 Code quality metrics and dashboards

To further help with maintaining the project overall code quality, static code analysis is used throughout the project. More specifically, sonar cloud that is integrated directly with the development workflow through the use of GitHub Actions in CI.

The default quality gate of sonar cloud is used, as it is deemed sufficient for the needs of the project. The coverage, while expected to be close to 100%, is sometimes below that because some parts are not deemed relevant for testing like `toString` and `hashCode`, as such the default of more than 80% coverage fits nicely. The team did also research at one point in enforcing a 0 issue policy, however, it was found that the false positive rate was high enough to cause pull requests to be blocked unnecessarily, needing team members to resolve issues and re-trigger CI. This caused the development to needlessly stall, as such it was decided to stick with the default "Maintainability rating is A" rule.

While the quality gate does not enforce, authors and code reviewers should check new issues and fix relevant issues and mark false positives. Likewise, code coverage should also be checked to make sure all relevant parts of the code are covered.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow
The coding workflow starts with the team member being assigned a work item, as is described in Backlog grooming and progress monitoring. Afterward they create a branch for their work item, we follow the GitHub flow with feature/fix branches per work item which all get merged into a main branch, that later is tagged to be deployed to production. The main branch is protected, so it can only be modified by pull requests. So, the next step, after creating the branch and pushing the changes, is to open a PR targeting main. Afterward the code review process defined in Guidelines for contributors is performed, and after everything is accepted the PR is merged in a merge queue to ensure that multiple PRs when merged together don't break.

Definition of done
A work item is considered done when the relevant criteria pass and tests have been added to confirm that no regression happens. When dealing with user stories, this means that the acceptance criteria are met, the BDD tests are implemented, and any auxiliary tests as needed are also implemented. For bug work items, the bug itself must be corrected, and at least one regression test must be added to prevent the bug from appearing again. For task work items, the acceptance criteria and needed tests are dependent on the task itself, so the code reviewer should consider if the objective of the task is fulfilled and which tests are warranted for such.

3.2 CI/CD pipeline and tools

The continuous integration of the project is based on two pipelines, one for the frontend and another for the backend. Both are developed using GitHub actions and run in pull requests, in the development branch, and in the merge queue.
The frontend pipeline validates the code style using the prettier formatter, code smells are caught using the eslint linter, and the application is built which additionally type checks the application with typescript. Finally, lighthouse is executed over the built application to test the quality of the application.
The backend pipeline runs both the unit tests and integration tests, the latter of which are uploaded to Xray to calculate requirements coverage, additionally the results of both executions are merged in a JaCoCo report which is uploaded to sonar cloud, to analyze the code coverage and maintainability of the project.
The continuous delivery is also developed using GitHub actions, and runs whenever there is a push to the development branch or a new version tag is created. This pipeline builds the backend and frontend docker containers and pushes them to the GitHub container registry. These will then be synced by Argo CD running in the machine and the deployment automatically updated.
When continuous delivery is run against the development branch, an additional job is run for end-to-end testing. This job runs in a self-hosted runner where the staging environment is located, it starts by first forcing the deployment to sync immediately and waiting for the new version to be available. Afterward, the e2e tests (BDD) are run directly against the staging deployment and the results uploaded to Xray. Finally, some test data is loaded into the staging environment and load tests are performed to ensure the application achieves the service level objectives.

3.3 System observability

Three major groups of metrics are collected to allow continuous evaluation of the operational health of the deployment. First the node/VM metrics like CPU utilization, memory usage, network ingress/egress, and storage usage and latency. These allow us to detect when the machine is overloaded and more resources are needed, or an application is hogging resources, causing slow-downs in other applications. The second set of metrics, are database metrics, active connections, read/write latency and throughput, and size. These, like node metrics, allow us to detect when the database needs more resources, but also allows us to correlate high latency application requests to database usage, which might indicate problems in the database or a poorly optimized query. Finally, application metrics are collected, like JVM statistics, http request statistics, and logs. These allow us to monitor application health and detect anomalous behavior, which can happen because of bugs in the application but also bad actors targeting the application.

3.4 Artifacts repository

The only artifact repository used in the project was the GitHub container registry. Some other artifacts are generated in the CI pipelines, mainly lighthouse reports, but these are not stored in any repository, instead being directly downloaded by developers from the actions page.

4 Software testing

4.1 Overall testing strategy

The test development strategy favored by the team is based on the classic Test Driven Development approach (TDD), where first a set of tests that model the module under implementation are defined and implemented, and only after that is the code written. In addition to the standard TDD workflow, the team also adopts Business Driven Development (BDD) for testing user stories and scenarios.

For defining unit tests, Mockito is used to mock all the dependencies of the unit under test to simulate various different conditions. Integration tests are used for testing the user exposed components to ensure they all properly work together, the spring test tools are used, mainly MockMvc, as it properly integrates with the spring security stack. Finally, end-to-end tests are used to test the user stories and are defined in cucumber and implemented with selenium. The classic testing pyramid approach is used, with user stories being comprised of a few e2e tests, normally one cucumber file with a couple of scenarios, more integration tests, testing the components needed to implement the user story in conjunction with one another, lastly many more unit tests exist to test each component in isolation.

The integration and unit tests are automatically checked by CI using GitHub actions on push to the main branch and in pull requests. These ensure that tests are always passing, and are also used to block pull requests from being merged which do not pass testing. The results are also uploaded to Xray and sonar cloud to ensure continuous coverage analysis.

4.2 Functional testing and ATDD

The team policy for writing functional tests is that they should be written before the implementation of the feature they will test. This helps ensure that the test doesn’t depend on any details of the implementation (closed box), it also follows more of the expected user perspective, which also doesn’t have any idea of the innards of the application. These tests are written using cucumber and selenium and should first be documented in Jira and reviewed by a team member.

The developer should write at least one of these tests before implementing a new feature in the application.

4.3 Developer facing tests (unit, integration)

Unit tests should be written whenever a new component is created, an existing component is extended, or when fixing a bug in the component (as a regression test). These tests are written with Mockito. While all components should have unit tests, special care should be given to services (business logic) and repositories with special operations and queries (i.e. native queries).

Integration tests should be written for each set of component trees (normally these start at the controller level, but can also start at the service level for background tasks) and are to be developed using the closed box approach and before the implementation is written to ensure no development details leak into the tests. These tests use the spring test infrastructure and the MockMvc (other alternatives were tested, but only MockMvc properly integrated with spring security).

4.4 Exploratory testing

Exploratory testing is to be performed always, before releasing a new production version. These tests are to be conducted in the staging environment to ensure realistic operating conditions. Besides obvious flaws and bugs, testers are also encouraged to document confusing interaction flows, bad accessibility, and general application perceived performance.

4.5 Non-function and architecture attributes testing

Performance tests are to be written for features that are expected to be executed frequently or that are expected to be more costly than normal. These tests are to be written using k6 and should aim to test at least 120% of the expected maximum load to ensure some margin in case of unexpected utilization bursts.