TQS: Quality Assurance manual

Malwina Schonhofer [116718], Juliusz Szymajda [116726], Piotr Czapla [116634], Maciej Adamczewski [116748]
V2023-06-26

1 Project management

1.1 Team and roles

Team Leader – Maciej Adamczewski
Product Owner – Juliusz Szymajda
QA – Piotr Czapla
DevOps – Malwina Schonhofer
Developers – all above-mentioned

Team Leader
Distributes in a fair way the work by the team elements and ensures that all members are working according to plan. Resolves conflicts that may appear between people. Make sure that the deadlines and deliveries are fulfilled.

Product Owner
Ensures that stakeholder interests are met. Is involved in the new increments to make sure the functionalities are as intended. Answers questions about features that the development may have.
QA
Ensures, in articulation with other roles, the quality of the product by promoting QA practices and their use by the team. Puts in practice instruments to measure the quality of the deployment.

DevOps
In charge of the deployment and production infrastructure and its required configurations. Starts the preparation of the deployment machines/containers, create and prepare the code repository so that the team has access and can start developing, cloud infrastructure, and database operations, among others.

1.2 Agile backlog management and work assignment

The agile tool used by the group was Jira and at the very early stages of development we used Notion since it makes it easier to communicate and comment on each other's work. The whole workflow was divided as such:
- four sprints were created each for every period of development and were filled with user stories (in this case called Jira issues).
- each user story had its tasks defined.
- the backlog served as a storage for tasks that had to be done.
- To work on a task, a branch was opened with its name.
- When a task was considered done by the element that was responsible for it, a pull request was done and once it was reviewed by a team member, it was merged to the main branch.

We did not used an estimation system for the tasks.
In case some element completed their tasks sooner than expected, some of the other tasks were reassigned to them or if someone's task was closely corelated with someone's else's, the task would be moved to this person.
Each time a new task was picked up it was marked as "In progress" and once it was finished it was marked as "Done".
Every task had its own branch and whenever the issue was complete, the branch was merged with the main branch. All the
branches' names had to follow the patter feature_name_of_the_service and commits had to be descriptive and not overwhelming.

2 Code quality management

2.1 Guidelines for contributors (coding style)

Since we did not use technologies which largely differ from Java Spring Boot (for example different flavors of SQL etc.), the set of following rules was followed throughout the whole project:
- Do not ignore exceptions (only when sure that is not a problem is when exceptions can be ignored) and use meaningful exceptions
- Fully qualify imports
- Use camelCase naming variable convention
- Use standard brace style
- Treat acronyms as words
- Use logs not prints
- When in doubt, be consistent with the surrounding code.

2.2 Code quality metrics

For static code analysis, the tool used was SonarCloud due to the fact that it helps maintain code quality and security by detecting code smells and bugs.
The analysis of SonarCloud was triggered whenever a push in the repository was done to the main branch. This ensured that each new piece added to the whole would not cause any trouble.
The quality gates used were the defined ones by SonarCloud since they are acceptable to guarantee that every code change committed to the project is free of issues and vulnerabilities. Continuous delivery pipeline (CI/CD)

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

As mentioned previously, every user story was divided into tasks that were distributed by the team members. Each task had its own branch whose name was feature_”name”. So, whenever we looked at a branch name, we knew what was being done.
When a task was complete, the developer would make a pull request to the main branch which had to be reviewed by other team member. The reviewers could approve it, request changes, or leave comments for improvement. Once the pull request was approved and the autonomous testing did not detect any issues, the code was merged with the main branch.
To ensure that the code was reviewed, a rule was defined in the repository.

The Definition of done for a story was as follows:

- The unit/integration tests were written, executed and passed
- The user story had been peer-reviewed
- Assumptions of User Story met
- Project could build without errors
- Produced code lived up to presumed functionalities

3.2 CI/CD pipeline and tools

The continuous integration pipeline was implemented with Github Actions since it makes it easy to automate the workflows.
We defined a workflow, which would go off each time there is a push on the main branch.
The workflows were integrated with SonarCloud, which gave us a code analysis every time something was pushed or pull requested to main branch. If the code analysis and/or the tests failed,
the code could not be merged with the branch.
The project also includs a CD Pipeline, this pipeline makes use of Github Actions and DockerFile which everytime there is a push on the main branch and the above-mentioned tests do not fail, the .jar file generated by the app gets dockerized and deploy into the Google-Cloud-Platform service.

3.3 System observability

In order to observe the current state of the system, we mainly focused on system logs since they are extremely useful for debugging purposes and on observing API endpoints. For observing API endpoints we used Swagger which enabled us to observe how our endpoints behave and react to given requests.

4 Software testing

4.1 Overall strategy for testing

The overall development strategy followed was Test-Driven-Developmen, since this strategy can be beneficial to the developers by decreasing the debug time, bugs and errors. The main tools and frameworks used were Junit5 for overall testing and Mockito for creating mocks in order to test different system independently. Also, the names of the tests had to be descriptive, identifying what should be returned and what functionality was being tested, e.g., givenCorrectData_whenSaveAdmin_thenReturnTrue and givenAdminId_whenGetAdminById_thenReturnAdmin.
For the functional tests, we used a Behavior-Driven-Development strategy with Selenium, creating the tests after
the backend and frontend of a user story were connected.

4.2 Functional testing/acceptance

For the functional testing, we decided to use Selenium with a webdriver. We used a Page Object Pattern with the Selenium tests to ensure clarity in the test responses.

4.3 Unit tests

Unit tests were developed to test the behavior of the repositories, controllers, services, and any other classes eventually created. Not only the “normal” scenarios were tested, but also some scenarios of errors.
To make the tests faster, we limited the context of some of them. For the repository tests, we used @DataJpaTest annotation to load only the @Repository spring components, and for the controller test we used @WebMvcTest to test the APIs (@Controller annotation). In the case of the controllers, we had to mock the beans used by the controllers with @MockBean.
In the services tests, we used Mockito to mock the repositories' behavior, making it independent of the persistence layer.

4.4 System and integration testing

The integration tests made sure the individual parts (repository, service, and controller) worked correctly together. These tests were created accordingly with the endpoints in the controllers. And as in the previous tests, there were tested “normal” cases and some with errors.