TQS: Quality Assurance manual

Roberto Rolão de Castro [107133]*, *Mariana Figueiredo Perna [108067]*, *Tiago Caridade Gomes [108307]*, *Rafaela Espírito Santo Dias [108782]

v2024-06-04

1 Project management

1.1 Team and roles

- DevOps Master – Roberto Castro
- Team manager – Mariana Perna
- QA Engineer – Tiago Gomes
- Product Owner – Rafela Dias

1.1 Agile backlog management and work assignment

The methodology follows a standardized life cycle for managing user story issues, strictly using Jira. The stages of this cycle are detailed below:

1. **Story Writing**: At the beginning of each iteration, or when a team member completes their tasks, new user stories are written and added to the project backlog. This step is crucial to maintain a continuous flow of work and to ensure all project requirements are properly documented.

2. **Task Distribution**: The created issues are then assigned to team members, ensuring an equitable distribution of work and that all necessary skills are engaged across different parts of the project.

3. **Pull Request Review**: Upon task completion, the responsible team members create a pull request on GitHub. This pull request undergoes a review process where other developers in the team comment, approve, or request changes. This iterative process continues until the code is deemed of sufficient quality by peers.

4. **Continuous Integration (CI) Testing**: Before a pull request can be merged, it must pass all Continuous Integration tests. This ensures that the new code does not negatively impact the existing codebase and meets all predefined quality and performance benchmarks.

5. **Integration and Issue Closure**: Once the pull request passes the CI tests and is approved, it is merged into the main development branch, and the corresponding issue in Jira is closed.

2 Code quality management

2.1 Guidelines for contributors (coding style)

2.1.1 Implementation of Custom Exceptions

In our software development efforts, we have adopted the use of custom exceptions to enhance error handling and ensure greater clarity in the way our application responds to specific conditions and errors. Custom exceptions allow us to precisely define how our application reacts to various exceptional scenarios, thereby improving the maintainability and reliability of our code.

2.1.2 Standardization of Testing Procedures

The naming convention for tests, structured as ***test<functionality>_when<case being tested>_then<expected outcome>***, facilitates understanding and maintaining the test suite. Enhancements can be made by:

- **Expanding Error Scenario Coverage**: Incorporate tests that validate the handling of defined custom exceptions.
- **Integrating System-Level Tests**: Include tests that evaluate the interactions between different components, particularly focusing on error management.
- **Employing Isolation Techniques in Unit Testing**: Utilize mocks and stubs to ensure each test is focused and independent, enhancing test reliability.

2.1.3 Utilization of Data Transfer Objects (DTOs)

DTOs are crucial for data encapsulation and ensuring separation of concerns between presentation layers and domain models, which enhances security and data management.

2.2 Code quality metrics and dashboards

For the purpose of ensuring that the quality of the code produced is of high standard, the team employs the use of SonarQube which allows for strict compliance with code quality standards. Some examples of real-time analysis metrics of SonarQube are cyclomatic complexity, code coverage, duplications, and maintainability. It is fully automated and is integrated within our Continuous Integration pipeline where it checks through every code commit that is made and only gives the go ahead after approving the quality of code committed. This also minimizes future problems which might occur after the deployment and the overall security of our application is enhanced.

Having the option of the real-time feedback on the dashboard of our SonarQube enables our team to solve the problem promptly to ensure that we maintain the quality as required. SonarQube plays an important role in estimating the quality of our software and helps to ensure that quality is intact through the provided reports and trends in quality as measured parameters.

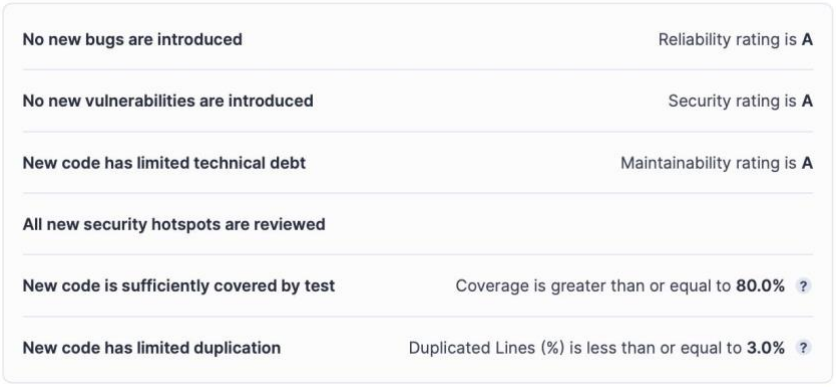

Figure 1: Quality Gates – SonarQube

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

Our development workflow contains GitFlow and Jira and performs as a role model for the project management and code integration. This helps to ensure that proper transitions from feature development to a product development environment can be made.

Workflow Steps:

1. **Feature Branching:** Every feature gets a new branch from the develop branch for unified tracking with a particular Jira ticket.

2. **Development and Commit:** To merge into the local branch, they type ‘git merge HEAD‘, thus incorporating the develop branch’s latest changes into the branch.

3. **Pull Requests:** Features that are implemented go through reviews through Pull Request that mentions the literals Jira ticket.

4. **Automated Testing:** Pull requests work as a basic comparable system that launches CI tests for quality confirmation.

5. **Merge:** Once a feature has been built, tested and reviewed the content is integrated back into develop and the Jira ticket is closed.

6. **Release Preparation**: The associated develop branch is then branched into or promoted into release for added edits and testing.

7. **Production Deployment:** The stable releases are then merged into the main branch and are sent to deployable production environments.

Definition of Done:

1. Peer-reviewed and approved code.

2. Passed all automated tests.

3.2 CI/CD pipeline and tools

This way, the testing and the deployment are set free by the CI/CD pipeline controlled by GitHub Actions hence enhancing the rate at which new software is developed.

CI Configuration:

In Java CI, a Maven based toolchain is used and is set to be initialized on pushes and pull requests to the dev and main branches. The configuration involves:

1. **Automated Builds and Tests:** This can be where the concept of continuous integration is highly effective in the sense that it ensures that the quality of code is kept high as the developers work on different projects.

2. **Dockerization:** Hence the process of deploying applications in a package or container for systematic use across environments.

CD Configuration:

1. **Build and Containerization:** This means that the application is built and shipped as a Docker format, as Docker images are often used.

2. **Registry Authentication:** However, to push or pull a Docker image, the authentication must be done through GitHub Packages interface.

3. **Deployment Execution:** In the case of Docker Compose, only the latest versions are used in a personally hosted Virtual Machine hence minimizing on time required to synchronize and maintain the general health of the services.

There are certain CD processes that are automated processes, and these include building of Docker images, pushing and deployment of Docker images via GitHub actions and this has enhanced how we work.

3.3 System observability

To ensure optimal application performance and reliability, we utilize Swagger for API management and documentation:

- **API Documentation with Swagger:** API Creation & Documentation: This tool plays a significant role in defining, creating and documenting restful APIs. Leveraging on the API documentation that Swagger provides which is quite clear and creates an interactive documentation that makes it easy for the developers and QA testers to actually understand how the API should be tested on the various endpoints. API services and their documentation are critical to backend development and testing, and this language helps guarantee that all of them follow the standards that will be easily understood by developers.

Thus, applying these tools and techniques, implemented in our CI/CD pipeline, guaranteed an efficient integration process, exhaustive testing, and cloud-based reliable depoware. com-oyntment of the production-grade software as well as intelligent API management with the help of Swagger.

4 Software testing

4.1 Overall strategy for testing

The rationale of our testing approach is BDD for the frontend and TDD where feasible for the rest of the application. This approach provides end-to-end testing that covers the user and developer viewpoints.

- **BDD for Frontend:** We employ Cucumber for BDD whereby we are able to write scenarios using natural language. This makes certain that the frontend acts in the appropriate manner and is able to fulfill the business needs. Also, Selenium is used for the browser automation for confirming that the interface works well in browsers and other environments.

- **TDD for Backend:** Our goal was to use TDD whenever possible, which is a good practice to follow. Although, there were cases of failure in the application of the TDD, it was used to write tests before developing codes wherever possible. This helped in identifying problems at the early stage and ensuring that the quality of the code is maintained.

4.2 Functional testing/acceptance

To achieve the optimum performance of web applications and the associated REST APIs, our functional testing and acceptance plan is carefully devised. Using Selenium in conjunction with browser automation, we emulate complex user inputs in order to thoroughly exercise the UI under different scenarios. This detailed process allows us to discover and solve possible glitches that might affect the audience, thus guaranteeing that the apps function correctly on all devices.

To this end, we leverage TestContainers in building realistic and controlled test environments. These environments are helpful when determining the functionality of REST APIs that are central to the applications' function. It is seen that TestContainers help to mimic various servers and network conditions and can be used to test APIs individually or combined with web applications. Such a two-pronged approach allows us to ensure that both the layers exposed to the user as well as the back-end of our applications are as solid as they can be before deployment.

4.3 Unit tests

As mentioned earlier, in the CliniConnect project, and in order to achieve maximum test coverage, strict unit testing was carried out to check the functionality of each component separately and to inspect how each of them worked when it was in isolation from the other components.

Models Layer: Testing and presenting the data models ensured that all models adhered to their outlined business rules and struck out to ensure that the attributes of the models function appropriately under different circumstances.

Persistence Layer: It allowed to define @DataJpaTest that supplied a controlled environment with TestContainers for more comprehensive testing. This made it possible to perform a thorough analysis of not only the methods identified within an interface but also of the frequently used methods and correct behavior in usual and edge cases.

Services Layer: Mockito was very helpful in test-driven approach of the services layer since it made it possible to mock the interactions of the repositories. This setup was centered at identifying business logic test cases while giving a lot of emphasis on how to manage test cases during error conditions.

Controllers Layer: End to end testing of the controller operations was done using MockMvc for generalized testing and using RestAssuredMockMvc for specialized testing. They employed this methodology ensured end-to-end testing while excluding the managed services from the testing, and at the same time checked for errors which could occur during the operation of the controller.

It was such a structured regime in testing that was important in assuring the propriety operation of each layer in CliniConnect system while still on deeming high standard of reliability and quality.

4.4 System and integration testing

In the CliniConnect project, we have laid down a robust system and integration testing plan and approach that helps us test all the components of our system so that they can work perfectly fine and as required. It is done in a closed and open box testing approach as mentioned earlier, the combination of CBT and PBT put into use contributes to the accomplishment of the above goal. In closed box testing, we tend to test the system from outside the organizational boundaries and the test results are based on the observable behavior of the system and not of the underlying structure or architecture of the system. On the other hand, open box testing enables the testers to do exhaustive probing to investigate the internal inter-module interactions.

In API testing, we use the basic utilities like the Postman and Swagger. These tools ensure that all RESTful services developed for a particular application are compliant with their specifications and are capable of handling different types of operations without compromising security. In that regard, we are running the automated API tests separately as a part of the CI/CD pipeline, which means that they are activated every time a new build is initiated. Moreover, it is advisable to incorporate Swagger for the contract testing process; thus, guaranteeing compliance to API descriptions.

They are performed as an integrational test as a part of CI/CD practice on a regular basis. This frequent check-and-balancing allows us to intervene early whenever specific problems with the integration of various system sub-programs occur. This frequent and comprehensive testing is vital in guaranteeing the optimal performance and stability of the CliniConnect system, thus providing excellent healthcare solutions.