TQS: Quality Assurance manual

Ricardo Martins [112876], Rodrigo Jesus [113526], José Pedro [113403], André Vasques [113613]

1 Project management

1.1 Assigned roles

Ricardo Martins - Team Coordinator (keeps everyone aligned, creates most of the sprint events)
André Vasques - Product Owner (maintains backlog, keeps track of what features to prioritize)
Rodrigo Jesus - QA Engineer (defines test strategy, creates and maintains test suites)
José Pedro - DevOps Master (sets up CI/CD, GitHub repo, docker and infrastructure as code)

1.2 Backlog grooming and progress monitoring

Work in JIRA is organized in weekly sprints directly aligned with the practical classes on Thursday and its progress is tracked with the usage of story points and burndown charts.
We also make a proactive monitoring of our code to verify that it indeed meets our defined requirements level coverage wih the help of test management tools integrated in JIRA.

2 Code quality management

2.1 Team policy for the use of generative AI

Our policy is simple about the Do's and Dont's of AI usage:

a) You can use generative AI or other integrated tools in IDEs to help you speed up the process for creating the Drafts for project documentation, basic entities classes, basic unit tests. But then the final implementation must only come after human revision of the generated content.

b) Now in terms of what we consider to be an unacceptable usage of AI is the copy and paste of AI code, especially if we are talking about more important parts of the code like our controllers and services implementations

2.2 Guidelines for contributors

Coding style

c) Our coding style follows the common Google Java Style, which means: 4-space indent, camelCase and also PascalCase for types and classes.

Code reviewing

In order for new code to be reviewed successfully and merged into the main or dev branches it must check the requirements bellow, unless of course we are dealing with unforeseen and urgent situations. The requirements are:

d) All PRs must be reviewed by at least 1 person for merging purposes but preferably 2

e) All available tests must pass and the project must build successfully

f) The SonarQube code coverage must never go bellow our defined value

2.3 Code quality metrics and dashboards

[Which quality gates were defined? What was the rationale?]

As per recommended in our classes we are using SonaQube for static code analysis, and we use SonarCloud to scan every PR to make sure it passes all our target values for Green Quality:

g) Coverage > 80%

h) Maintainability Rating >= A

i) Speed talvez

j) Mais outro valores como "no new problems introduced"

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Coding workflow

Our workflow and branch organization for our project follows the practices we learned in IES classes, which are:

Master:

k) Must be protected so that no one can push directly to it without first creating and getting a pull request approved.

1. This is the final branch where we push the finished files before each Milestone/Sprint delivery.

Hotfix:

1. Created from the master branch.
2. Used when a critical issue needs to be fixed quickly and pushed to master to restore a working version.
3. Branches created for this should follow the format: hotfix/<Initials>/<Issue or problem>
   Example: hotfix/RM/21-navbar-became-invisible ; hotfix/RM-RJ/payment-service-failing

Dev:

1. Created from master branch.
2. Has the same rules as master, which means only push allowed through approved pull requests.
3. This is the integration branch into which we merge our working code towards the final solution.

Release:

1. Created from dev when we are in the final phases of preparing the files for a Milestone release.
2. Once completed, it gets merged into master and then back into dev.

Features:

1. Created from dev.
2. These are the branches where we implement our specific tasks and if possible, they work on a specific defined issue.
3. Branches should follow the format: new/<Initials>/<Issue or problem>
   Example: new/RM/10-create-navbar ; new/AV/11-create-basic-services
4. These branches are merged into dev, not into main.

Tags:

- Every time we finish a release and merge it into master, we must create a version tag using semantic versioning: such as <MAJOR>.<MINOR>.<PATCH> Example: v0.1.0 ; v0.2.0; v1.0.0
- Now in terms of our issues tags they are completely transparent, so if it says frontend then it means working on frontend files, if says docker it means docker files, DB for database modeling and so on.

Definition of done

As specified before with our acceptance criteria for PRs, since each user story related to code opens one or more branches, our definition of done for said user story is going to be much similar to that of PRs, that being:

l) All PRs must be reviewed by at least 1 person for merging purposes but preferably 2

m) All available tests must pass and the project must build successfuly

n) The SonarQube code coverage must never go bellow our defined value

o) The code has been thoroughly tested and the new features fully documented.

3.2 CI/CD pipeline and tools

In terms of our Continuous Integration environment, we are using GitHub Actions to run the following:

p) First making sure to Checkout → Build (mvn clean compile)

q) Run Unit tests + JaCoCo

r) Integration tests (Testcontainers)

s) SonarCloud analysis

t) In a final part, when implemented it will also run Docker build and push to GitHub Packages

Now in terms of Continuous Development it's also possible with the help of GitHub Actions:

- On main merge, deploy Docker Compose stack to dev server.
- On Git tag, publish images as vX.Y.Z and trigger staging deploy.

In progress - As part of the build, .puml diagrams under docs/ are rendered into images using the plantuml-maven-plugin. This ensures diagrams are always in sync with the source code. The generated .png files are included in the Maven site and published with the GitHub Pages documentation.

3.3 System observability

We collect metrics via the Micrometer dependency, store said data in Prometheus and then expose these values to the Grafana Dashboards for better visibility.
Values such as: request rate, number of requests, requests avg duration, percent of 2xx and 5xx requests, total exceptions, error rate, latency percentiles, JVM metrics, CPU usage

In progress - In terms of Alerts, we wa st up some to make sure the system never goes to slow or consumes too much resources from the host without first sending and alert to our monitoring system:

u) Latency of the system never to high

v) CPU usage > 80%

4 Software testing

4.1 Overall testing strategy

Our approach consisted of using a mix of the different testing tools and approaches where we focused more on TDD.
More than half of our Tests are Unit Tests made using the JUnit 5 + Mockito dependencies and which we used mainly for testing and getting coverage for the entities and services. Then a quarter of other tests are Integration Tests using Testcontainers and SpringBoot for in depth testing of our controller classes. Finally, maybe around a tenth of our tests are End2End tests using tools like RestAssured and Selenium, and we used them to guarantee a complete functioning workflow for our entire application.
Then at the end of all this when we already had a functioning MVP we created and used performance testing with the K6 Grafana tool to ensure our solution was not only functional but also efficient

4.2 Functional testing and ATDD

[Project policy for writing functional tests (closed box, user perspective) and associated resources. when does a developer need to develop these?
We are using Xray so...

4.3 Unit tests

Our policy for writing unit tests consists of using Mockito for external dependencies and therefore making sure the tests are independent of external services, and secondly of writing at least 1 unit test for each service and entity.

4.4 System and integration testing

First using Testcontainers to emulate PostgreSQL and mock-payment containers.

Then Integration tests / End2End tests that validate the entire system workflow from the point of finding->booking->paying->using->completed

For the API Testing maybe we could set-up our CI to cover all of the REST endpoints...

4.5 Non-function and architecture attributes testing

To ensure the system meets performance and resilience expectations, as was explained before we adopted the k6 tool for automating load and perform stress testing.
As such we defined several test profiles, located in the `/tests/performance/` folder, each of them covering a specific class for the non-functional behavior we learned in the classes.

w) smoke_test.js: which verifies that all core endpoints respond and that the system is stable, being the test with smallest load and therefore faster execution.
x) average_load_test.js: simulates what we considered to be regular expected traffic our application will have (NUMERO AINDA A DEFINIR 50) to then validate its baseline performance.
y) stress_test.js: now in this type of test we increased the load until we observed system degradation, all with the purpose to determine its max throughput.
z) spike_test.js: a sudden type of test where we send a burst of traffic to observe if the system's resources are able to deal with this sudden spike in load.
aa) soak_test.js: the type of test which takes the longest time to perform as it holds a stable load over a long period, so we can detect if there are memory leaks or slow resource exhaustion over long periods of time.
bb) breakpoint_test.js: finally for this type of test we ramp traffic until the failure threshold is reached, being really useful to help us quantify system limits.

Then in the final stages of the project if possible we would implement in our **CI Integration:**

cc) smoke_test.js and average_load_test.js to run on every PR to dev and main.
dd) stress_test.js and breakpoint_test.js to be executed before each milestone delivery.