TQS: Quality Assurance manual

Pedro Monteiro [97484]*, Daniela Dias *[98039], Eduardo Fernandes [98512], Hugo Gonçalves [98497]
v2022-06-09

1 Project management

1.1 Team and roles

Team Member	Role	Responsibilities
Eduardo Fernandes	Team Coordinator	Ensure that there is a fair distribution of tasks and that members work according to the plan. Actively promote the best collaboration in the team and take the initiative to address problems that may arise. Ensure that the requested project outcomes are delivered in time.
Daniela Dias	Product Owner	Represents the interests of the stakeholders. Has a deep understand of the product and the application domain; the team will turn to the Product Owner to clarify the questions about expected product features. Should be involved in accepting the solution increments.
Hugo Gonçalves	QA Engineer	Responsible, in articulation with other roles, to promote the quality assurance practices and put in practice instruments to measure the quality of the deployment. Monitors that team follows agreed QA practices.
Pedro Monteiro	DevOps Master	Responsible for the development and production infrastructure and required configurations. Ensures that the development framework works properly. Leads the preparing the deployment machine(s)/containers, git repository, cloud infrastructure, databases operations, etc.
---	Developer	ALL members contribute to the development tasks.

1.2 Agile backlog management and work assignment

To easily keep track of the tasks and job that needs to be done, as well as quickly assign it to the members of the team, we are using the project management tool ***Jira***. The Jira platform is an agile project management tool that allows us to easily manage tasks and issues related to the project as well as do the necessary planning required by the project. We can also the tasks into several ***Sprints***, organize them using ***Epics***, setup deadlines, etc.

More information about Jira software can be found here.

2 Code quality management

2.1 Guidelines for contributors (coding style)

Adopting a common coding style in our projects helps us to easily read and understand the code that is being written by our partners. Adopting a pattern inside the team also reduces the time a team member takes to edit and change the code written by another member if needed.

Once we are using mainly two different languages in our projects – Java and JavaScript – that have their own specificities, we are going to use different conventions for each one of the languages.

For Java, used in the backend modules of our system, we will try to stick with the Google Java Style. This is one of the most famous style conventions for Java and, therefore, there are a lot of tools and plugins available for the different IDEs that may help us enforcing this style.

For JavaScript, we will also use the Google JavaScript Style and will try to stick with it in our Frontend projects.

2.2 Code quality metrics

To better evaluate and improve the quality of our code, we are using Codacy. Codacy is a platform that will perform a static analysis of our repositories and indicate possible problems and improvements that could be done. It will also alert us of any possible security vulnerability and make suggestions so we can fix them.

We will stick to the recommended quality gate as we agree that it is enough to make sure our code will be ready for production and stable enough. Those recommended metrics are:

- Issues below 20%
- Complexity below 10%
- Duplication below 10%
- Coverage above 60%

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

In order to make sure that all the developed code meets the requirements, was reviewed and won't break already existent code we are adopting the GitHub Flow. The GitHub Flow gives some recommendations on how the development flow for new features should be, i.e., what steps each feature should follow until it gets to the final version of the product. This means that when developing a new user story or feature in general we should follow the next steps (in order):

1. **Create a new branch using a short and self-descriptive name.**

   This step will allow the developers to perform their job in a controlled, stable, and safe environment. This help the developers who are developing the feature to be sure no one will

interfere with their job, as well as gives an opportunity for other developers to review the job that's being done on that particular user story.

2. **Perform changes in the created branch**

   Once a branch for the new user story was created, all the code related to that user story should be committed to this branch.

3. **Create a Pull Request and ask collaborators for review**

   Once the feature is completed, a pull request should be opened to the "develop" branch. Each Pull Request should include a self-describable title and a short description including:

   - The news/changes that Pull Request will provide.
   - A link to the JIRA user story related to that specific Pull Request.

   The developer should ask other collaborators to review and approve his pull request, and the latter ones should review the code and either comment/ask for changes or approve the changes.

   In the end of each sprint, a Pull Request to the main branch should be done and merged, if possible.

4. **Merge the Pull Request**

   Once the Pull Request was created, at least one approval will be required to merge the pull request. In order to ensure this, the main and develop branches where locked and configured to only be mergeable once there is at least one approval and all the **checks** have passed. The checks matter will be discussed in a further point in this document.

5. **Delete the branch**

   After the Pull Request is accepted and merged into one of the main branches (main, develop, etc.), the origin branch of that pull request should be deleted. This will avoid a pollution of branched in the git repository.

Another very important metric is the **Definition of Done** (**DoD**) for each user story. This helps us to understand at what point in time, and after which steps, a user story can be considered "ready". In our case, a user story will be considered ready every time it successfully passes through the following steps:

1. The implementation of the user story is ready and meets all the acceptance criteria.
2. The user interface for the user story is accordingly to the design.
3. The code refactoring is complete.
4. The user story has unit and integration tests written and those tests were executed and passed.
5. The feature is deployed in the production environment.

3.2 CI/CD pipeline and tools

Continuous Integration and Continuous Delivery/Deployment are very important paradigms as they allow developers and organizations in general to deploy new features quickly as well as deploy bug fixes in an efficient way.

To handle the CI/CD pipelines, we are using Circle CI software. Circle CI is a continuous integration platform that allows developers to configure **pipelines** that will be executed when certain conditions are met – when some code is pushed to a certain branch in the repository, for example. These pipelines are very flexible and can run several **jobs.** Each job will execute a certain action like building a project, performing the unit testing, or deploying the new changes to a remote environment.

There are also another CI/CD pipelines like GitHub Actions, Jenkins, etc. We've chosen Circle CI for this project as we already had some previous knowledge about how it works as well as its simplicity.

The pipelines for each project should be defined in using a config.yml file present inside the ".circleci" directory inside the root of the project (.circleci/config.yml). Inside this file we can setup all the jobs we want to run. With these jobs we can, for example, run the tests and make sure the code is acting as it should, deploy an artifact for that version of the code, etc.

The following lines present one of the config.yml files used in one of our projects (Core Backend).

```
version: 2.1

jobs:
  build:
    machine: true
    steps:
      - checkout
      - run: mvn package -Dmaven.test.skip=true
      - run:
          name: Copy JAR file
          command: |
            mkdir -p ~/artifacts/
            cd ./target
            find . -name "*.jar" -exec cp {} ~/artifacts/ \;
      - store_artifacts:
          path: ~/artifacts
          destination: artifact-file
  test:
    machine: true
    steps:
      - checkout
      - run: mvn verify
      - run:
          name: Save test results
          command: |
            mkdir -p ~/test-results/junit/
            find . -type f -regex ".*/target/surefire-reports/.*xml" -exec cp {}
~/test-results/junit/ \;
            find . -type f -regex ".*/target/failsafe-reports/.*xml" -exec cp {}
~/test-results/junit/ \;
          when: always
      - store_test_results:
          path: ~/test-results/junit

  deploy:
    docker:
      - image: arvindr226/alpine-ssh
    steps:
      - checkout
      - run: ssh -oStrictHostKeyChecking=no -v $USER@$IP
"~/SendASnack/Core/Backend/deploy.sh"

workflows:
  test-build-deploy:
```

```
jobs:
  - test
  - build
  - deploy:
      requires:
        - test
        - build
      filters:
        branches:
          only:
            - develop
            - master
```

In this file we are firstly setting up all the jobs that will be run on our pipeline. In this case, these jobs are "**build"** and "**test"**. In each one of these jobs we are defining the steps that each job should follow. In the "build" job, for example, we are instructing the pipeline to package the project ignoring the tests using "`mvn package -Dmaven.test.skip=true`". After the code was packaged (the .jar was created) we are creating a new folder "artifact" and copying the generated JAR to it. After that, we call the step "store_artifacts" to store the generated JAR inside the workflow run, so we can get that artifact in the future, if needed.

The CD was also implemented using Circle CI and were implemented in the same config file as the CI pipelines. The CD is made by the "deploy" filter in the configuration file that sets that all the changes in develop, and master branches will be updated in the production environment. This is possible by connecting, via SSH to the production machine and execute a predefined script that will push the new changes from GitHub and re-build the containers that contain the apps.

4 Software testing

4.1 Overall strategy for testing

When possible, we've chosen to apply the **Test-driven Development** (**TDD**), principle, i.e., we started by creating the tests for a feature accordingly to how it should work, and once the tests were created then we implemented the feature so it could pass the tests. Once it passed the tests we would then refactor and improve the code that implements that same feature.

To test the backend components, and once they were developed using Spring Boot, we've used Junit 5 and the Spring Boot Starter Test dependency. The latter, provides a huge number of useful resources and annotations that will help us to write tests for Spring Boot.

4.2 Functional testing/acceptance

Functional tests will also be created using Cucumber + React, for our frontend applications, and Cucumber + Spring Boot, for our backend applications. These tests will be write taking into account the user perspective.

4.3 Unit tests

Unit tests for the major features implemented were written using a developer perspective, i.e., before implementing the user story/feature the same developer that would implement that feature would also write the tests to test it.

In Spring Boot projects, Junit 5 was used to perform the unit testing.

4.4 System and integration testing

Integration tests will be created to test parts of the system that can only be properly tested when integrated with some modules. Examples of these features are the connections to the databases. In order to implement these tests we've used Test Containers + Junit to create, on the fly, all the necessary integration that we would need.