TQS: Quality Assurance manual

João Borges [98155], Miguel Monteiro [98157], Vicente Samuel [98515], Vasco Sousa [97636]
v2020-05-03

1 Project management

1.1 Team and roles

João Borges is our team manager, he is in charge of distributing work as well as making sure everyone is doing their job. He is responsible for preparing our service frontend.
Miguel Monteiro is our QA engineer, he is in charge of keeping the project's quality at a certain level. He is responsible for preparing our service backend.
Vicente Samuel is our product owner, he is making sure we are making the product he devised. He is responsible for preparing our delivery frontend.
Vasco Sousa is our DevOps master. He is responsible for preparing our delivery engine and deploying the environments of each component, assuring CICD principles.

1.2 Agile backlog management and work assignment

We will be using Jira to help us in having a consistent workflow and helps us distribute work. Thanks to Jira's, we were able to separate our work in 3 forms: epics, which would represent each app; stories, which represente a feature in one of the apps; tasks, which represent the preparations needed for a feature to be functional.

2 Code quality management

2.1 Guidelines for contributors (coding style)

We will be using AOS/GJF since it’s a simple coding style for java application and there will be some leeway since this project must be completed as soon as possible, so having a properly tested and working application is more important than keeping a perfectly clean code. We choose this style mainly because Intellj provides an extension that can help with keeping the code in said style.

2.2 Code quality metrics

We decided to make sure that whenever a new push is made to the project, 65% of the features should be tested before the changes are accepted; To verify this we use SonarCloud. SonarCloud is also used as our continuous integration tool.
We also decide that before any merge request is accepted, it has to be reviewed by at least 2 other members of the group.
Some security issues were raised, mainly due to CORS configuration, which we ignored for development purposes.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

We will be using gitflow. This workflow relies on the usage of branches for each person to work on, and after a feature is completed a pull request is made to a development branch, and after we complete a stable software product, the changes made to this branch are pulled to the main branch. We chose to use this workflow due to being one of the best workflows for group projects.

We plan on having at least two people review each pull request made.

We believe a user story is completed when every feature related to that user story is working as intended and is well tested.

3.2 CI/CD pipeline and tools

Our CI/CD pipeline is implemented with GitHub Actions. We have various jobs defined.

First, regarding code analysis, we will be using SonarCloud as a way to prevent code that has not been properly tested to be submitted to the main branches, since it can provide quality gates checks, code coverage and check if the functions are relatively secure.
As for delivery and deployment, there are three different environments: the dev, the test and the prod environment. The development environment contains the running application with source code of the **dev** branch, the prod environment with the **main** branch, both which update automatically on branch changes, running docker containers with the aid of a self-hosted runner.
For frontend testing, we have a job that deploys a temporary testing frontend, and then, using selenium on a maven project, runs the cucumber tests pointing to that instance.

4 Software testing

4.1 Overall strategy for testing

For the backend we will be developing the features and the test together, in other words, after a feature is believed to be completed it should be properly tested to see if we are getting the expected result before we advance to the next one.
For the frontend we will be using behavior driven development and cucumber to make sure our page is not only properly connected to the api but also working as intended.

4.2 Functional testing/acceptance

While most features can and should be tested manually, there should be at least 1 simple test for each user story to make sure that if any changes are made we can immediately find out if a user story was altered in an unexpected way that could cause problems.
The framework used was a mix of cucumber with selenium.

4.3 Unit tests

For unit testing, all of the unit's functions should be tested in regards to their most common outcomes, to make sure everything was properly implemented and that no unexpected results are obtained.

4.4 System and integration testing

The most commonly used features should be tested properly to make sure they were properly implemented and can work together. This implies asserting, for example, the persistence of data on endpoint calls or the compatibility of external services (like the AddressResolver) with our own services.