deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

TQS: Quality Assurance manual

Gonçalo Silva [103668], João Matos [103182], Pedro Rasinhas [103541]
v2020-05-25

1 Project management

1.1 Team and roles

1.2

Team Member	Role	Description
Pedro Rasinhas	Team Manager & QA Engineer	Make sure that jobs are distributed fairly and that everyone follows the plan. Encourage teamwork at its finest and take the initiative to handle any potential issues that may develop. Make sure the requested project results are delivered on schedule. Ensure that the quality assurance practices are put in practice and that instruments to measure the quality of the deployment are used. Monitors that team follows agreed QA practices.
João Matos	Product Owner	Represents the interests of the stakeholders. Has a deep understanding of the product and the application domain; the team will turn to the Product Owner to clarify the questions about expected product features.Should take part in accepting the solution increments.
Gonçalo Silva	DevOps	Responsible for the development and production infrastructure and required configurations. Ensures that the development framework works properly. Leads the preparation of the deployment machine(s)/containers, git repository, cloud infrastructure, databases operations, etc.
ALL	Developer	ALL members contribute to the development tasks.

1.3 Agile backlog management and work assignment

We utilize Jira as our project management tool for efficient task and job tracking, as well as seamless assignment to team members. Jira is an agile platform designed to facilitate the management of project-related tasks and issues, while also enabling effective project planning. It offers various features such as task categorization into Sprints, organization through Epics, and the ability to set deadlines, among others. With Jira, we can streamline our workflow and ensure smooth progress throughout the project.

2 Code quality management

2.1 Guidelines for contributors (coding style)

Implementing a standardized coding style in our projects greatly enhances code readability and comprehension, especially when collaborating with partners. By adopting a consistent pattern within the team, we can minimize the time required for team members to modify and update code written by others, when necessary.

Considering that our projects primarily involve two distinct programming languages, Java and JavaScript, each with their unique characteristics, we will employ separate conventions for each language. This approach allows us to cater to the specific requirements and best practices associated with Java and JavaScript, ensuring code consistency and maintainability across the board.

For Java, used along our backend, we decided to adopt the Oracle Java Code Convention. For JavaScript, used on our frontend we decided to use the Google JavaScript Style.

2.2 Code quality metrics

To enhance our code quality assessment and improvement processes, we have adopted SonarQube. SonarQube is a powerful platform that conducts static code analysis on our repositories, providing insights into potential issues and areas for enhancement. It effectively identifies security vulnerabilities and offers recommendations to rectify them.

In order to maintain code readiness for production and ensure stability, we will adhere to the recommended quality gate provided by SonarQube. This gate encompasses several key metrics, including:

- Issues: below a threshold of 20%.
- Complexity: below 10% to ensure code simplicity and ease of maintenance.
- Duplication: below 10%, reducing redundancy and promoting maintainability.
- Coverage: above 60% to ensure adequate testing and minimize the risk of undiscovered defects.

By following these recommended metrics, as suggested by SonarQube, we can ensure that our code remains production-ready and meets the required standards of quality and stability.

3 Continuous delivery pipeline (CI/CD)

3.1 Development workflow

To ensure that all developed code adheres to requirements, undergoes thorough review, and does not introduce any issues to the existing codebase, we have decided to implement the GitHub Flow methodology. This approach provides guidelines on the development workflow for new features, outlining the necessary steps that each feature should follow until it reaches its final version in the product. Our choice was based on us being a small team, so we don't really need to work with multiple production versions, therefore we can have only a single *main* branch that is the latest working version that can be deployed. This workflow has a few **must follow** steps:

1. **To perform a new task, a new branch is created from the master branch and is explicitly named to reveal its purpose**

   This step allows the developers to perform their job in a controlled, stable, and safe environment. This helps the developers who are developing the feature to be sure no one will interfere with their job, as well as gives an opportunity for other developers to review the job that's being done on that particular user story.

2. **Code changes must be committed to the local branch as frequently as possible.**

   This step means that once the branch for a new user story is created, all the code related to that user story must be committed to this branch, as frequently as possible.

3. **Initiate a pull request to request code review**

   Once the feature is completed, a pull request needs to be open to the *main* branch. Each Pull Request should include a self-describable title and a short description including:

   - The changes that the Pull Request will add;
   - A direct connection to the JIRA user story.

4. **Merge the pull Request**

   To ensure a controlled and reviewed integration of code changes, our workflow requires at least one approval for a Pull Request before it can be merged. The specifics of the checks will be discussed in a later section of this document.

5. **Delete the branch**

After the Pull Request is accepted and merged into the main branch, the branch that made the pull request (the feature/fix/… branch) should be deleted, to avoid branch pollution on the repository.

For each user story it's necessary to exist a **Definition of Done**, that helps understand at what point in time and after which steps a user story is actually **Done (ready to release)**. The steps we consider are the following:

1. **Fulfillment of acceptance criteria:**
   The user story should meet all the specified acceptance criteria, ensuring accurate implementation of the desired functionality.

2. **Code implementation and review:**
   The code changes related to the user story must be implemented and reviewed for quality and compliance with coding standards.

3. **Passing unit tests:**
   Unit tests specific to the user story should be written and pass successfully to ensure the correctness of the implemented code and detect any regressions.

4. **Successful integration tests:**
   Relevant integration tests should be conducted to ensure seamless integration of the user story's functionality with other system components, avoiding unintended issues.

5. **The feature is deployed in the production environment**

3.2 CI/CD pipeline and tools

To handle the CI/CD pipelines, we used GitHub Actions to manage the workflows.

The CI workflow:

```yaml
name: SonarCloud
on:
  push:
    branches:
      - main
  pull_request:
    types: [opened, synchronize, reopened]
jobs:
  tests-and-sonarcloud:
    name: Build and analyze
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
        with:
          fetch-depth: 0  # Shallow clones should be disabled for a better relevancy of analysis
      - name: Set up JDK 17
        uses: actions/setup-java@v3
        with:
          java-version: 17
          distribution: 'zulu' # Alternative distribution options are available.
      - name: Cache SonarCloud packages
        uses: actions/cache@v3
        with:
          path: ~/.sonar/cache
          key: ${{ runner.os }}-sonar
          restore-keys: ${{ runner.os }}-sonar
      - name: Cache Maven packages
        uses: actions/cache@v3
        with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2
      - name: Run Docker compose file
        run: docker-compose -f docker-compose.yml up -d
      - name: Wait for MySQL to start
        run: |
          while ! docker exec loadconnect-db curl -s http://localhost:3306 > /dev/null; do
            sleep 1
          done
      - name: Build and analyze
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  # Needed to get PR information, if any
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=TQSProject23_core_backend
```

This workflow allowed us to validate the pull request and then merge into the main branch, by giving us the information on the static code analysis.
When talking about a CI configuration file there are multiple sections and steps: Event triggers, Jobs, and steps. This workflow is triggered by two events, push and pull request actions. The jobs are the instructions that the runner has to execute, this execution being made, in this case, on an

Ubuntu-based environment.Steps are individual actions or tasks performed sequentially during CI workflow.

In our case, this CI configuration ensures that whenever code changes are pushed or pull requests are made, the project's code will be built, tested, and analyzed using SonarCloud. The results of the analysis can then be reviewed in the SonarCloud platform to identify code quality issues, security vulnerabilities, and other metrics that help improve the overall codebase.

The CD workflow:

```yaml
  deploy:
    name: Deploy
    needs: [build]
    runs-on: ubuntu-latest
    environment:
      name: 'production'
      url: ${{ steps.deploy-to-webapp.outputs.webapp-url }}

    steps:
      - name: Download artifact
        uses: actions/download-artifact@v2
        with:
          name: core_backend

      - name: Deploy to Azure Web App
        id: deploy-to-webapp
        uses: azure/webapps-deploy@v2
        with:
          app-name: 'loadconnect'
          slot-name: 'production'
          publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE }}
          package: '*.jar'

```

```yaml
  build:
    name: Build
    runs-on: ubuntu-latest
```

This workflow allowed us to keep the deployment active and updated, make updates to the deployed version on each push to the main branch.

These two jobs define the steps required to build a Java application using Maven, create an artifact, and then deploy it to an Azure Web App.
The artifact is created in the "Build" stage, and the deployment to the Azure Web App occurs in the "Deploy" stage. This ensures a streamlined and automated process for building and deploying the Java application.

For our frontend we used Firebase, which is a platform for developing web and mobile applications but we only used it for the deployment of our React websites.

This CI/CD configuration is triggered when something is pushed to the “main” branch and automatically builds and deploys the project to Firebase Hosting using our specified Firebase project and service account credentials. It enables a streamlined deployment process for the project, ensuring that the latest changes are reflected on the live hosting environment.

```yaml
# This file was auto-generated by the Firebase CLI
# https://github.com/firebase/firebase-tools

name: Deploy to Firebase Hosting on merge
'on':
  push:
    branches:
      - main
jobs:
  build_and_deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: npm ci && CI=false npm run build
      - uses: FirebaseExtended/action-hosting-deploy@v0
        with:
          repoToken: '${{ secrets.GITHUB_TOKEN }}'
          firebaseServiceAccount: '${{ secrets.FIREBASE_SERVICE_ACCOUNT_LOADCONNECT_AD7D6 }}'
          channelId: live
          projectId: loadconnect-ad7d6
```

4 Software testing

4.1 Overall strategy for testing

On the backend we opted to use a TDD (Test-Driven Development) approach which means that we first create the tests for each component and then we implement and develop the code for the respective component until all the tests pass. For the unit testing we opted to use JUnit5, Hamcrest, Mockito and SpringBoot MockMvc. After the successful implementation of the unit tests, in each isolated module, we move on to the integration tests, and verify their success, always following the same approach, if they fail, we fix them until they work, For the integration tests, we decided to use JUnit5 and REST-Assured frameworks.

For the front-end tests, we used BDD (Behaviour-Driven Development) to improve teamwork and guarantee that the development process complies with the intended functionality and results of our application. BDD entails writing executable specifications that can be understood by both technical and non-technical stakeholders and are written in a human-readable manner. We may close the communication gap between developers, testers, and business stakeholders by adopting BDD. This will lead to a common understanding of the anticipated behavior and enable more efficient testing and development procedures.

4.2 Functional testing/acceptance

Functional testing plays a vital role in software development by validating and ensuring that specific features of an application function correctly and meet the expected requirements. The Selenium framework allows us to simulate user interactions and assess if the application responds as anticipated during testing. Since some test cases were created before the actual development of the application, without detailed insights into its internal workings, we relied on black-box functional testing. Additionally, as developers, we took the perspective of end-users to evaluate if the application performed as expected in various scenarios, thus adopting user viewpoint functional testing. To enhance test organization, reduce redundancy, and improve code readability, we implemented the Page Object Model pattern.

4.3 Unit tests

White box testing entails closely examining the source code to look at how software functions internally. With this method, testers can produce unit tests that specifically target problems with complex logic, inconsistent programming, and flawed data structures. In developer-perspective unit testing, developers create and run the tests based on requirements and anticipated results. This testing strategy's primary goals are to achieve thorough code coverage and avoid the later stages of the software development lifecycle from discovering component problems. The advantages of both methodologies were successfully combined in our unit testing strategy.

For unit tests, we initially adopted a Test-Driven Development (TDD) strategy. After determining the robustness and initial failure of the tests prior to component implementation, developers defined their

desired outcomes for a specific component and then continued to develop the component code until all unit tests passed successfully. The QA engineer meticulously examined the source code to find any potential untested issues during the pull request phase after all of the developer-authored unit tests had passed. These particular code anomalies were addressed by creating additional tests, if necessary.

Unit tests mainly concentrated on testing the service layer by simulating the controller layer and the service layer by mocking the repository layer. We used a number of frameworks, including JUnit Jupiter, Hamcrest, Mockito, and Spring Boot MockMvc, to complete these assessments. Using the Jacoco and SonarCloud frameworks, we evaluated the quality of the code and the test coverage.

4.4 System and integration testing

Once all unit tests have been successfully executed and each component has been completed, the integration of these components into the larger system becomes necessary. Integration tests prove invaluable in simulating the real-world interactions between units, providing a glimpse into how they operate within the context of the entire application. The integration testing approaches of open box and developer-perspective align with the earlier explanations, where tests are crafted by examining the code and developers script them based on their expectations and feature requirements, respectively. In contrast, black box integration tests do not require in-depth knowledge of the code or the underlying system intricacies. Instead, they focus on assessing system response time, verifying expected function outcomes, evaluating usability, and troubleshooting potential issues.

Our integration tests primarily focused on examining the interplay between the controller, service, and repository layers. This involved testing the interactions between the controller and service layers while simulating the repository, scrutinizing the interactions between the service and repository layers, and evaluating the comprehensive interaction among the controller, service, and repository layers. To conduct these integration tests, we heavily relied on frameworks such as JUnit Jupiter, Hamcrest, and RestAssured.