TQS: Quality Assurance manual

Bruno Páscoa [107418], David Cobileac [102409], Guilherme Lopes [103896]
V2024-05-25

Índice

1 Project management

1.1 Team and roles

- Bruno Páscoa: QA Engineer, Product Owner. Coordinated most of the decisions regarding the integration with management/CI/CD tools (Github Actions, Jira, Jenkins and Sonarcloud). Main authority regarding test development and Supabase integration.
- David Cobileac: Team coordinator. Designed and integrated the patient interface. Main authority regarding React/Frontend related problems.
- Guilherme Lopes: DevOps. Main authority regarding server development.

1.2 Agile backlog management and work assignment

In terms of backlog management. Most issues were created and assigned a user story and story points value during the first and second sprints (before any functional (non-mockup) code was developed). Issues would then be assigned from the backlog to the current sprint and divided.

The issue assignment criteria were based on the estimated time to implement each issue rather than actual dificulty.

2 Code quality management

2.1 Guidelines for contributors (coding style)

While the notion of coding style was only introduced to us after around 90% of the code was already written and was, thus, not enforced, most of us naturally tended to follow Oracle's Coding Conventions for Java[1] and, while there might have been some minute differences from developer to developer, coding styles tended to be consistent for the same developer.

2.2 Code quality metrics and dashboards

In terms of static code analysis, we decided to integrate our projects to automatically evaluate the code on Pull Requests and Merges (with main). Additionally, some of our developers opted to use the VSCode plugins Code Coverage and Sonar Lint to resolve potential issues without the need to have to constantly push changes (Sonar Lint allowed a "correct-as-you-go" approach to issue detection by highlighting issues as they were being written and Code Coverage provided a semi-accurate coverage report, devided by folders and files, after running tests, as well as highlighting covered and uncovered code).

In terms of quality gates, we considered the default quality gate from Sonarcloud to be sufficient.

The 80% coverage threshold, while hard in the 1st coding sprint (due to the amount of Entity classes that are hard to fully cover), it incentivized us to design 3-layer tests (controller, service and repository tests) while still providing some leeway for less testable segments of our code. We also kept the issues as-is, as they didn't particularly pose a problem with our own coding approaches (code duplication even avoided some boilerplate by incentivizing the use of Lombok in entity classes).

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

In terms of code review, it was agreed that any one person (besides the one who wrote that specific piece of code) could review it. The main purpose of these code reviews were to spot any potential issues Sonarcloud might have missed, as well as doing manual acceptance tests on the static code itself (that is, the code was evaluated not only on its cleanliness, but also whether or not it resolved the problem as it was supposed to).
Definition of Done:

1. Issue must have a score.
2. Issue must be properly described (preferably in the "given when then" format).
3. A separate branch related to the isssue must be opened.
4. The issue's code must be functional.
5. All unit tests must pass.
6. It must pass SonarCloud's quality gate.
7. Pull request must be created.
8. Code reviewed by, at least, 1 person aside from the people assigned to the issue.

[1] https://www.oracle.com/technetwork/java/codeconventions-150003.pdf

Note: the main "unit" of work and the one this Definition of Done is not the user story but rather the Jira's Issues, which are subdivisions of the User Stories (which are mapped as Epics).

3.2 CI/CD pipeline and tools

The CI pipeline was mostly done on Github Actions and consisted mostly of running the tests and sending the results to both Sonarcloud and Jira (the action would fail if the quality gate wasn't passed). Jira would then automatically create the relevant tests (although associating those with the issues had to be done manually). Sonarcloud also reported the results in the pull request itself and the developers were notified by email of the results.

In terms of CD, to avoid the restriction on inbound connections to the Virtual Machine, a subscription-based webhook redirector (Smee) was used and then connected with Jenkins to automatically fetch new changes, rebuild the project and then re-run it. The entire project was turned into containers orchestrated using docker compose to facilitate this deployment (the containers themselves weren't uploaded to Docker Hub as we didn't see a need for it for now).

4 Software testing

4.1 Overall strategy for testing

In terms of general testing strategy, we opted for TDD, due to its ease to set up and it allowed for a more programmer-centric approach.
However, due to time constraints, automated Integrations tests are only being developed in this sprint.

4.2 Functional testing/acceptance

As the developer in charge of integrating the API was usually handled by a different developer than the one responsible for developing the API, the acceptance was done during the integration itself. These tests usually followed a closed box (where the frontend developer looked only at the Controllers and not the Services, where most of the processing was done).

4.3 Unit tests

Unit tests where the same scenario was tested across the 3 layers (Controller, Service and Repository) separately, while the remaining components were mocked as needed.
Besides those, unit tests were also designed to ensure a correct connection and response to Supabase, our authentication and database provider (and the only time, besides manual and integration tests where the Supabase server was called).

4.4 System and integration testing

In terms of integration testing, we opted for a closed box, user facing tests by testing 2 main usage scenarios using Selenium Webdriver: checking future consultations and scheduling a new consultation (both as a patient).

4.5 Performance testing

In terms of performance testing, we mainly aimed to test the effect of a medium-to-large load on response times. As such, we used Jmeter to test 1000 lightweight GET requests (obtain all consultations) and 2000 POST request (schedule a new consultation), which are more computationally expensive.

In terms of results, the GET operation had a 9.5% fail rate while the POST requests had a 20.6% fail rate, which indicates that we need to improve the systems capabilities to handle larger concurrent loads (assuming the issue is not with the limitations of the virtual machine itself).