TQS: Quality Assurance manual

André Morais [93236], Eduardo Santos [93107], Pedro Bastos [93150], Pedro Santos [93221]
v2021-06-14

1 Project management

1.1 Team and roles

<u>Team Coordinator – *Pedro Bastos*</u>
Tasked with fairly coordinating the team and fairly distributing tasks amongst its members. Taking the initiative solve problems that may appear and doing regular checkups to ensure everyone and everything is working according to plan. Must also make sure the project's expected outcomes are being delivered in time.
<u>Product Owner – *Pedro Santos*</u>
Must have a deep understanding of the problem presented, successfully representing the interests of the stakeholders. Should be able to answer questions that may arise amongst the team members relative to the expected features that the system must have. Is also involved in accepting solution increments making sure the correct functionalities are delivered and working as intended.
<u>QA Engineer – *André Morais*</u>
Responsible with ensuring the quality of the product being developed, by promoting quality assurance practices and ensuring they are being used, and by utilizing the tools available to measure the quality of the project and its increments.
<u>DevOps Master – *Eduardo Santos*</u>
In charge of the development and production infrastructure and its required configurations, ensuring it works correctly. Should also initially prepare the deployment machines/ containers ensuring they are ready for deployment, create and prepare the git repository so all members have access and can begin coding, and prepare other needed technologies such as a cloud infrastructures or database operations.

Developer – *All*

All members will contribute to the development of the project's tasks.

1.2 Agile backlog management and work assignment

For the backlog management and work assignment we'll be utilizing GitHub's Project Management features.

In the projects section we'll include 2 projects, one for the Business Initiative (Covid Tests Deliveries) and one for the Deliveries Engine, each one containing their own tasks based on user stories separated into "To Do", "In Progress" and "Done" columns.

When beginning work on a user story, a developer must first create an issue (which can represent features, bugs, and other development tasks), which will represent the story being worked on. Issues have a small title, a description detailing what needs to be done, labels to easily identify what category they belong to, a specific project which they belong to, and can be assigned to one or more team members, therefore facilitating job assignment. When all work is complete, the issue should be marked as closed.

When working on an issue, the developer responsible should make sure it's marked as "In Progress", so that other group members know that it's being worked on, when an issue is close, it's automatically moved to the "Done" column.

2 Code quality management

2.1 Guidelines for contributors (coding style)

For this project we'll be following the coding style AOSP Java Code Style.

2.2 Code quality metrics

The main tool we'll be utilizing for static code analysis will be Sonar Cloud. Due to our projects being separate, they will also be analyzed individually, meaning there will also be 2 projects on Sonar Cloud. Every push to a branch and pull requests to "main" and "develop" will trigger the analysis, this way, developers will be able to verify the quality of their code during development, and a final time before merging their changes.

The quality gate will be the built-in quality gate, recommended by Sonar Cloud, which focuses on keeping new code clean, to minimize the project's debt time and therefore lowering the amount of effort needed to improve old code.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

To facilitate the development of new features we'll be utilizing the Git Feature Branch Workflow, with a main branch that will be updated whenever a release is ready, and a develop branch which will include the changes made during product development. Whenever a release is due, a new release branch with the format "release/x.x.x" will be created from develop. This branch will be used to add needed documentation, fix bugs, or improve new code's quality. When everything is working as intended, a pull request from the release branch to the main branch will be created. After it's been approved, the branches will be merged and the release branch must also be merged into develop, to make sure it's up to date with any changes made.

When contributing to the develop branch, developers must create a branch which will map to an open issue, after the necessary changes are made and the issue is complete, a pull request must be made to merge the feature branch into develop, this request should be reviewed by at least one other assigned member to make sure the new functionality is working as expected and the new code follows the project coding style guidelines.

As stated above, a feature branch must map to an open issue, which maps to a user story on the project's backlog.

A user story is considered done when all functionalities needed for it to work have been developed, it has been subject to testing (both static as well as by using the system) and these were successful, the code has been reviewed and the system with its new changes is ready to be used by the target users.

3.2 CI/CD pipeline and tools

To implement Continuous Integration into our project we'll be using GitHub Actions, which provides an easy way to build pipelines and run them when certain conditions are meant.

We'll be creating scripts which run on every push and pull request, each script will belong to a project. These scripts will build the project and run the available tests, if those phases are successful, the project it belongs to will then be analyzed, and the results will be available on the Sonar Cloud platform. Both scripts run on the latest ubuntu image.

1. **deliveries-engine.yml:**
   This workflow will trigger 2 jobs:
   build - build the deliveries-engine Maven project, checking if it succeeds
   sonar - runs the SonarCloud static code analysis for the deliveries-engine project, checking if it passes the quality gate
2. **t-tracker.yml:**
   This workflow will trigger 2 jobs:
   build - build the t-tracker Maven project, checking if it succeeds
   sonar - runs the SonarCloud static code analysis for the t-tracker project, checking if it passes the quality gate

For the **CD** part, there are also two scripts, one for each project, but, in this case, each one of them will run on a GitHub Self-Hosted Runner installed on the given virtual machine.

These runners allow us to easily deploy our projects, when pushing/pull requesting to specific branches.

Each project has its own ***docker-compose.yml***, which will deploy all the parts of the same project, this parts being:

- **SpringBoot project**
- **MySQL Database**
- **Frontend**

There was an issue with the ***sudo*** command, as the runners were running on one of the group members' user, and this depends on the UA's IDP, this could give permissions/timeouts errors, when executing the ***sudo docker-compose up*** command.

To get around this, I created a new user, giving him root permissions. This can be done with the following commands:

```
$ sudo adduser [username]
```

```
$ sudo usermod -aG sudo [username]
```

After this step, we just have to enter the **VM** through **SSH**, using the previously created user's credentials:

```
$ ssh [username]@deti-tqs-01.ua.pt
```

And install the runners through this new user. This prevented the previous errors from ocurring.

4 Software testing

4.1 Overall strategy for testing

For testing of our backend functionalities, we're planning to use TDD, writing tests for the feature we are developing before developing the actual methods and logic needed. This approach requires the developer to think about what exceptions and what the intended return data is, before implement the methods themselves.

To test our user interface, we're choosing to use BDD, integrating Cucumber with Selenium, so that our tests emulate real scenarios.

All tests are developed using Junit5 as the base framework.

For isolating components we'll be using Mockito, so that we can test them individually, without interactions between different components. SpringBootMockMVC will be used to simulate interactions with our API when developing integration tests. And finally, as stated above, for our user interface we'll use Selenium to simulate user behavior, and Cucumber to represent real usage scenarios.

4.2 Functional testing/acceptance

For functional testing we opted for using Selenium, which can used with both Chrome and Firefox by using their respective drivers.

We also chose to use Cucumber, which facilitates writing tests from the end-user's perspective. To accomplish that, the different webapp features should be separated into individual feature files, which in turn must included all needed scenarios that could be played out by users, both valid scenarios as well as invalid, to make sure our system is validating and responding correctly to the user's input.

To make these tests more readable and maintainable, we use the PageObject Pattern, which means that for every page on our application, there will be a class representing that page, and the tests themselves will utilize those classes.

4.3 Unit tests

Unit Testing should be done for all appropriate classes, repositories, services, controllers, and any other extra classes that are developed and used.

All relevant methods should be tested with, at least, one valid and one invalid input, to make sure there are no unexpected behaviors.

When testing a component, all interactions with other components must be mocked, to successfully isolate it.

Tests should also have a descriptive name to better identify what exactly is being tested, closely following the name “when{condition}_then{result}”. For example, when testing a method which verifies if a product is in stock, good method names would be “whenProductIsInStock_thenReturnTrue”/ “whenProductDoesntExist_thenThrowProductNotFoundException”.

4.4 System and integration testing

Integration tests should be developed for all controller endpoints, to simulate real interactions with the rest api. These tests will be very similar to the unit tests developed for the controllers, but there will be no mocking of the services, therefore relying on the actual implementation.

These tests utilize an in-memory H2 database so that the tests are done safely and don’t interfere with the state of the actual database.