TQS: Quality Assurance manual

Diogo Marto [108298], Miguel Pinto [107449], Tiago Pereira [108546]

V2024-06-03

1 Project management

1.1 Team and roles

Team Leader – Tiago

- Ensures fair distribution of tasks.
- Promotes collaboration within the team.
- Addresses problems proactively.
- Ensures timely delivery of project outcomes.

Product Owner – Miguel

- Represents stakeholder interests.
- Has deep knowledge of the product and application results.
- Clarifies questions about product features.
- Involved in accepting solution increments.

QA Engineer – Miguel

- Promotes quality assurance practices.
- Implements tools to measure deployment quality.
- Monitors that team followed the established quality assurance practices.

DevOps master – Diogo

- Manages development and production infrastructure.

- Ensures the development framework functions properly.
- Prepares deployment machines/containers, manages the git repository.

Developer – Everyone (includes possible future new-comers)

- Contributes to development of tasks.
- Tracks work through pull requests, commit history in team repository and Jira backlog.

1.2 Agile backlog management and work assignment

Backlog management tool: Jira (see it here).

User stories:

- Formatted as the following “As a [role], I want [feature], so that [benefit]”.
- Grouped in Epics for core system parts or functions (admin app, client app, authentication, etc.).
- Linked to a branch, pull requests and tests (all using Jira).

Prioritization: core parts are to be prioritized (such as authentication) and may break normal branch and pull request workflow, these priorities will be defined by the team leader.

Sprint duration: 1 week throughout the whole project with goals to meet team-wise.

Work assignment: tasks assigned based on availability, expertise, and workload of each team member.

2 Code quality management

2.1 Guidelines for contributors (coding style)

- Indentation: We use spaces for indentation, with 4 per level.
- Naming Conventions: Variables, functions, and classes follow the camelCase convention for consistency.
- Comments: We tend to minimize the use of comments as we think the code should be auto descriptive.
- Commit Logic: When committing changes, please ensure to follow the commit message guidelines that is Type: Message e.g. Fix: fix visual bug in frontend. Clear and descriptive commit messages help in understanding the rationale behind the changes and with the verbs in the infinitive.
- Branches: Branches for features are created via Jira and have the TP-X User Story.
- Error Handling: Use meaningful error messages and appropriate error handling mechanisms.
- Code Modularity: Break down complex functionalities into smaller, modular components. This enhances code reusability, maintainability and makes the code more auto descriptive.

2.2 Code quality metrics and dashboards

We use sonarcloud to run static code analysis on our code when it is pushed to the development branch. We have the following quality gates:

- Code coverage above 70% - this guarantees good coverage of our code. It isn´t higher because some parts of our frontend have quite a lot of parts that doesn’t make sense to test.
- There are no issues with the code – this is to follow the clean code style and that there are no security issues.
- No more then 3% of duplicated lines – this originates coder that is easier to maintain

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

We are using GitHub flow. Each user story in Jira maps to a feature branch, when the user stories are completed a pull request is created to the development branch, this pull request is then reviewed by other people who didn't develop the story and compared against our definition of done (this includes passing certain actions on GitHub) if the requirements the pull request is accepted and the branch is merged to development. When a release is ready the development branch is merged into the main branch.

Definition of done:

- Code completed (backend and frontend)
- Successful build
- Passed the acceptance criteria of story
- All test pass
- SonarQube quality gate passes

In Jira, when there are no assignees to a story the story has the state *TO DO.* When somebody is working on a story the state is IN PROGRESS. When somebody finishes working on a story, they open a pull request and change the state of the story to IN REVIEW. When the story is reviewed and accepted its state is changed to DONE by the product owner.

3.2 CI/CD pipeline and tools

We used GitHub actions to create CI/CD pipeline. We have 3 actions defined:

- **CI** action is responsible for building and running tests of the application. The test results are then pushed to sonarcloud and Xray. This action runs every PR and push to main or development.
- **Docker publish backend** and **frontend.** These two actions run whenever there is a push to the main and they build and publish docker images to GitHub container registry.

In order to achieve continuous deployment, we have on the server a scheduled task that checks if there are any new alterations to the published images. When there are new versions of the images, we run docker-compose to create new containers based on the new images and remove the old ones. This happens without user intervention.

3.3 System observability

We have logging in the services and controllers' classes in the backend. On sonarcloud, we can also see issues if there exist in our code. On Jira we have access to the test results via the Xray plugin which then are linked to each story, we can also see the state of each story (e.g. if it is done). We also have a quality assurance dashboard that allows us to quickly get an overview of the hole project in Jira.

4 TODO - Software testing

4.1 Overall strategy for testing

We follow a BDD approach where acceptance criteria are defined for each user story. These criteria are then translated into automated tests using the Gherkin language. This approach ensures that all functionalities are tested from a user's perspective, providing a clear understanding of the system's behavior

Test driven development is not mandatory, but there are policies for each test category.

There should be tests for service logic, repository access (with real database), controller logic and functional using Cucumber. Notice entities are excluded as there should be no major logic besides database constraints that are also handled in the service logic.

It is mandatory to test happy cases which are the ones that confirm said functionality exists and some unhappy cases might be considered (especially on the test where some dependencies are mocked).

4.2 Functional testing/acceptance

- Policy: Functional tests are written from a user's perspective (black box testing). These tests validate the end-to-end functionality of the application and ensure that the system meets the defined requirements.
- Tools: Cucumber, Selenium
- Resources: User stories and acceptance criteria defined in Gherkin language.

4.3 Unit tests

- Policy: Unit tests are written from a developer's perspective (white box testing). These tests validate individual components or methods to ensure they work as intended.
- Tools: JUnit, Mockito
- Resources: Source code, design documentation.

4.4 System and integration testing

- Policy: API tests validate the RESTful endpoints of the application. These tests ensure that the APIs perform as expected and adhere to the specified contracts. Tests made to the API layer are more black box, tests between boundary of API and service layer and service layer and persistence layer are more white box testing.
- Tools: JUnit, Mockito, MockMvc, RestAssured
- Resources: API specifications, endpoint documentation.