TQS: Quality Assurance manual

Raquel Paradinha [102491], Paulo Pinto [103234], Tiago Carvalho [102142], Miguel Matos [103341]

1 Project management

1.1 Team and roles

Our team is comprised of four members. All of us contributed as developers for the final product, with an adittional role for each one.

NMEC	Name	Role
102491	Raquel Paradinha	DevOps Master
103234	Paulo Pinto	Quality Assurance Engineer
103341	Miguel Matos	Team Manager
104142	Tiago Carvalho	Product Owner

1.2 Agile backlog management and work assignment

The group had a meeting in the beginning of each weekly iteration, where the Team Manager assigned tasks to be completed during the week. In order to support this dynamic, we made use of Jira as our backlog manager, which allowed us to better organize the team's workload and keep record of each iteration's progress.
Work was split into User Stories, each one belonging to an Epic, and each story was assigned to one or two developers, in order to be tested, implemented and documented.
At the end of each iteration, the team assessed the week's results, so as to better plan the next week.

Code quality management

1.3 Guidelines for contributors (coding style)

For code writing, we followed the most common Java good practices, like using camelCasing for variable and method names and constant names being UPPERCASE.

The team was encouraged to keep their code clean and commented to increase readability and overall code quality.
Tests were named using a specific structure that aimed to help make clear the goal the test meant to acomplish:

when<condition>_then<expected-result>

which meant we could easily tell by the method name what that piece of code was meant to check and do.

1.4 Code quality metrics

For new code, we required at least a 50% code coverage. On overall code quality, we enforced at least 80%, as to test our solution in a more robust way.
Also, before accepted, pull requests had to be reviewed by at least one other developer.

2 Continuous delivery pipeline (CI/CD)

2.1 Development workflow

To ensure work is being done and assigned in organized and efficient ways, the team made use of the Jira platform to help us stick to the Agile methodology. It served as our go-to project management tool, and provided us with a centralized hub to track tasks, assign responsabilities and monitor our progress. The whole project was split into *Epics*, which were in turn split into User Stories, each belonging to it's own weekly sprint and assigned to a developer.
We used GitHub as our code repository, allowing for easy code collaboration and version control. We used GitHub and Jira side by side, creating a branch for each User Story we meant to develop, and creating Pull Requests when changes were stable, tested and documented, aiming to receive team feedback and approval to merge the new changes into the main branch. We required at least one team member to give merge approval before allowing code to enter the main branch, in order to keep the final code as clean as possible.

2.2 CI/CD pipeline and tools

The CI pipeline was implemented using SonarQube and GitHub actions. When a push ocurs on a main branch, or a pull request is made, this action is triggered, which leads to the the solution being automatically tested. If any tests fail, the action does too and the user is notified. This was implemented to ensure that the code that reaches the repository meets specific guidelines, like code coverage in terms of tests or code clean of code smells.

For the CD pipeline, we first tried to resort to Google's Cloud Run. However, we were having some issues with it, so decided to resort to the VM that was made available to us in the context of the project. A CD pipeline was implemented using GitHub actions. This pipeline, when triggered, would send the contents of the repository over to the VM, build the docker image associated to the Dockerfile in the repo, and deploy the container on the VM, without any access from the user needed.

3 Software testing

3.1 Overall strategy for testing

Our strategy was based on TDD for the Take it Easy backend, we started by thinking about how we wanted it to work and did tests to ensure that it would work that way, then we developed the software accordingly. For the frontend, we used BDD and cucumber since it made more sense in a website environment to use BDD and cucumber made the implementation easier to understand.

3.2 Functional testing/acceptance

In our software development process, we have utilized open-box (white-box) testing extensively for our functional testing efforts. Since we developed the frontend in-house, we had deep insight into the internal code, architecture, and design. This enabled us to conduct thorough open-box testing by designing test cases that specifically targeted the individual components, functions, and integration points within the frontend. By leveraging our knowledge of the internal workings of the application, we were able to test our use cases and validate the correctness of our frontend implementation. Open-box testing allowed us to identify and address defects, logic errors, and issues in the codebase at an early stage, ensuring a high level of quality and reliability in our software. This approach helped us in refining the frontend functionalities, optimize performance, and deliver a robust and user-friendly application to our customers.

3.3 Unit tests

In our unit testing approach for the backend, we heavily emphasized the developer perspective. Our development team took a code-centric approach to design and execute comprehensive unit tests. With the idea we had of how we wanted our backend to work, we wrote targeted tests that covered individual units of our use cases. The developer perspective allowed us to catch and address defects early in the development cycle, ensuring a robust and reliable backend. Regular unit testing cycles provided prompt feedback and facilitated quick bug resolution, leading to improved code quality and stability.

3.4 System and integration testing

In our approach to the system and integration testing, we heavily relied on the developer perspective to ensure a comprehensive evaluation of the software. By leveraging our deep understanding of the system's internal architecture and design, we designed and executed tests that assessed the integration and interaction of different components. Our developers took the lead in simulating real-world scenarios and user interactions, validating the system's behavior from an external standpoint. This approach allowed us to identify any inconsistencies, defects, or issues that could impact the overall functionality and user experience. By adopting a feedback-driven development process, we ensured that any identified problems were promptly addressed and resolved. The developer perspective, combined with our adherence to test-driven development principles, played a crucial role in enhancing the reliability, stability, and overall quality of our system during integration testing.