TQS: Quality Assurance manual

Afonso Campos [100055], Diana Siso [98607], Isabel Rosário [93343], Miguel Ferreira [98599]
v2022-06-09

1 Project management

1.1 Team and roles

For this project, Isabel Rosário assumed the role of Team Coordinator, Diana Siso was the Product Owner, Miguel Ferreira was the QA Engineer and Afonso Campos filled out the role of DevOps Master. All of the team members worked as Developers.

1.2 Agile backlog management and work assignment

Concerning the agile backlog management practices of the work group, the team is using the Jira tool to keep track of the progress of the multiple tasks that must be completed for the construction of a minimal viable product.

The Team Coordinator created the project board on this platform and, as the project unfolds, creates and edits the tasks as well, ensuring they are each in the correct work column. It is also the Team Coordinator's job to assign the tasks to the various Developers and make sure their progress and completion is going according to the established timeline for the project. It is the responsibility of each

Developer to move the tasks assigned to themselves to the In Progress or Done columns, according to their own work.

The Jira tasks are user story oriented, and it is advisable that each member completes a task before taking on the next one. If one Developer ever takes on too many tasks and fails to complete any of them, it is up to the Team Coordinator to notice and bring attention to the situation, as well as define a plan to bring the organizational system back on track.

The link to the group's Jira board is the following:
https://chateau-du-vin.atlassian.net/jira/software/projects/CHATEAU/boards/1

2 Code quality management

2.1 Guidelines for contributors (coding style)

- Don't catch generic exceptions: it obscures the reason for failure, try to make exception handling as specific as possible.
- Fully qualify imports: use `import foo.Bar;` instead of `import foo.*;`.
- Never use deprecated Java libraries.
- Use Javadoc standard comments.
- Define fields in standard places: define fields either at the top of the file or immediately before the methods that use them.
- Limit variable scope: keep the scope of local variables to a minimum.
- Use consistent indentation.
- Limit line length.
- Use *TODO* comments.
- Use logs – but sparingly.

2.2 Code quality metrics

Since we're using two public repositories, one for the management app and other for the store app, to analyze our code we'll be using Sonar Cloud. The projects will be analyzed individually, through an automatic github action that triggers the analysis for each push and pull request. The quality gate used will be the default Quality Gate set by Sonar Cloud because we think that it meets the project scope.
As this analysis will be run with every push to the repository, the developer will be able to better control the quality of his code, because the "New Code" analysis will be run by Sonar Cloud.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

For the development of this project we decided to adopt a Git Feature Branch Workflow.
With that said, we'll have a ***main*** branch, containing a stable version of the product that will be updated only when there's a new version totally complete and tested, a ***develop*** branch that contains the version of the application that is currently being developed and various ***release*** branches that will contain a version ready to be functionally tested for a specific user-story.
Because we're following a *Feature Branch Workflow*, we'll create a branch everytime we want to develop a new feature for the product.

The branches will be named according to the part of the project we're working (e.g.,*backend-creating_models*, *frontend-creating_home_page*, *deployment-creating_docker_file*, *ci-setting-sonar-cloud, etc*…).
Everytime a feature is done and tested, the developer makes a pull request from the feature branch to the *develop* branch that must be reviewed and accepted by the QA Engineer. When the pull requests are made by the QA Engineer we established that it should be the Team Leader reviewing and accepting the pull requests. When all the features planned for the version being developed are complete and tested, a pull request is made to a release branch.
This branch will be named according to the user story in question, e.g. ***release-manager_sees_all_tasks*** and will contain the full product ready to be functionally tested.
When the product is tested and stable, a pull request is made to the ***main*** branch.
Because the Jira issues are mapped to user stories, they are only complete when a user story is totally complete.
For them to be really completed, they must be fully implemented and tested (a ***release*** must be tested, approved and merged to the ***main*** branch).

3.2 CI/CD pipeline and tools

Since we're using GitHub as our code repository, we'll use Github Actions which is GitHub's own CI/CD platform. For the ManagementApp we will have a *build.yml* file that will trigger two actions:

- Running the ManagementApp Docker Compose (containing the Maven project and the Web App) and launch the Maven tests;
- Generating a Jacoco Report
- Running a code analysis on Sonar Cloud and verify if the code passes the defined quality gate;

For the StoreApp, we'll also have a *build.yml* that will trigger the same actions as the ones described above but this time for the StoreApp Docker Compose file.
These actions will be run every time there's a *push* to the specific repository.
For the CD pipeline we'll have two *deploy.yml* files that will make the project to be deployed and run on the given Virtual Machine through a GitHub Self Runner.

4 Software testing

4.1 Overall strategy for testing

For testing the backend functionalities we start by implementing the feature itself and then writing the tests. Although a TDD would be preferred we found this method to be more efficient for our workflow.
With that said, a feature is developed and then immediately tested. Only then the feature will be merged with the rest of the app.
For this testing we will be using Junit5, Mockito to isolate the components of our app, making it easier to test each component individually, Cucumber to plan real scenarios and Selenium.
To test the User-Interface and the functionality of our app we're using Cucumber and Selenium, as stated above, adopting a BDD.

4.2 Functional testing/acceptance

These tests will be done through the user's perspective. We'll start by planning a set of requirements that each user story has to accomplish and then translate it into a scenario written in Cucumber.

After that, we'll implement the tests based on the Cucumber files by using Selenium and a PageObject Pattern.

4.3 Unit tests

Unit Tests will be made from the developer's perspective to test methods that aren't automatically generated by Java or Spring and need to be used in the product. These tests should test the success and the failure of these methods in order to be sure the product is properly working.

4.4 System and integration testing

Integration testing will be done with the resource of an in-memory database (H2, in this case) and will verify the controllers. The approach we chose will load the full Spring Boot Application but with no API client involved (we'll be using MockMvc). By testing our product with this approach we'll make sure that each component of the project (repository, service, controller and database) are working and relying on each other.