deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Bernardo Pinto 105926, Filipe Obrist 107471, Liliana Ribeiro 108713, Lia Cardoso 107548
v2024-06-03

1 Project management

1.1 Team and roles

Every member plays a dual role: in addition to their specialized responsibilities, all contribute as developers to the solution. The following outlines our team structure, roles, and specific responsibilities.

Team Members and Assigned Roles

Team Coordinator: Liliana Ribeiro
Product Owner: Lia Cardoso
QA Engineer: Filipe Obrist
DevOps Master: Bernardo Pinto
Detailed Role Descriptions

Team Coordinator (Team Leader) - Liliana Ribeiro:

Responsibilities:
Ensure equitable task distribution and adherence to the project plan.
Foster collaboration and proactively address potential team issues.
Guarantee timely delivery of project outcomes.

Product Owner - Lia Cardoso:

Responsibilities:
Represent stakeholder interests.
Provide a comprehensive understanding of the product and domain.
Clarify expected features and requirements to the team.
Participate in the acceptance of solution increments.

QA Engineer - Filipe Obrist:

Responsibilities:
Promote quality assurance practices and implement quality measurement tools.
Ensure the team adheres to agreed QA practices.
Collaborate with other roles to guarantee quality in deployment.

DevOps Master - Bernardo Pinto:

Responsibilities:
Manage development and production infrastructure and configurations.
Ensure the development framework operates smoothly.
Lead the setup of deployment environments (machines/containers), git repositories, cloud infrastructure, databases, etc.

Developer (All Members):

Responsibilities:
Contribute to development tasks, as evidenced by pull requests/commits in the team repository.
Participate in feature implementation, bug fixing, and code review.

1.2 Agile backlog management and work assignment

In our project, we implemented Agile practices to manage the backlog and assign work efficiently. Our approach revolves around user stories and is structured according to sprints, which were defined to align with the academic discipline requirements.

For backlog management, we used Jira, dividing the project into five distinct sprints. The first two sprints focused on conceptualization and elaboration, while the remaining three were dedicated to development.

Sprint 1: Conceptualization

The primary goals during this phase included setting up the development environment, organizing the initial backlog, and defining the personas, user stories, scenarios, and epics that guide the subsequent stages of the project.

Sprint 2: Elaboration

This sprint aimed to establish the foundation for development. We defined the CI pipeline, outlined the SQE strategies, prepared a detailed specifications report, and created mockups using React.

Sprints 3-5: Development

During the three development sprints, we focused on implementing user stories based on priority, with the core user stories tackled first.

For the **work assignments**, for each sprint we first assign specialized responsibilities to each team member according to their role. After that, we distributed the development tasks evenly among everyone to ensure a fair workload.

2 Code quality management

2.1 Guidelines for contributors (coding style)

For the coding style of our project, we adhered to the AOS rules for Java. Here's a brief summary of the conventions used:

Naming Conventions:

Variables and Methods: camelCase (e.g., calculateTotal(), orderAmount).
Classes and Interfaces: PascalCase (e.g., OrderManager, ProductController).
Constants: All uppercase with underscores (e.g., MAX_CONNECTIONS, API_ENDPOINT).

Indentation and Spacing:

Indentation: Four spaces per indentation level; no tabs.
Braces: Opening braces on the same line as control statements or method declarations, closing braces on a new line.
Spacing: Single space around operators, no spaces inside parentheses.
Documentation:
JavaDocs: Detailed JavaDocs for all classes and methods.
Example (Method):

```
/**
* Processes the order with the given order ID.
*
* @param orderId The ID of the order to process.
* @return True if the order was processed successfully, False otherwise.
*/
public boolean processOrder(int orderId) {
    // Implementation
    return true;
}
```

Code Structure:

Imports: Grouped by standard library, third-party, and local application imports.
Methods and Classes: Focus methods on a single responsibility and organize them logically.
Error Handling:
Use specific exceptions with meaningful error messages.
Example:

```
try {
   processOrder(orderId);
} catch (OrderNotFoundException e) {
   logger.error("Order not found: " + e.getMessage());
} catch (InvalidOrderException e) {
   logger.warn("Invalid order: " + e.getMessage());
}
```

Testing:

Test Naming: Descriptive names reflecting tested functionality (e.g., testOrderProcessingWithValidData).
Test Coverage: Minimum of 80% coverage, with unit and integration tests.

2.2 Code quality metrics and dashboards

Throughout the project, we used SonarQube for static code analysis and monitoring to guarantee good code quality. With the integration of SonarQube into our continuous integration pipeline, every commit and pull request automatically analyzes our code. The main goals of this analysis were to find bugs, technical debt, code smells, and security flaws.

The quality gates were pre-established to set minimal requirements that had to be met before merging any changes. In order to guarantee comprehensive testing and enhance code stability, we first established a code coverage criterion of at least 80%. Furthermore, we required that no critical or blocker bugs were present to prevent high-impact errors.

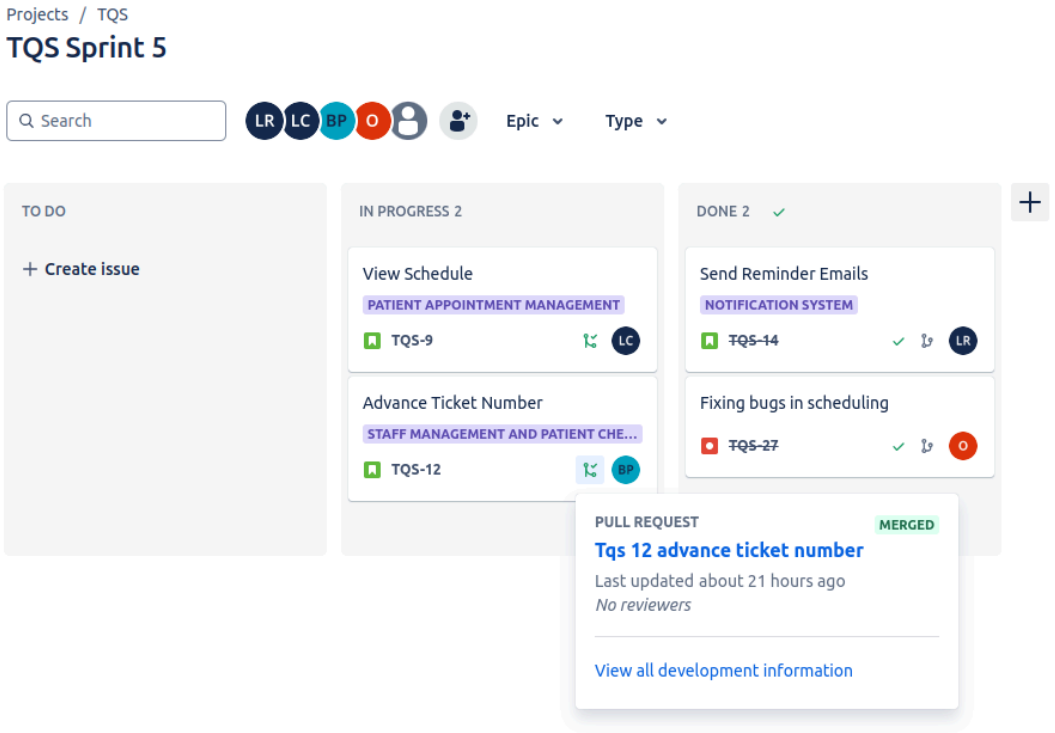
We defined that it was necessary for code smells to keep their 'B' grade, or better yet, to lower technical debt by encouraging readable, well-written code, also guaranteed by the AOS code style. A restriction of less than 5% was placed on the technical debt ratio and a limit on duplicated code, requiring it to be fewer than 3% of total lines. Also the Reliability and Maintainability measures are defined to be ‘A’ score.

The implementation of these quality gates was intended to uphold uniform standards, minimize technical debt, improve security, and encourage thorough test coverage.

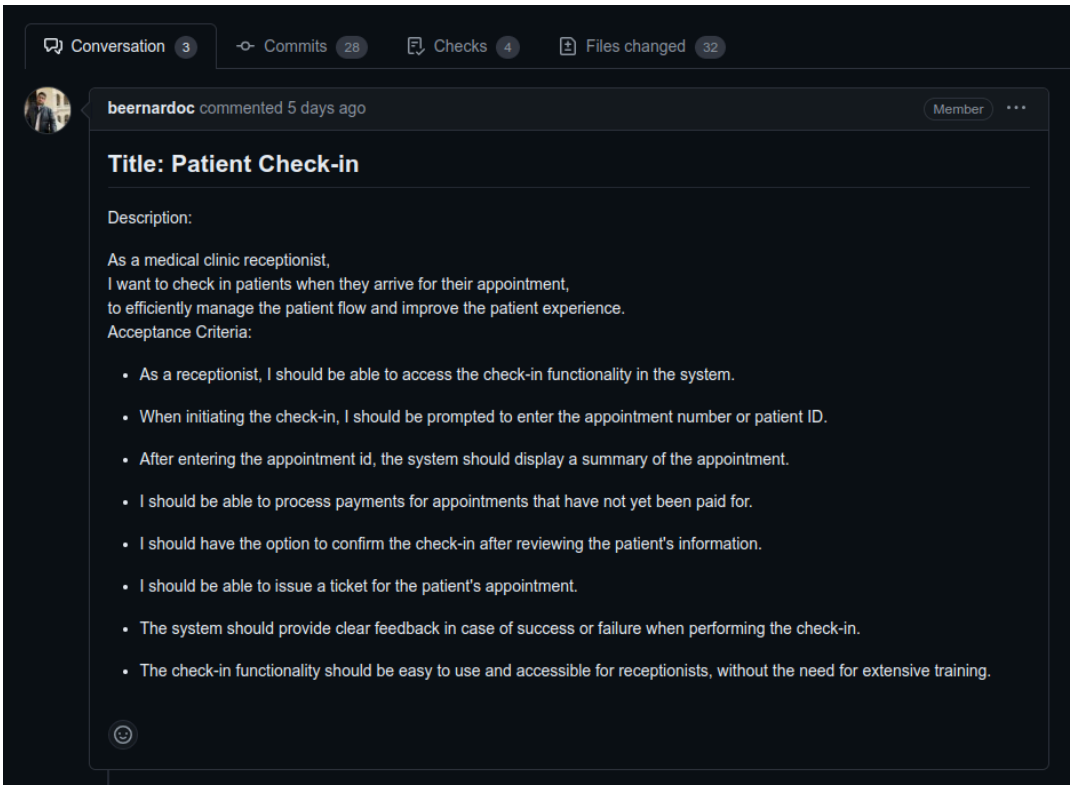
3 Continuous delivery pipeline (CI/CD)

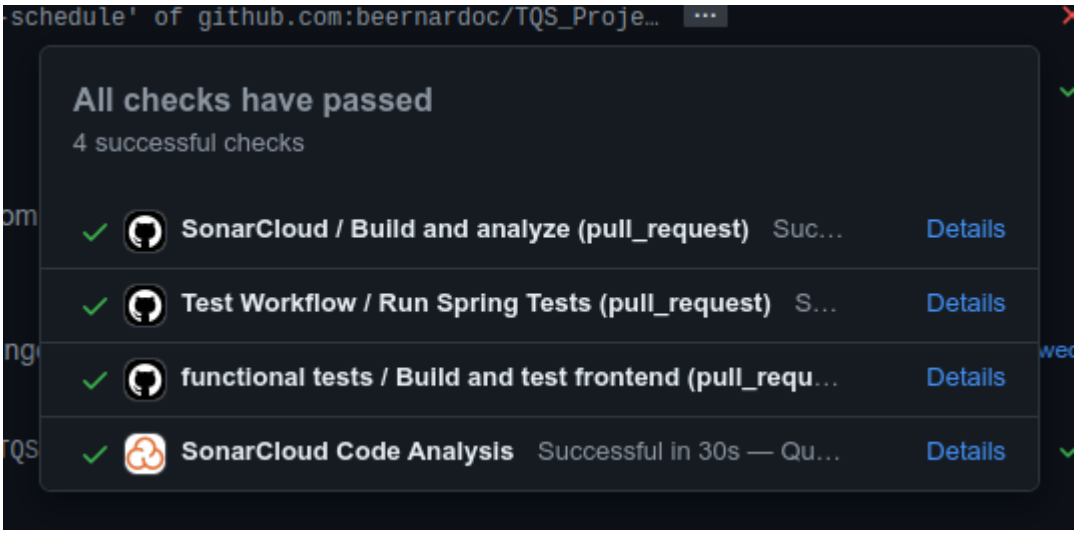
3.1 Development workflow

First, the developer retrieves the relevant branch code in Jira, creating a branch from the dev branch.

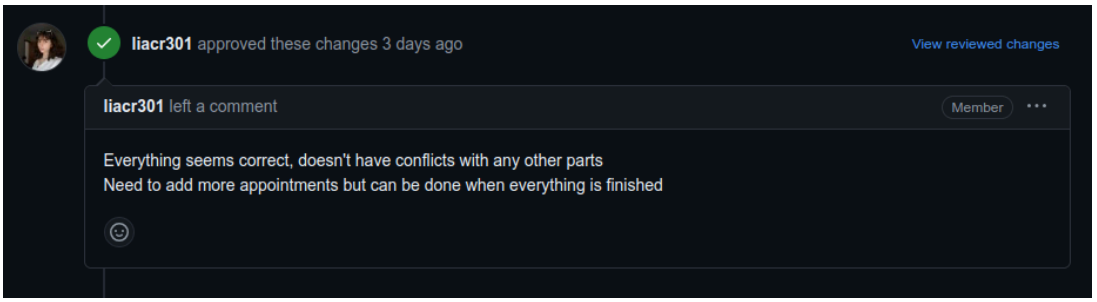


The developer works on the assigned task, staying updated with the latest changes in the dev branch. It's recommended for the developer to make regular commits to keep the work progress online. Once the task is complete, the developer submits a pull request and waits for feedback from the continuous integration (CI) system, resolving any issues identified by the CI.





After all tests pass in the pipeline, the developer requests a review from other developers, who will test, provide feedback, identify issues, or approve the pull request.



The developer then addresses any issues raised by the reviewers. Finally, when the quality is approved by all, the QA Engineer merges the pull request with the dev branch. The user stories are considered done when the branch is merged into the dev branch.

3.2 CI/CD pipeline and tools

In the project, we use a Continuous Integration (CI) approach to ensure that each code increment is automatically tested and analyzed. Continuous integration is essential to quickly detect errors and maintain code quality.

Configuration and Tools

SonarCloud (build.yml):

- **Triggers:** The SonarCloud pipeline is triggered when there is a push to the main branch or when a pull request is opened, synchronized, or reopened.
- **Environment Setup:** We use an Ubuntu virtual machine (ubuntu-latest). The code is checked out and cloned on the machine using the actions/checkout@v3 action.
- **JDK Setup:** JDK 17 is configured using actions/setup-java@v3 with the zulu distribution.
- **Cache:** We implement caching for SonarCloud and Maven packages to optimize execution time:
  SonarCloud Cache: ~/.sonar/cache; **Maven Cache:** ~/.m2, with the key based on the hash of the pom.xml files.
- **Build and Analyze:** In the final step, the code is built and analyzed using Maven, running tests and code analysis with the Sonar Maven plugin.

Maven Tests (mavenTest.yaml):

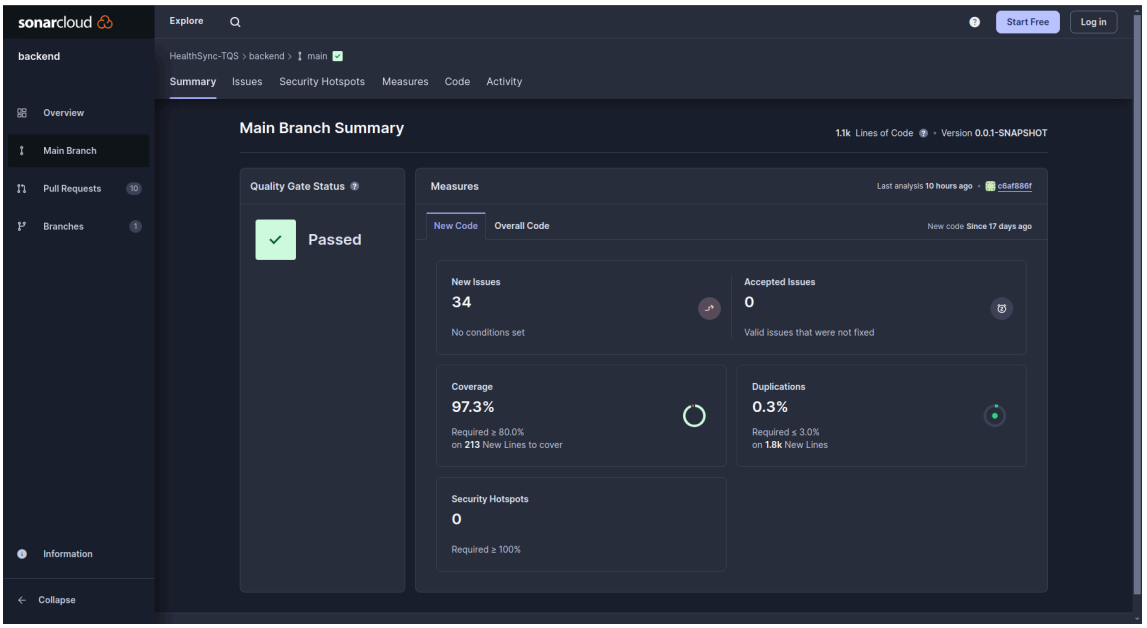
- **Triggers:** This pipeline is triggered when there is a push to the dev branch or when a pull request is made to the dev branch.
- **Environment Setup:** Similar to SonarCloud, the Ubuntu virtual machine (ubuntu-latest) is used, and the code is checked out with actions/checkout@v2.
- **JDK Setup:** JDK 17 is configured using actions/setup-java@v2 with the adopt distribution.
- **Build and Tests:** The backend code is built and tests are executed using Maven (mvn -B clean test).

For continuous delivery, we use Docker containers to ensure consistent and reproducible environments, simplifying deployment across different stages (development, testing, production). The use of containers allows for greater agility and flexibility in software distribution, ensuring the application runs consistently in any environment where the container is executed.

- We define a custom **network** (tqs_project_network) to facilitate communication between the services.
- The **MySQL database** service is set up with essential environment variables for database initialization and user credentials. Health checks are implemented to ensure the database service is ready before dependent services start.
- The **backend service** has its own build context and depends on the database and is configured with the necessary environment variables for database connection and application settings.
- We defined three **frontend services (staff, board, and customer)**, each with its own build context. These services depend on the backend service and are configured to ensure proper communication with it.

3.3 System observability

We used SonarCloud for system observability to maintain high code quality through detailed static analysis. It allowed us to quickly identify and fix bugs, security vulnerabilities, and maintenance issues, seamlessly integrating into our CI/CD pipeline for automated checks.
https://sonarcloud.io/summary/new_code?id=HealthSync-TQS_TQS_Project



4 Software testing

4.1 Overall strategy for testing

Our overall strategy centered on ensuring comprehensive test coverage through a combination of different testing methodologies and tools. We implemented the following practices:

We adopted **test-driven development (TDD)** as the main methodology for our development process. This method made sure that tests were developed before the real code, which resulted in a codebase that was more reliable and thoroughly tested.
Behavior-Driven Development (BDD): We used Cucumber in conjunction with BDD to develop acceptance tests for the user interface.
Combination of Tools: For BDD, we intended to use Cucumber, while for API testing, we would use REST-Assured.
Continuous Integration (CI): To automatically perform tests on each project, we integrated our testing strategy into the CI pipeline.

4.2 Functional testing/acceptance

For functional testing, we adopted a **closed box testing** perspective, meaning that tests were written without knowledge of the backend implementation, simulating a **user perspective**. We used Behavior-Driven Development (**BDD**) with **Cucumber**, employing **Gherkin syntax** to define acceptance criteria in a natural, readable format.

4.3 Unit tests

For unit tests, we focused on verifying the smallest units of code in isolation. These tests were written with full knowledge of the internal implementation, following an **open box testing** approach. The tests were created before implementing the actual code (**TDD**), starting with a skeleton structure. We used **Mockito** to mock dependencies, enabling focused testing of each unit. The unit tests included database tests, service-level tests, and API endpoint verification.

4.4 System and integration testing

System and integration tests were written using an **open box testing** approach to verify the interaction between different modules or services, including external systems. These tests aimed to replicate **real-world use cases**, ensuring that the entire system functioned as expected. We used Spring Test for seamless integration testing in the Java environment.
For API testing, we used automated testing for RESTful APIs, verifying responses and status codes. External services were mocked to isolate and thoroughly test the API endpoints themselves. REST-Assured was the primary tool used for this purpose, providing comprehensive and automated testing capabilities.