TQS: Quality Assurance manual

Patrícia Dias [98546], Pedro Lopes [97827], Raquel Ferreira [98323], Sophie Pousinho [97814]

V2022-06-23

1 Project management

1.1 Team and roles

The roles assigned in the team are the following:

- Team Leader - Sophie Pousinho
- Product Owner - Patrícia Dias
- QA Engineer - Raquel Ferreira
- DevOps Master - Pedro Lopes
- Developer - all

Team Leader

Distributes in a fair way the work by the team elements and ensures that all members are working according to plan. Resolves conflicts that may appear between people. Make sure that the deadlines and deliveries are fulfilled.

Product Owner

Ensures that stakeholder interests are met. Is involved in the new increments to make sure the functionalities are as intended. Answers questions about features that the development may have.

QA Engineer

Ensures, in articulation with other roles, the quality of the product by promoting QA practices and their use by the team. Puts in practice instruments to measure the quality of the deployment.

DevOps Master

In charge of the deployment and production infrastructure and its required configurations. Starts the preparation of the deployment machines/containers, create and prepare the code repository so that the team has access and can start developing, cloud infrastructure, and database operations, among others.

1.2 Agile backlog management and work assignment

The agile tool used by the group was Jira which allows linking a Source Code Management tool like Github. Before starting developing, the team defined all the epics to be done during the development. This allowed us to define the flow of every sprint in the following way:

- The epics were reviewed and the user stories (issues in Jira) were added to each of them
- In every user story were also defined tasks
- Other tasks that had to be done during the sprint were added to the backlog so that the group did not forget what was necessary to do
- The tasks were attributed to the elements according to the estimation given to them by the team
- To work on a task, a branch was opened with its name
- When a task was considered done by the element that was responsible for it, a pull request was done and only after at least 2 members of the team reviewed it the task was merged to the dev branch

The template used for the user stories was the following: **As a <user>, I want <a goal>, so that <benefit>**.

The estimation scale used goes from 1 to 5 where 5 is the more complex and/or time consuming and 1 is the simpler and/or quicker to do.

In case some element completed their tasks sooner than expected, some of the other tasks were reassigned to them.

To help the elements of the team maintain the status of development updated, some automation rules were added to Jira:

- When a branch is created, then move issue to in progress
- When a pull request is merged, then move issue to done

Every task had its own branch and whenever the issue was complete, the branch was closed. All the branches' names and commits done had to follow the rule defined by Jira, where they needed to start with the key of the task. In case something was wrong with the tasks, another branch (**fix/<problem>**) was opened to resolve the problem. If there was a need for improving/adding tests, refactoring code or others a new branch would be created (**imp/<test-something>**).

2 Code quality management

2.1 Guidelines for contributors (coding style)

The guidelines used for the database were the standards of PostgreSQL, namely, using snake_case for identifiers (names of databases, tables, columns, etc) and the SQL keywords in UPPER CASE.

```
UPDATE my_table SET name = 5;
```

Rules followed specifically for ReactJs:

- Always use double quotes (") for JSX attributes, but single quotes (') for all other JS
- Always include a single space in self-closing tags
- Wrap JSX tags in parentheses when they span more than one line
- Always self-close tags that have no children
- Define always a class name or/and id to make functional testing easier
- If your component has multi-line properties, close its tag on a new line
- Remove all console.log()

In all other cases, the rules followed were the following:

- Do not ignore exceptions (only when sure that is not a problem is when exceptions can be ignored) and use meaningful exceptions
- Fully qualify imports
- Use camelCase naming variable convention
- Follow field naming conventions (Static final fields (constants, deeply immutable) are ALL_CAPS_WITH_UNDERSCORES)
- Use standard brace style
- Treat acronyms as words
- Use logs not prints
- Keep methods/functions short (Split your code into multiple smaller functions)
- When in doubt, be consistent with the surrounding code.

2.2 Code quality metrics

For static code analysis, the tool used was SonarCloud due to the fact that it helps maintain code quality and security by detecting code smells and bugs. Since SonarCloud works by project, there were 2 projects in SonarCloud, one for the specific part and another for the generic one.

The analysis of SonarCloud was triggered whenever a push in the repository was done to the dev and main branches. This allowed the team to make sure that both the developed code and the final version of the code maintained the quality defined.

The quality gates used were the defined ones by SonarCloud since they are acceptable to guarantee that every code change committed to the project is free of issues and vulnerabilities. The only case we dismissed the SonarCloud analysis was when there was a Cross Origin hotspot since it was not relevant for the project.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

As explained before, every user story was divided into tasks that were distributed by the team members. Each task had its own branch whose name was the task's name. So, whenever we looked at a branch name, we knew what was being done.

When a task was complete, the developer would make a pull request to the dev branch which had to be reviewed by at least two other members. The reviewers could approve it, request changes, or leave comments for improvement. After the reviews, the DevOps master merged the code into the dev branch.

At the end of a sprint, a pull request was made from the dev branch to the main branch (same rules as the pull request into dev) and the code was merged.

To ensure that the code was reviewed, a rule was defined in the repository.

A user story was considered done when:

- The Product Owner approved the user story
- The unit tests were written, executed and passed
- The user story had been peer-reviewed
- Integration testing performed and compiled
- The code refactoring was complete
- Produced code for presumed functionalities
- Assumptions of User Story met
- Project could build without errors
- User story was tested against acceptance criteria

3.2 CI/CD pipeline and tools

The continuous integration pipeline was implemented with Github Actions since it makes it easy to automate the workflows.

We defined two workflows, one for the generic part and another one for the specific. In each one, two dockers were run, one with the Spring Boot part and another with the frontend and the IP address, because it was not always the same. This allowed the *mvn verify* command part to execute the functional tests with Cucumber as well as the unit and integration tests.

The workflows were integrated with SonarCloud, which gave us a code analysis every time something was pushed or pull requested to the dev or main branches. If the code analysis and/or the tests failed, the code could not be merged with the corresponding branch.

Both projects also include a CD Pipeline, this pipeline makes use of Github Actions and self-hosted runners. This pipeline includes two workflows, one to push the docker images to the container registry at https://github.com/sophjane?tab=packages&repo_name=tqs-delivery-p1g2, this workflow is ran in a github runner.

The other workflow is ran in the virtual machine provided. This last workflow ensures that the volumes used are preserved as well as they keep the down time to a minimum.

4 Software testing

4.1 Overall strategy for testing

The overall development strategy followed was Test Driven Development (TDD), since this strategy can be beneficial to the developers by decreasing the debug time, bugs and errors. The main tools and frameworks used were Junit5 and Mockito. Also, the names of the tests had to be descriptive, identifying what should be returned and what functionality was being tested, e.g., *whenValidInput_thenCreateRider* and *testGetAllOrdersByInvalidUser_ThenReturnResourceNotFound*.

For the functional tests, we used a BDD strategy with Selenium and Cucumber, creating the tests after the backend and frontend of a user story were connected.

4.2 Functional testing/acceptance

For the functional testing, we decided to use Cucumber combined with Selenium on a headless browser webdriver, by using a headless webdriver we allowed the tests to be run in an environment that was not capable of graphical rendering such as GitHub Runners, which were used to build a CI pipeline. We used a Page Object Pattern with the Selenium tests to make them clearer and easier to read. These tests were developed with the user stories in mind, therefore a user story was always reflected in a Gherkin Feature.

These tests were created after the frontend and backend were connected, starting with the Cucumber feature and then the pages class and steps part of the Page Object Pattern.

4.3 Unit tests

Unit tests were developed to test the behavior of the repositories, controllers, services, and any other classes eventually created. Not only the “normal” scenarios were tested, but also some scenarios of errors.

To make the tests faster, we limited the context of some of them. For the repository tests, we used *@DataJpaTest* annotation to load only the *@Repository* spring components, and for the controller test we used *@WebMvcTest* to test the APIs (*@Controller* annotation). In the case of the controllers, we had to mock the beans used by the controllers with *@MockBean*.

For the repository tests, we also used @Testcontainers, which allowed us to test the database engine we were using for the project, using a TestEntityManager to access the database.

In the services tests, we used Mockito to mock the repositories' behavior, making it independent of the persistence layer.

Finally, in the controllers' tests, we mocked the services used by the controllers with @MockBean. In the generic service, we used MockMvc and in the specific one, we used RestAssuredMockMvc.

4.4 System and integration testing

The integration tests made sure the individual parts (repository, service, and controller) worked correctly together. These tests were created accordingly with the endpoints in the controllers. And as in the previous tests, there were tested "normal" cases and some with errors.

To do this kind of test, we needed to load the whole application context, and to do that we needed the @*SpringBootTest* annotation. We also used @*Testcontainers* for the database with the docker accordingly with the database engine we were going to use.

We used TestRestTemplate simultaneously with RestAssured.