deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Joana Amaral Gomes [104429], Artur Denderski [122282], Maria João Machado Sardinha [108756], Mateusz Kubiak [122385]
v2023-04-18

1 Project management

1.1 Team and roles

All members of the team contribute as **Developers** in the project, additionally:

- Maria João Sardinha is the **Team Coordinator**, responsible for making sure everyone has a fair share of tasks and works according to the plan, promoting teamwork and taking initiative to solve any issues that come up, and keeping the work on track to meet project deadlines.
- Joana Gomes is the **Project Owner**, who represents the interests of the stakeholders, has knowledge of the product and the application domain and is responsible for clarifying the questions about expected product features. The Project Owner is involved in accepting the solution increments
- Mateusz Kubiak is the **Quality Assurance Engineer**, responsible, in articulation with other roles, of promoting the quality assurance practices and implementing tools to assess the deployment's quality effectively. Finally, the QA Engineer monitors that team follows agreed QA practices.
- Artur Denderski is the **DevOps master**, in charge of the development and production infrastructure and required configurations. The DevOps master guarantees that the development framework works properly. Leads the preparation of the deployment machine(s)/containers, git repository, cloud infrastructure, databases operations, etc.

1.2 Agile backlog management and work assignment

In the Agile methodology, project management revolves around iterative development, emphasizing flexibility and adaptability to change. Here's how Agile practices are implemented in the project, leveraging JIRA as a tool:

- **Epics and User Stories:**
  - Epics represent large, high-level features or components of the project.
  - User stories break down epics into smaller, manageable units of work from an end-user perspective.
  - JIRA is used to create and manage epics and user stories, allowing for prioritization and tracking.
- **Backlog Management:**
  - The backlog is a prioritized list of epics and user stories representing the work to be done.
  - The Product Owner continuously refines and prioritizes the backlog based on business value and stakeholder feedback.
  - JIRA facilitates backlog management by providing features for creating, organizing, and prioritizing backlog items.
- **Sprint Planning and Execution:**
  - Sprints are time-boxed iterations, where the team works on a set of user stories.
  - During the sprint, developers work on implementing the selected user stories, updating progress in JIRA.

1. **Conceptualization Phase:**
   - ➔ During this phase, the focus is on defining the project's overarching goals and requirements.
   - ➔ Epics are identified to represent major features or components of the project.
   - ➔ In JIRA, epics are created to capture these high-level requirements, and initial user stories may be outlined.
2. **Elaboration Phase:**
   - ➔ In this phase, epics identified in the conceptualization phase are broken down into detailed user stories.
   - ➔ User stories are refined with acceptance criteria, specifying the conditions that must be met for the story to be considered complete.
   - ➔ The Product Owner prioritizes and refines the backlog based on business value and stakeholder input.
   - ➔ Tasks are created and assigned within user stories to prepare for development.
   - ➔ JIRA is used to manage the backlog, refine user stories, and plan the upcoming sprint.
3. **Development Phase:**
   - ➔ During this phase, the development team works on implementing the selected user stories.
   - ➔ Developers update the status of their tasks in JIRA, providing visibility into sprint progress.
   - ➔ At the end of the sprint, a sprint review is conducted to demonstrate completed work to stakeholders, and a retrospective is held to reflect on team performance and identify areas for improvement.
   - ➔ JIRA is used to track progress, capture feedback, and plan for the next sprint.

2 Code quality management

2.1 Guidelines for contributors (coding style)

For coding, we followed standard Java best practices: camelCase for variable and method names, and UPPER_CASE for constants.

We aimed to keep our code clean and well-commented to improve readability and quality.

Tests were named using a specific pattern to clarify their purpose:

when<condition>_then<expected-result>

This naming convention made it easy to see what each test was checking.

2.2 Code quality metrics and dashboards

To ensure high standards of code quality, we utilize various metrics and dashboards to monitor and evaluate our codebase.

We measure the percentage of code covered by automated tests to ensure that our tests are comprehensive. Higher code coverage indicates a greater likelihood that the code has been thoroughly tested.

We track metrics to identify overly complex code that may be difficult to maintain or prone to errors. Simplifying complex code enhances maintainability and reduces the risk of bugs.

Pull requests had to be reviewed by at least one other developer.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

To make sure that work was assigned and completed in an organized and efficient manner, we used Jira as our project management tool. Jira helped us adhere to Agile methodology by providing a platform for tracking tasks, assigning responsibilities, and monitoring progress.

We divided the project into Epics, which were further broken down into User Stories. Each User Story was assigned to a developer and included in a weekly sprint.

We used GitHub as our code repository for easy code collaboration and version control. When changes were stable, tested, and documented, we created Pull Requests to get feedback and approval from the team before merging them into the main branch. At least one team member had to approve a merge to maintain the quality and cleanliness of the main codebase.

3.2 CI/CD pipeline and tools

We manually reviewed and tested the code before merging it into the main branch. Each team member was responsible for running tests locally and ensuring that their code met the project's quality standards. Peer reviews and code reviews were conducted to catch any issues early and maintain code quality.

For continuous delivery, our plan was to deploy our applications using a virtual machine (VM) provided for the project. The intended deployment process included managing the deployment through custom scripts. These scripts were designed to:

- Transfer the repository contents to the VM.
- Build the Docker image based on the Dockerfile in the repository.
- Deploy the container on the VM.

However, due to unforeseen circumstances, we were unable to utilize the VM for deployment. Instead, we continued to deploy our applications manually, which involved transferring files, building images, and starting containers without automated scripts. This manual process, while functional, highlighted the need for automation to reduce errors and improve deployment efficiency.

4 Software testing

4.1 Overall strategy for testing

Our overall strategy for testing aimed to ensure high-quality software through a blend of methodologies and tools. While we incorporated some elements of Test-Driven Development (TDD), our primary focus was on leveraging a mix of Behavior-Driven Development (BDD) and traditional testing practices to achieve comprehensive test coverage. While the ideal was to write tests before developing the corresponding code, practical constraints sometimes led to adjustments in our workflow.

We utilized Mockito for mocking dependencies and Spring Boot's testing framework to facilitate seamless integration testing. This combination allowed us to effectively test various aspects of our application, from unit tests to integration and functional tests.

Testing was emphasized as a critical component throughout the development process. All test results were reviewed and considered during the integration and continuous deployment (IC) process to maintain code quality and system reliability.

4.2 Functional testing/acceptance

Our project policy for functional testing was to write tests from a user perspective, focusing on how the system behaves under various conditions. These tests, often referred to as "closed box" tests, were designed to validate that the system meets the specified requirements.

4.3 Unit tests

Unit tests were a cornerstone of our testing strategy, aimed at verifying the functionality of individual components (methods, classes) from a developer's perspective.

4.4 System and integration testing

Integration and system tests were crucial for verifying that different parts of the application worked together as intended.
API testing was an integral part of our testing strategy to ensure that our application's endpoints functioned correctly and efficiently.

By adopting these testing practices and leveraging appropriate tools, we aimed to ensure that our application met high standards of quality and reliability. All test results were systematically reviewed and incorporated into our CI/CD process to facilitate continuous improvement and deployment.