universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Daniel Francisco [98188], Henrique Sousa [98324], Nuno Fahla [97631], Tiago Costa [98601]
v2022-06-23

1 Project management

1.1 Team and roles

Team Coordinator - Henrique Sousa - Distributes tasks, makes sure that members work according to plan
Product Owner - Nuno Fahla - Understands the product and application domain, answers questions from the team to clarify how the product is expected to work
QA Engineer - Tiago Costa - Promotes quality assurance practices, monitors the team to see if the QA practices are being followed
DevOps master - Daniel Francisco - Configures and creates the development and production infrastructure, prepares containers, deployment machines, git repository, database operation and others.

1.2 Agile backlog management and work assignment

The stories were written within Jira, where we can associate them with one of the developers in the project. When the developer assigned with a task started working on it, he would move the task to the in development section. When the pull request of the branch for that user story was accepted, he would move it to the done section. At the end of a spring, everything that wasn't in the done section would be added back into the next spring, as well as new user stories.

2 Code quality management

2.1 Guidelines for contributors (coding style)

Some coding rules we adopted included using TODO comments to specify what needs to be done in a section, commenting code to make it more readable, writing short method names, not ignoring exceptions or capturing generic exceptions, among others.

2.2 Code quality metrics

For a pull request to be accepted, it would need to be reviewed by at least 1 person and also go through sonarqube, with 75% test coverage, 0 bugs, vulnerabilities and hotspots as well as at most 1 hour of code smells.

75% test coverage should be enough to make sure that the parts of the code that need to be tested are tested. Bugs, vulnerabilities and security hotspots are very dangerous things to let through to the main code, so they shouldn't be accepted in pull requests. Code smells make the code less readable and makes it harder to work on the code, so each pull request should at most add 1 hour of code smells.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

We used Jira as our main tool for the workflow, which lets us map users directly to the stories we placed on our current spring.

Our definition of done is the action described in the user story can be accomplished in our system without the possibility of making big mistakes, the code must pass the quality gate and 1 person has to have reviewed the code.

3.2 CI/CD pipeline and tools

CI - On each pull request, the code is analyzed by sonarqube, to check if the quality gate is met. All of the tests are run as well to see if any of them fail. If both pass successfully, it’s then analyzed by another member of the team before accepting the request.

CD - all of our components are in docker containers, which are automatically deployed onto a VM using Jenkins.

4 Software testing

4.1 Overall strategy for testing

During development, we used a TDD approach, where we wrote the tests first, then the code to pass the tests. We used tools like Mockito, Junit 5, Cucumber, Selenium and others to write our tests.

We started by analyzing the component, based on the type of test we were going to make (user or developer perspective, depending on the type of test). From that, we wrote down a list of tests to make and proceeded to make them using the tools mentioned above.

After we wrote them, we would make the code to pass the project, check if all the tests pass and then we would also check if it passed the quality gate as well as if the tests passed on our CI pipeline.

4.2 Functional testing/acceptance

For these tests, we looked at the project from the user's perspective and we figured out how the user would interact with our application. From that perspective, we wrote actions that the user would do and we wrote functional tests for those, using Selenium and Cucumber as our main tools.

4.3 Unit tests

For unit tests, we looked at our project from the developer's perspective, figuring out which sections of our project should be tested and how they connect to other sections, so we could Mock those connections. We wrote unit tests for the sections we identified during this process, like the business logic, using Junit 5 and Mockito.

4.4 System and integration testing

For integration tests, we looked back at the connections we had on the unit tests and we looked into testing those. From this, we wrote API tests as well as database tests, using test containers, Junit and assertj.