deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Martinho Martins Bastos Tavares [98262], Tomás Candeias [89123], Rodrigo Lima [98475], Afonso Boto[89285]
v2022-06-01

1 Project management

1.1 Team and roles

Role	Description	Assignee
Team Leader	Coordinate the team and prioritize and distribute work, making sure that the project outcomes are delivered in time	Rodrigo Lima
Product Owner	Represents the client of the project, and is the one with knowledge of the project's requirements, involved in reviewing increments	Tomás Candeias
QA Engineer	Guarantees that the team follows the determined QA practices, and is the one that defines the QA criteria and processes	Martinho Tavares
DevOps Master	Responsible for infrastructure, deployment of services, repository and overall CI/CD management	Afonso Bôto

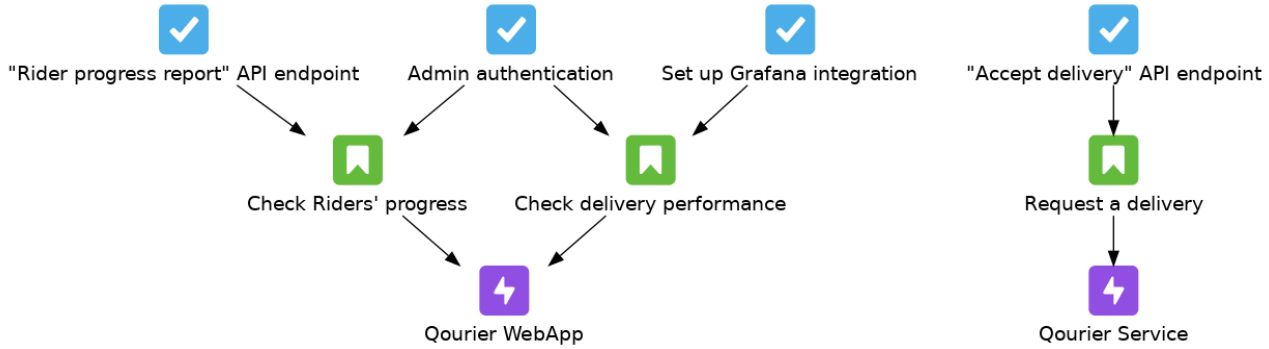
Role	Description	Assignee
Developer	Development of the project's code	All members

1.2 Agile backlog management and work assignment

The project was divided into two parts: the Delivery Engine, which offers the delivery service from which companies can make delivery requests, and the concrete Laundry Service which is an implementation of one of those services that requests deliveries. Being divided in two, the project's team of developers was also split into pairs, where each pair focuses on one of the two parts. With this in mind, we decided to make use of two distinct development branches, one for each part of the project, since the code for both parts is shared under one single repository.

For backlog management and work assignment we use Jira, as we were already familiarized with it. We follow an Agile workflow, defining Epics for parts of the application covering a multitude of related user stories (e.g. the Delivery Engine management web app for the Admin), User stories to focus our attention in providing value with each increment (e.g. check performance of Riders in the Delivery Engine), and Tasks which have to be developed for the User stories (e.g. Grafana integration into the management web app).

With this, we devised an hierarchy of work units, with Epics at the bottom and Tasks at the top, in order of work granularity. Below is an example graph demonstrating this organization.



This organization has an impact on what user stories will be developed in parallel. As will be explained further, the user stories are mapped to feature branches in the repository, and so stories that are dependent on the same tasks should not be developed in parallel. In the image above, for example, the "Check Rider's progress" and "Check delivery performance" stories should not be done at the same time since they both depend on the "Admin authentication" functionality, but they can be developed in parallel with the "Request a delivery" story.

2 Code quality management

2.1 Guidelines for contributors (coding style)

The coding style we adopted is the Google Java Format, with AOSP style indentation (4 spaces instead of 2). This style is enforced in the CI pipeline, with the import ordering and removal of unused imports being automatically handled by the pipeline and changed in a commit.

We initially opted for manually formatting the rest of the code ourselves to minimize the risk of merge conflicts in case we forget to pull changes that the pipeline may perform. Since we were using IntelliJ IDEA, we took advantage of its formatting plugin for this style (Intellij IDEA plugin). However, its formatting rules were conflicting with the ones checked by the GitHub action enforcing the style in the CI pipeline, so we ended up letting the action automatically format the code.

2.2 Code quality metrics

We integrated static code analysis into our CI pipeline, which is run on every developed feature and on merges to the main branch. The platform we use is SonarCloud, and we opted for using the default Quality Gate (Sonar way), as we want to focus on the quality of new increments (since we are starting a project from scratch, all code will be evaluated since it's new). Because we don't have any specific quality necessities for our project, we believe Sonar's default quality requirements for new code are enough for our purposes.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

The workflow we adopt is the gitflow workflow, which specifies the usage of development branches with different roles and meaning. In this workflow, the user stories are mapped to feature branches, which focus on the development of that user story.

The work from both development branches is merged into the release branch two days before the end of each sprint. At the end of the sprint, the release branch is merged back into the main branch. We provide two days of lifetime on the release branch to develop bugfixes, so ideally the features for the current sprint are mostly done two days before it ends.

We make use of GitHub's Pull Request feature to incorporate work from the feature branches into the development branches and from the release branch into the main branch. The merges from the development branches into the release branch are not done with pull requests. Ideally, since they work on different parts of the project, there shouldn't be merge conflicts.

The pull requests into the development branches have as reviewers all the developers responsible for that part of the project. Pull requests from the release branch into the main branch have the Product Owner as the sole reviewer, in order to assess the overall project increment. This pull request should only be accepted by the Product Owner at the end of the sprint. This is a manual process because there may be problems during development or other unforeseen circumstances that may shift the time that the merge must take place.

A user story is considered developed when all tests defined for it pass (functional tests derived from the feature file, which in turn are written based on the respective Jira story's acceptance criteria, as well as unit and integration tests for the involved modules). This means that each pull request from a feature branch should pass the CI pipeline, which includes the Coding Style checks, the Maven tests and the Sonar static code analysis. After these pass, the pull request should be approved by the assigned reviewer, which is the available developer for that part of the project that is not the one who created the pull request. After the approval, the reviewer may merge the pull request.

3.2 CI/CD pipeline and tools

For Continuous Integration we opted for using GitHub Actions, for its simplicity to define workflows and the fact that it's already ingrained in our VCS.

[TODO: CD]

4 Software testing

4.1 Overall strategy for testing

We adopt a Behavior Driven Development (BDD) approach, using Cucumber to map user stories to feature definitions. Therefore, we start the development of each story by creating the feature file incorporating that user story, and then we develop the tests with the defined acceptance criteria in mind. Implementation of the necessary components for the story functionality is left to be done after the tests are created.

In light of the user story organization that we described in section 3.1, the tests developed for the tasks are given a different treatment. We use Test Driven Development (TDD) to define the tests first for these units of functionality (such as an API). Therefore, we declared that these aren't mapped to Cucumber feature files, and are simply realized in Java unit and integration tests.

4.2 Functional testing/acceptance

The functional tests should start with the Cucumber feature file laid out in Gherkin, and then have the Cucumber steps implemented. The minimum functionality should be developed so that the tests run, but fail. This should be present in one or more commits, but they should be the first. Only afterwards is the actual functionality developed.

Taking a behavior driven approach (BDD), the tests should not be concerned with the application's internal functionality, and should only be concerned with its output to the user's general actions, which are at the highest level of abstraction. In this way, the tests should be written in the perspective of the user, in a closed box fashion.

4.3 Unit tests

Unit tests are developed for singular modules (classes) of the application. Test driven practices are followed (TDD), and so the tests should be developed before the functionality of the modules is fully written, in a similar way that it's done for the functional testing: write the minimum functionality to make the tests successfully run, but fail.

These tests are written to test the application's internal modules, and so they are written taking the developer's view of the module.

4.4 System and integration testing

System and integration tests are done in the same way as unit tests, but not with as much urgency, as they depend on more than one application module. They should similarly take the developer's perspective into account (open box), and not the end-user's, since the objective is to test the application's internal functionality.

In terms of Rest API testing, however, the integration tests may also be done with focus on a closed box perspective, as it's the entrypoint for application developers to interact with our platforms.