TQS: Quality Assurance manual

Alexandre Gazur (102751)
Daniel Ferreira (102885)
Emanuel Marques (102565)
Ricardo Pinto (103078)
v06-06-2023

1 Project management

1.1 Team and roles

Alexandre Gazur - Team Leader, PickUs backend and QA
Ricardo Pinto - QA Engineer, PickUs backend and QA
Daniel Ferreira - Product Owner, Web eStore
Emanuel - DevOps Master, PickUs frontend (admin dashboard and ACP dashboard)

1.2 Agile backlog management and work assignment

User stories are defined in our Jira backlog. They may be added, updated or removed as things change (agile). A few of the not implemented stories are prioritized every iteration (1 week). Each user story corresponds to 1 github branch. After the feature is implemented, the branch is pull requested to the main branch, and if it passes the tests and quality gate, the pull request is manually accepted.

2 Code quality management

2.1 Guidelines for contributors (coding style)

In general, we follow the AOSP coding style, since we think it makes a lot of sense and all members like the conventions.

2.2 Code quality metrics

For static code analysis, we use SonarCloud. All pull requests must pass the quality gate to be accepted, as well as all the tests.

We follow the Clean as you Code paradigm and Sonar's default quality gate (but extended to overall code) to ensure no bugs, no security hotspots, at least 80% coverage and minimal technical debt.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

We follow the Github Flow, because it is simple and effective. Developers choose a prioritized user story from our Jira backlog, make a branch for it, implement it, then make a pull request to the main branch.

As for refactoring, some is done at the end of the iteration, and the rest is done in the last 2 iterations (18/05/2023 - 31/05/2023).

Definition of done:

the user story is doable in our system;

the story has tests associated and all of them are passed;

the frontend, backend and database are connected to eachother and behaving as expected.

3.2 CI/CD pipeline and tools

We have a github workflow (yml file in our main repository) with 2 jobs that run on every pull request (and every time that pull request is updated) as well as when something is pushed onto the main branch. The first job runs the Java tests (unit tests, functional tests, integration tests, etc). Then, the second job performs Sonar analysis and checks if the quality gate is passed.

4 Software testing

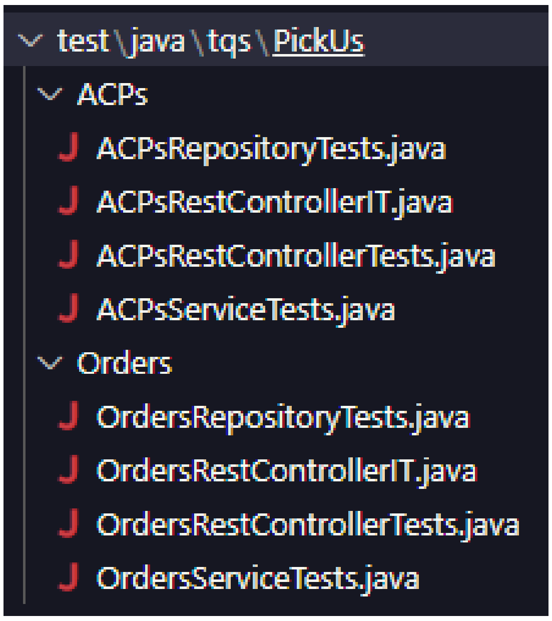
4.1 Overall strategy for testing

We use a mix of 2 strategies: Test After Development (TAD) and Test Driven Development (TDD).

In TAD, we first implement the features and then write tests for it. We chose this strategy because the team members are used to it and have a lot of experience using it. Not only that, we are not very sure of the specific Java code architecture (classes, functions, …) at the beginning, so it becomes hard to use Test Driven Development.

When the QA Engineer has time to work on the project, but features are not yet implemented, or when we know clearly what Java classes and functions are needed for a feature, we use TDD as it has many advantages over TAD.

We use tools like REST Assured, JUnit, Mockito.



```
test\java\tqs\PickUs
  ACPs
    ACPsRepositoryTests.java
    ACPsRestControllerIT.java
    ACPsRestControllerTests.java
    ACPsServiceTests.java
  Orders
    OrdersRepositoryTests.java
    OrdersRestControllerIT.java
    OrdersRestControllerTests.java
    OrdersServiceTests.java
```

4.2 Functional testing/acceptance

We planned on functional testing using Selenium IDE to ensure the management website (admin dashboard and ACP dashboard) is working correctly. However, we didn't have time to test this.

4.3 Unit tests

For unit testing, we use a mix of open box testing and developer perspective testing.

In open box, tests are written during development; this is useful, for example, to make sure a complex Java function is working properly.

For simpler logic, we use developer perspective testing, where tests are written after development. For example, when a developer writes a simple Java function, he may know that, very probably, the function works as expected; tests for this function can be postponed so the developer can focus on implementing the feature.

We tested repositories, services (mocking the repository) and rest controllers (mocking the service). To test ACPs logic, we have 10 unit tests, and to test orders/deliveries logic we have 7 unit tests.

Useful tools for unit testing include JUnit and Mockito.

4.4 System and integration testing

For integration testing, such as the API, we use black box testing.

For example, we execute a GET call, maybe with some URL parameters, and verify that the returned response is as expected. Or, we execute a POST call, and confirm that the database is updated as expected.

For ACPs logic, we have 4 integration tests, and for orders/deliveries logic we have 1 integration test.

Useful tools for integration testing include REST Assured.