TQS: Quality Assurance manual

Carolina Marques [85084], Diogo Silveira [85117], Edgar Morais [89323], Gabriela Santos [51531]
v2020-05-15

1 Project management

1.1 Team and roles

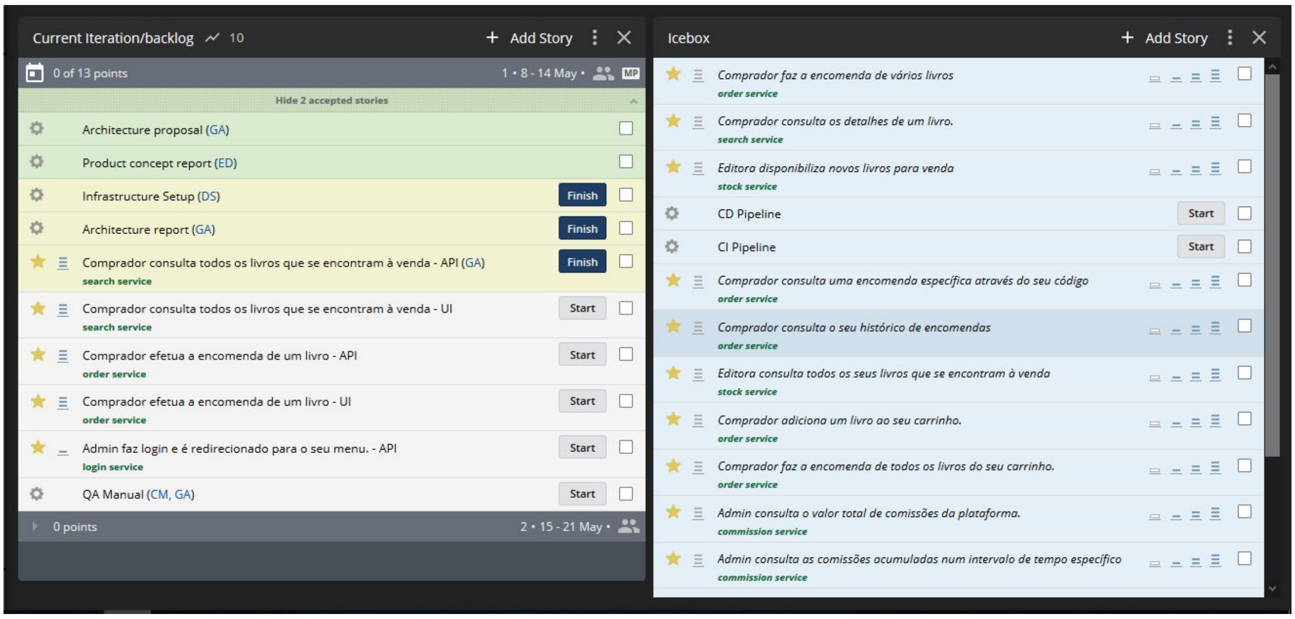
Gabriela : Team Manager + Developer . This member will ensure a fair distribution of tasks, effective backlog management and monitor the pace of the Sprint. Will also support DevOps in case of need.
Diogo : DevOps + Developer. Responsible for the development and production infrastructure, along with the required configurations.
Edgar : Product Owner + Developer. Represents the interests of the stakeholder, by providing insight on the product's features and establishes the acceptance criteria that the developers must follow.
Carolina : Software Quality Assurance Engineer + Developer . Will monitor every phase of the software development process, providing insight on the best QA practices and suggesting regulations for the team to follow. Will also establish the quality gates for accepting a new feature for the product.

1.2 Agile backlog management and work assignment

Overview of the main Agile practices adopted in the project:

- Iterations - Each sprint will take place as a one-week iteration. The goal of the sprint is to increment as much as possible the amount of features included in the product, while taking into account the time restrictions. The product owner establishes the priority for each task, while the team provides input on the number of points for each task and the amount of technical work needed.
- Management of the Product Backlog - The team manager will determine how much work is needed to complete each task by reviewing the points and assigns it to a team member according to their schedule and development strengths. The sequence of the backlog should also reflect the Product Owner's priorities.
- User stories - The scope of the project and its main features are translated into user stories. Each user story will correspond to an entry in the product's backlog. They are described in detail in the Product Specification Report.
- Scrum meetings: We hold the first official meeting every Friday after delivery the previous' sprint results in the practical class and set up the next sprint. Every Wednesday, we also hold a pre-delivery meeting. There are daily updates on the features' progress on Slack by the developers.

Pivotal Tracker was selected as the main project management tool and our project can be accessed in the following link: [https://www.pivotaltracker.com/n/projects/2447537].



Architecture proposal (GA)
Product concept report (ED)
Infrastructure Setup (DS) Finish
Architecture report (GA) Finish
Comprador consulta todos os livros que se encontram à venda - API (GA) Finish
search service
Comprador consulta todos os livros que se encontram à venda - UI Start
search service
Comprador efetua a encomenda de um livro - API Start
order service
Comprador efetua a encomenda de um livro - UI Start
order service
Admin faz login e é redirecionado para o seu menu. - API Start
login service
QA Manual (CM, GA) Start

Comprador faz a encomenda de vários livros
order service
Comprador consulta os detalhes de um livro.
search service
Editora disponibiliza novos livros para venda
stock service
CD Pipeline Start
CI Pipeline Start
Comprador consulta uma encomenda específica através do seu código
order service
Comprador consulta o seu histórico de encomendas
order service
Editora consulta todos os seus livros que se encontram à venda
stock service
Comprador adiciona um livro ao seu carrinho.
order service
Comprador faz a encomenda de todos os livros do seu carrinho.
order service
Admin consulta o valor total de comissões da plataforma.
commission service
Admin consulta as comissões acumuladas num intervalo de tempo específico
commission service

Figure - Example of the results of our first meeting sprint (I2)

The first iteration (I1) of the project consisted of the pre-planning stage, thus it was not accounted for as a full sprint in our Pivotal Tracker. The outcome of the first work iteration is displayed below:

Achievements - I1
Product concept was thoroughly developed in the Product Specification Report.
Architecture proposal.
Orchestration of the main infrastructure (Maven project and DB) using docker containers.
Set up of the backlog using Pivotal Tracker, in the form of user stories and epics. Missing only the estimation of each story.
Breakdown of the UI and presentation of the first prototypes.

Missing - I1
Nothing. The team is ahead of the sprint's milestones.

2 Code quality management

2.1 Guidelines for contributors (coding style)

Coding Style Adopted:

- *Exceptions* :
  - We will not ignore them, always handling them according to the error we will obtain
  - Won't use generic exceptions (ex:Throwable) since this can obscures the failure- handling properties of our code
- *Fully qualify imports*, making it obvious what classes are being used and it makes the code more readable.
- Use *Javadoc standard comments* . Besides making the methods understandable, our public methods are part of an API and therefore require Javadoc.
- Write *short methods* to keep them small and focused.
- *Fields* will be initialized in standard places (we will initialize them at the top of the page)
- Follow field naming conventions
  - Non-public, non-static field names start with m.
  - Static field names start with s.
  - Other fields start with a lowercase letter.
  - Public static final fields (constants) are ALL_CAPS_WITH_UNDERSCORES.

2.2 Code quality metrics

Sonarqube was our option to analyse the code overall performance .

Key aspects that will be taken into account:

- Readability
- Overall Performance
- Usability
- Security
- Maintainability

Quality gates defined for accepting a new feature:

- Minimum unit testing coverage of 70%;
- Fixing critical/ major bugs;
- Critical Vulnerabilities will have to be fixed;
- Critical code smells will be eliminated;
- New code cannot add more than one hour of technical debt to the project (SonarQube tends to overestimate how much time it take to solve an issue) ;
- Security Spots will have to be analysed and will only be fixed if the team agrees to do so.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

We have adopted a Feature-branching workflow with code reviews. Each user story found in Pivotal Tracker has a corresponding development branch in our GitHub repository. As soon as a developer finishes its feature, they create a pull request for the respective branch and assign the rest of the team as reviewers. The link to the pull request is added to the user story in Pivotal Tracker (*Code section*) and the story should be marked as “Delivered”. After the team successfully completes the code review process and the feature is officially accepted (and merged to the master branch), the new feature will appear as accepted in the Pivotal Tracker. This tool also attaches the name of each developer to the feature that they will be responsible for, so it also works as record of the delegation of tasks.

The main actors of the code review stage are the Product Owner and the QA Engineer. They are responsible for the assessment of the feature, in terms of meeting the required functionality envisioned by the Product Owner and thoroughly checking if the required quality gates established by the QA engineer have been met. Overall, all the members of the team should comment on the code quality and test the endpoints or user interface by themselves, before approving. SonarQube results should be attached to each pull request. Optionally, attaching a Postman collection with the working endpoints of the feature is also recommend. In the end, most of the interactions regarding the code review process happened on Slack, the team’s communication tool, since it would speed up the refactoring process compared to a formal approach of only leaving reviews on the pull request itself.

3.2 CI/CD pipeline and tools

[Description of the practices defined in the project for the continuous integration of increments and associated resources. Provide details on the tools setup and config.]
CI pipeline and tools : Github Actions
CD pipeline and tools : Heroku
[Description of practices for continuous delivery, likely to be based on containers]

4 Software testing

1.1 Overall strategy for testing

For a typical API feature, the testing strategy consists of the following steps, in order:

- Unit tests without mocking using Junit5 (mainly for Model objects);
- Unit tests with dependency isolation using Mockito and Junit5 (for services);
- Unit tests for controllers using Mockito, MockMvc and Junit5;
- Integration tests for the repositories using Junit5. We want to properly test our repository methods against a replica of the production database, which could be counted as integration. However, the repository is being tested in isolation without any other components. This methodology will help us in terms of detecting any problems with the repositories early on, before ta full integration test;
- Full integration tests for the controller-service-repository pipeline with MockMvc and Junit5.

Before each step, running a static code analysis with SonarQube is recommended, in order to check the levels of code coverage and the impacts of refactoring the implementation code.

For a typical UI feature, the testing strategy consists of running functional tests with Selenium WebDriver and Junit5.

Overall, a developer that is responsible for a feature will write all the tests themselves, so that we can optimize the time spent on testing on the person that is most familiar with the content of the implementation code. We are planning to experiment on the Commission feature and have another developer writing the tests, to check if it is actually more beneficial to have a fresh pair of eyes on the code for the testing process.

In terms of TDD, we decided not to follow its key principle, which would be writing unit tests before implementing code. The reason lies on the fact that the timeframe for implementing the project is very short, so we needed to figure out the implementation details as we go, since we don't have a long pre-planning stage to write all the entities and methods of the problem as a team. Nonetheless, we aim to accomplish a detrimental TDD guideline, which is trying to develop with testing in mind. Our methods should be succinct and without complex control statements. We also spent a considerable amount of time modelling the project's entities and choosing how would the services work so that they could be easily tested.

Regarding BDD, we opted not to use Cucumber. Adopting a cost-benefit perspective, we did not think that we had a lot to gain from using natural-language constructs to test the project with the given timeframe for delivery. Furthermore, we consider that this tool would be more appropriate when working within a larger group of people that have varying levels of knowledge of the technical aspects of the implementation code, for example, people from management backgrounds, which was not the case here. By having adopted a peer code review methodology, we also feel that everyone on the team has a good overall understanding of the implementation code.

1. Functional testing/acceptance

Functional tests should be written using a black box approach, which means that the inner workings of the implementation code that is being tested are unknown to the tester. Therefore, the tester should assume a user perspective, even if they are person that actually wrote the code.

Policy for functional testing:
The goal of these tests is to simulate the typical handling of the interface of the marketplace by a real-life user. These tests should portray the use cases that were described in the Product Specification Report and they should verify if the system is fully compliant. Selenium WebDriver is going to used as the tool to test the behavior of the interface with a Chrome .

It is very important that the names of the testing methods correctly describe the intent of each test. The name of the class that contains these tests should end with "IT". They should be stored in a separate package called "acceptance".

2. Unit tests

Unit tests should be written using a white box approach, which means that the inner workings of the implementation code that is being tested are fully known to the tester. Therefore, the tester should assume a developer perspective, even if they are not the person that actually wrote the code.

Policy for unit testing without mocking:
These straightforward unit tests should be mainly applied to Entities, DTO and Validators/Utils classes. The main goal is to cover each line of code in isolation and check if that line's behavior is working as expected. The only tool that should used is Junit 5.

It is very important that the names of the testing methods correctly describe the intent of each test. The name of the class that contains these tests should end with "Tests". They should be stored in a separate packages called "model", "dto" and "validator".

Policy for Unit testing with dependency isolation:
These tests should be applied to Service classes and Controllers. The main goal is to test their performance with dependency isolation. In other words, the behavior of their external dependencies (different services or repositories) can be controlled by Mockito and thus detached from the code that is meant to be tested independently. Therefore, these external components cannot impact the expected behavior of a unit of code anymore. MockMVC was used to simulate the behavior of an application server for the controllers. Mockito and Junit 5 should also be used.

It is very important that the names of the testing methods correctly describe the intent of each test. The name of the class that contains these tests should end with "Tests". They should be stored in a separate packages called "service" and "controller".

3. System and integration testing

Our integration tests should be written using a white box approach, which means that the inner workings of the implementation code that is being tested are fully known to the tester. Therefore, the tester should assume a developer perspective, even if they are not the person that actually wrote the code. The tester will check if the contents of the JSON response that is being sent after a request to our API match their intent when writing the feature.

Policy for integration testing:
The aim of these tests is to verify the interactions between all the integrated components of the application. For the application pipeline to be working properly, the controllers should be able to interact with the available services and these should make calls to the repositories' methods, without any status failure caused by our end.

MockMVC should be used to simulate the behavior of an application server for the controllers.

A replica of the production database should be used during this stage, so that we can truly check the performance of the database with an EntityManager. New entries should be written purposely for these tests and they should be persisted and flushed to the DB. The test should confirm if these entries are part of the JSON response of the API request to the controller. Before writing the full integration tests that cover the whole system, we will be testing the repository separately in order to detect and fix early mistakes. This approach will make it easier to pinpoint the source of problems.

It is very important that the names of the testing methods correctly describe the intent of each test. The name of the class that contains these tests should end with "IT". They should be stored in a separate packages called "integration" and "repository".

For further API informal testing and documentation purposes, the developer should also set up a Postman collection for the team that contains their features' new endpoints and makes it easy to check the content of the responses. The UI developer can offer a black box perspective and check if the contents of the requests match their needs.

4. Performance testing [Optional]

We have not decided yet if we have a reasonable timeframe to implement performance testing for our API. It will depend on the outcome of Iteration 3.