deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Filipe Gonçalves [98083], Gonçalo Machado [98359], Catarina Oliveira [98292]
v2020-05-03

1 Project management

1.1 Team and roles

Filipe Gonçalves	Gonçalo Machado	Catarina Oliveira
Team Manager	DevOps	QA Engineer
Product Owner		

1.2 Agile backlog management and work assignment

The backlog for the project is made using **Jira Software**, and the issuing flow using user stories is the following:

- Creation of user stories, new ones for users with no tasks for the current sprint or new ones for each new sprint.
- Distribution of issues, between all the team's elements, by point system and technological knowledge; the team gathers and meets and is decided which issues each element will take
- When the issue is set as **DONE**, by our **Definition of Done** being fulfilled, the element in question, would open a **Pull Request** in **GitHub** and notify all members of the team
- For a **Pull Request** to be validated, at least half of the group members, must review it and approve it, else the owner of the **Pull Request** would improve the quality of the code, with coherent tests
- After approval, the branch would be merged to the **dev** branch
- In the end, the branches were eliminated and the issues closed, restarting the process yet again

The **point system** is defined by the difficulty of the issue and time spent, so a task with 9 points is typically a complex and time consuming one, while a task with 5 points can be either difficult but fast, or easy but with a lot of time spent on it.

2 Code quality management

2.1 Guidelines for contributors (coding style)

One of the guidelines the group uses, while creating the Controllers, is using the **DTO** pattern, Data Transfer Object, which, in a **Request Body** annotation, does not leave our Object in an **HTTP** message, making it more secure.

Secondly, we try to make code eligible for both professors and colleagues, which means that we had to adopt some methods, mentioned in **ASOP** [1]:

- Do not import full libraries (**import java.util.***), and choose to simplify reading with more objective imports (**import java.util.Date**)
- Do not capture generic exceptions (**Exception**), but implement our own exceptions:
    - **BadRequestException**
    - **AddressNotFoundException**
    - **BadLegoDTOException**
    - **BadOrderLegoDTOException**
    - **BadScheduledTimeOfDeliveryException**
    - **ClientNotFoundException**
    - **LegoNotFoundException**
    - **OrderNotFoundException**
- Do a brief comment at the start of each important file as to explain the use of the file
- Use comments as a way to know what's left to do (**// TODO …**) or while debugging errors
- Try and maintain a simple and coherent code structure, without having methods with more lines than necessary
- Do not repeat unnecessary code in a single file
- Have suitable names for the tests, like **test_<method>_<parameters>_<return-value>** for the non functional tests and **<step>** for the functional tests

2.2 Code quality metrics

For static code analysis, we use **SonarQube**, which implements many pertinent evaluations on the code, while also reporting bugs, vulnerabilities, code smells, and more.

The quality gates we use are the default provided by **SonarQube**, which fail the code analysis review when:

- **Coverage** is below 80.0%
- **Duplicated Lines** are greater than 3.0%
- **Maintainability Hailing** worse than A
- **Reliability Rating** worse than A
- **Security Hotspots Reviewed** are less than 100%
- **Security Rating** is worse than A

SonarQube is implemented for new written code, using their service **SonarCloud** in our workflow: each time a **Pull Request** is open in a branch, the code will be analyzed for that specific branch.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

For the workflow of the project, we use **Jira Software** and **GitHub Flow**. While **ZenHub** works as the two of them in one, we could not get it started, because of an internal problem. In the moment of the decision, we chose to use these tools instead of waiting for a fix.

With **Jira Software** we can create issues, sprints and a suitable backlog. We can also look at the completion rate of each task (**Todo**, **In progress**, **Done**) as well as which task should be implemented first by our point system (explained in **1.2**). We can also set up epics for each user story and create issues on each epic, which helps organize the workflow of the project.

With **GitHub** we can create different branches using the next naming structure:

```
bug/ - to resolve bugs
imp/ - to improve features
new/ - for new features
exp/ - experiments/ trash
```

We maintain the organization by creating branches for each epic or issue, creating **Pull Requests** when the implementation is done and **Merge** branches when the **Pull Request** is accepted by our **Definition of Done** and by at least half the team members, which guarantees that most of the team is always involved with the work done by the colleagues and that everyone is contributing for this process. In a team of three, we need at least another person to review the work done to merge the **PR**. Nonetheless, for each **Pull Request** we create, we also use **SonarCloud**, as mentioned before, to know if we should merge or not. These three criteria are what we, as a team, view as visible and important in maintaining a good development workflow.

Definition of Done, like the name entitles, is our own concept of when a user story/work is implemented in our project. In our case, our **Definition of Done**, will depend on the acceptance criteria we give to each user story and to the work each needs to have. In a general manner, we define a user story as done, when:

- We have implemented both tests and code
- When we have completed both the frontend and backend
- When we can guarantee the connection between frontend and backend works
- When we can use the application and the user story works

3.2 CI/CD pipeline and tools

Continuous Integration is being provided by **GitHub Actions**, which we connect to **SonarCloud**, as mentioned before. Everytime we create a **Pull Request**, the **SonarCloud** will analyze the code committed and fail it, or pass it, depending on the quality gates mentioned in 2.2. It also helps with analyzing the **Spring Boot** unit tests and web tests with **Selenium**. As for the Integration Tests, as we need to have the whole system deployed, we will build the docker image and test them.

Continuous Deployment is being provided by **GitHub Actions**, **GitHub Packages** and **self-hosted runners**. When an element of the group creates a **Pull Request**, the **GitHub** workflow will generate a **docker** image and push it to **GitHub Packages**. In the Virtual Machine given to us, it will be placed a self-hosted runner, which will pull the docker image created by the workflow, when the workflow that pushes the image is successful. Then, it will run docker-compose down, build and finally up, to deploy the project.

We chose **Continuous Deployment** instead of **Continuous Delivery**, because we preferred to have an automated system that would deploy our project, instead of manually doing so. We also recognized that using a Continuous Deployment pipeline would let us see in real time the changes we made to the project, if the deployment was successful, and notice any bugs or errors, instead of having to wait till a new deployment with Continuous Delivery.

4 Software testing

4.1 Overall strategy for testing

For the test development strategy, we adopt **Test Driven Development** (**TDD**), as best as we can, and whenever possible. Everytime a **Pull Request** is made, or a commit is pushed, we try to integrate both tests and code development for the tests made. Consequently, only **Pull Requests** with valid tests and code for them where admitted.

We try to use the best tools for each specification, which includes **Selenium** and **Cucumber** for web testing, **JUnit**, **Rest-Assured**, **Mvc Testing** and **DataJPATest**, for the others.

4.2 Functional testing/acceptance

The Functional Tests are developed using **Selenium** and **Cucumber**.

In these tests, we try to create different Scenarios for each User Story we can implement in the frontend. For this, we should have both frontend and **Rest APIs** running, as the web apps need to connect to the specified endpoints.

We test with different database states, with data and without data, as well as with **API** connection and without, which can give us a more diverse space of possible scenarios.

Each user story contains a <user-story>Steps.java file and a <user-story>.feature, which maintains the code organized. Each step should be a valid step for a simple user while using the web application.

4.3 Unit tests

The Unit Tests have the sole purpose of testing only a portion of the system, guaranteeing that the most basic methods of the whole system work as expected.

Consequently, we use this type of testing on repositories, controllers and services, and tested each method they have.

For the Controllers, we use **MockMvc** and **RestAssuredMockMvc**, in the specific delivery API and engine, respectively. As such, we could use mocks in the place of the services used, isolating the behavior of the Controllers.

For the Services, we use the **Mockito** extension, so we can test every method independently, both when they succeed and when they don't or shouldn't, regarding the different scenarios and use cases.

FInally, for the Repositories, we use the annotation **@DataJpaTest**, and test each method we personally created, and also some methods we frequently use, like **findAll()**, when they had a correct return or when they couldn't find anything.

4.4 System and integration testing

The Integration Tests are created in order to evaluate if the system works as a whole and any response to a client request does not raise any problems or exceptions.

Furthermore, we create an IT test file for each Controller we have, and test each method, both with bad request results and with a successful request.

With this, we can test and understand if the system works as we planned and devised.

References

[1] - AOSP Java Code Style for Contributors. Retrieved From:

https://source.android.com/setup/contribute/code-style