universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Inês Leite [92928]**, **Orlando Macedo [94521]**, **Óscar Fernandez [101631]**, **Pedro Marques [92926]**, **Rui Fernandes [92952]
v2020-05-03

1 Project management

1.1 Team and roles

This project was developed by a team of 5 members, each member was a developer. Furthermore, Rui Fernades was the team coordinator, Inês Leite the product owner, Orlando Macedo and Óscar Fernandez were the quality assurance engineers and Pedro Marques was the DevOps master.

1.2 Agile backlog management and work assignment

During the development of the project, the workflow was planned by using PivotalTracker integration with github. Since this project is composed of two identifiable subprojects, there was a section in Pivotal for each of its components (WeDeliver and DrinkUp).
Each user story to implement was added to Pivotal and a branch was created for each one.

- WeDeliver - https://www.pivotaltracker.com/n/projects/2500281
- DrinkUp - https://www.pivotaltracker.com/n/projects/2500453

2 Code quality management

2.1 Guidelines for contributors (coding style)

The project is mainly written in Java, Typescript and Flutter. With that in mind, it is important to have a well defined coding style nomenclature that allows easier reading and mitigation of errors. The coding styles adopted will be shown next.

1. Java code style - Android Open Source Project - https://source.android.com/setup/contribute/code-style
2. Typescript code style - Angular coding style - https://angular.io/guide/styleguide
3. Flutter code style - Style Guide for Flutter Repo - https://github.com/flutter/flutter/wiki/Style-guide-for-Flutter-repo

Furthermore, the mobile flutter app was developed using the BlocPattern.

2.2 Code quality metrics

Code quality is achieved using the integration of the github repositories with sonarcloud. The sonarcloud project can be found at https://sonarcloud.io/organizations/tqs-project-1/projects.

The quality gate used as the following requirements:

1. On new code
   a. Coverage - at least 65%
   b. Duplicated Lines - no more than 3%
   c. Maintainability Rating - A
   d. Bugs - no more than 0
   e. Critical Issues - no more than 1
   f. Major Issues - no more than 2
   g. Reliability Rating - A
   h. Security Hotspots Reviewed - 100%
   i. Security Rating - A
2. On overall code
   a. Coverage - at least 70%
   b. Duplicated Lines - no more than 5%
   c. Maintainability Rating - A
   d. Reliability Rating - A

e. Security Rating - A
f. Uni Test Success - 100%

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

For code development the github workflow was used. To that end, all development was committed to the dev branch. Each new feature had to necessarily take the code that existed in the dev branch at that time. When all development was complete, a pull request was made to the release branch. At this time the application was released with version v1.0.0. Finally a merge with main, in order to update the main branch of the repository in conformity with the released branch.

Naming branches

The development and bug branches had to follow a specific nomenclature of names. This is due to the use of PivotalTracker to manage the backlog and the use of Github Actions for CI/CD.

- Feature branches: <UserStoryID>-feature-<BranchName>
- Bug branches: <UserStoryID>-bug-<BranchName>

Using this nomenclature it's possible for PivotalTracker to automate the beginning and delivery processes of User stories. This way of creating branches also allows for a better framing of the branches that must be the target of integration (CI) and deployment (CD) tests.

Code review

To submit a new feature it was necessary to create a pull request from the branch where the feature was developed for the dev branch. This pull request should be well documented. Merging with the dev branch would only take place if the new code successfully passed all CI rules. Not only that, the code should also be reviewed by at least one other project member. Being merged if all these conditions were met.

When the functionality branch is merged with the dev branch, the User Story is considered delivered.

3.2 CI/CD pipeline and tools

With regard to CI/CD workflows, the project is using for that purpose the **Github Actions**. This feature of Github allows the implementation of a CI/CD easily. The links to each file are located in the readme of the documentation repository.
As it’s shown on the file, the first tests to be made are the unit tests and integration tests. They are made in different phases. Firstly the **Unit Tests** and after they all pass, the **Integration tests** are executed. At the same time, the **Sonarcloud** plugin is registering static code analytics, like code coverage (with the help of **Jacoco** plugin) and at the end of the maven verification the data collected is sent to the **Sonarcloud** project.
Lastly, if the new code passes all the tests, the application is deployed on **Google Cloud**. The deployed application can be found at https://webmarket-314811.oa.r.appspot.com.

One last note, to point that this workflow runs every time that happens a push to one of the following branches:

1. Main
2. Release
3. Dev
4. **-feature-**
5. **-bug-**

The workflow also executes when a pull request is opened or reopened. And at last, if the pull request receives some new update (commit to the branch being merged).
When it comes to the mobile app, a CI pipeline was also developed that would execute static code analysis and the developed tests.

4 Software testing

4.1 Overall strategy for testing

With regard to the tests themselves, they were divided into several levels, depending also on the aspects that had to be tested. These levels are listed below.

1. At the data model level, we used tests with @DataJpaTest that allow the tests of custom queries and the integrity of the data model in general
2. At the service level, simple unit tests, mocking all the dependencies. This permits the verification of all the logic associated with the specific service.
3. At the rest controller level, first using @WebMvcTest. This tool was used with the purpose of mocking service dependencies that allows the verification of the controller logic. This tool also permits the verification of the endpoint parsing.
4. Again at the rest controller level, using @SpringBootTest and @TestPropertySource to test the endpoints with all the associated logic. The database used was also the database for production.
5. Lastly, all data transfer objects and util methods were tested to mitigate unnecessary errors.

4.2 Functional testing/acceptance

Functional testing must be done using the black box methodology. This means that the functionalities must receive a certain input and its output must be tested in order to verify if it corresponds to what was expected or not.

A very good explanation of this type of test can be found at
https://www.guru99.com/functional-testing.html.

4.3 Unit tests

Unit tests must follow an open box methodology where all intrinsic logic is tested. It is important to check for example that method calls are done correctly and in the correct order. For this, mocking modules is imperial and should be done whenever possible.

A very good explanation of this type of test can be found at
https://www.guru99.com/unit-testing-guide.html.

4.4 System and integration testing

For system and integration tests, the two test methodologies, open box and closed box, should be used. Closed box will allow you to check inputs and outputs and open box will allow you to check the solidity of the internal logic.

Another very good guide can be found at https://www.guru99.com/system-integration-testing.html.