TQS: Quality Assurance Manual

Made by:

Guilherme Amorim, 107162;

Rafael Ferreira, 107340;

José Gameiro, 108840;

Ricardo Quintaneiro, 110056

Contents

1. Product Management

1.1 Team and Roles

The team roles were assigned from a qualitative perspective of every team member. After the analysis of the individual characteristics, we settled on the following configuration:

Product Owner	**QA Engineer**	**DevOps**	**Team Coordinator**
Guilherme Amorim	Rafael Ferreira	José Gameiro	Ricardo Quintaneiro

The team roles were assigned based on an analysis of each member's strengths. The following configuration was established:

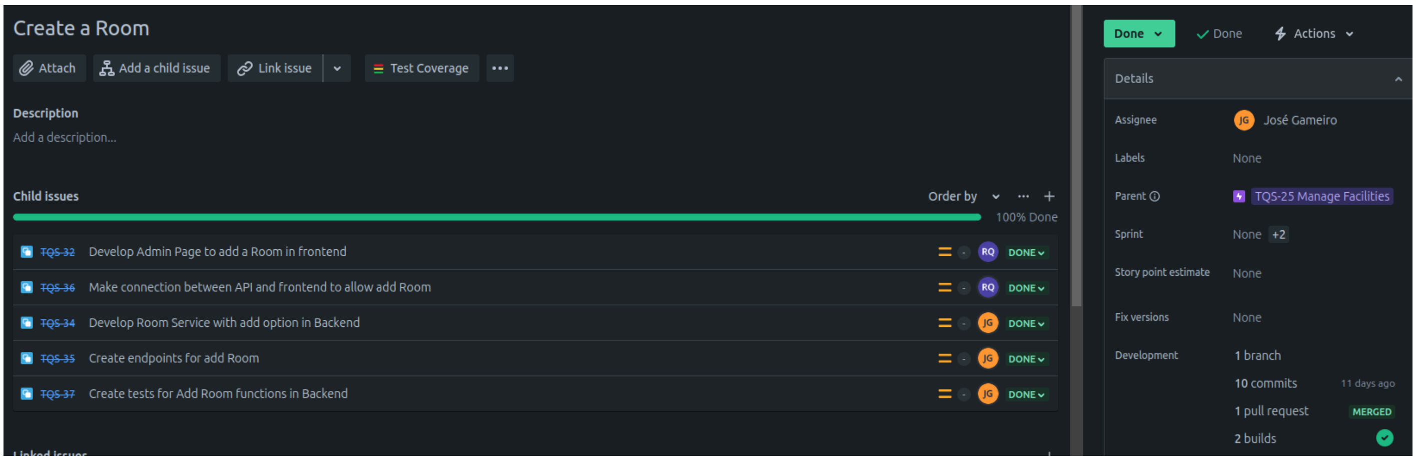
- Product Owner: Knowledgeable about the product's development and key features.
- QA Engineer: Set guidelines and rules to ensure code quality and consistency.
- DevOps: Automate and unify processes, focusing on testing and deployment.
- Team Coordinator: Distribute work evenly and ensure adherence to schedules.

Additionally, all members were also responsible for development tasks, adhering to code conduct and ensuring their work met the standards required for pull requests.

1.2 Agile backlog management and work assignment

The team made use of Jira for backlog management. Within Jira, we created epics that encompassed a wide variety of user stories designed to meet the product owner's requirements. Each user story included smaller subtasks that were assigned to team members based on their expertise and availability.

The process followed a structured workflow. During each sprint, the team focused on tackling an epic, ensuring that the user stories were well-organized within the sprints. This organization was crucial for adhering to Scrum methodology. Sprint planning meetings were held at the start of each working week to review and prioritize the user stories to be implemented. In these meetings, the team discussed the scope of work, set clear goals for the sprint, and ensured that everyone was aligned on the tasks ahead. This approach facilitated effective collaboration and helped the team maintain a steady progress towards the project goals.



Create a Room

Attach | Add a child issue | Link issue | Test Coverage

Description

Add a description...

Child issues — Order by — 100% Done

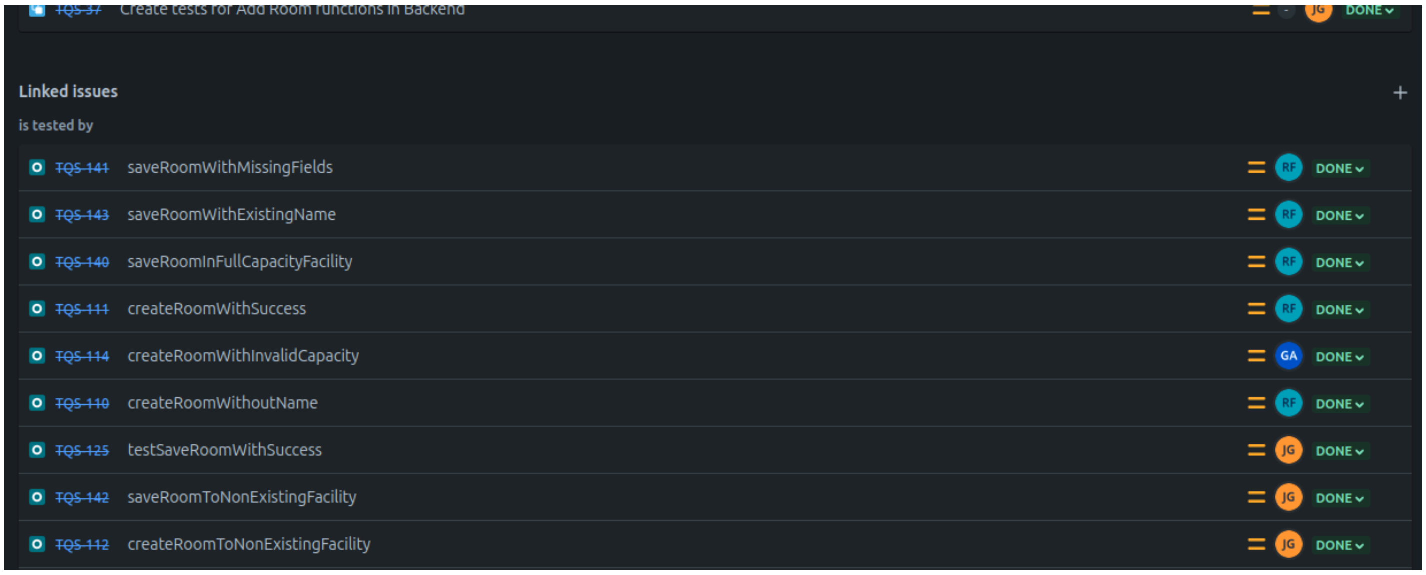
~~TQS-32~~	Develop Admin Page to add a Room in frontend	RQ	DONE
~~TQS-36~~	Make connection between API and frontend to allow add Room	RQ	DONE
~~TQS-34~~	Develop Room Service with add option in Backend	JG	DONE
~~TQS-35~~	Create endpoints for add Room	JG	DONE
~~TQS-37~~	Create tests for Add Room functions in Backend	JG	DONE

Done — ✓ Done — Actions

Details

Assignee	José Gameiro
Labels	None
Parent	TQS-25 Manage Facilities
Sprint	None +2
Story point estimate	None
Fix versions	None
Development	1 branch
	10 commits 11 days ago
	1 pull request MERGED
	2 builds ✓

Fig.1a) Example of a user story created with subtasks defined and an epic associated



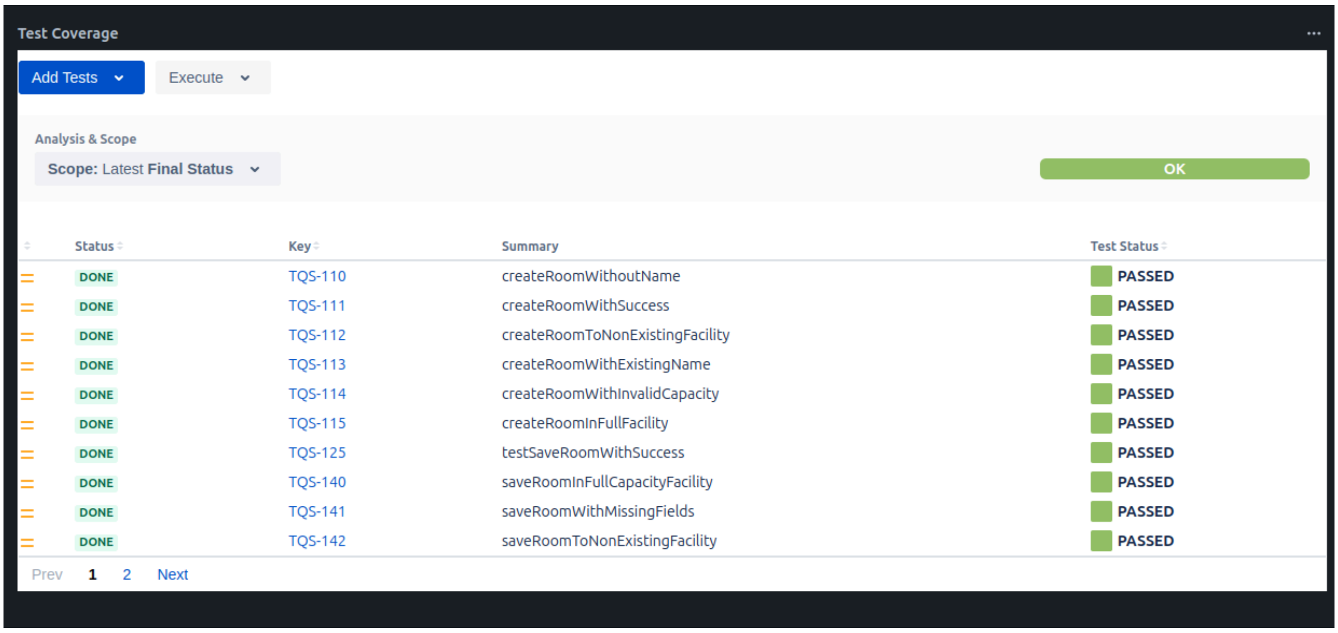
~~TQS-37~~ Create tests for Add Room functions in Backend JG DONE

Linked issues

is tested by

~~TQS-141~~	saveRoomWithMissingFields	RF	DONE
~~TQS-143~~	saveRoomWithExistingName	RF	DONE
~~TQS-140~~	saveRoomInFullCapacityFacility	RF	DONE
~~TQS-111~~	createRoomWithSuccess	RF	DONE
~~TQS-114~~	createRoomWithInvalidCapacity	GA	DONE
~~TQS-110~~	createRoomWithoutName	RF	DONE
~~TQS-125~~	testSaveRoomWithSuccess	JG	DONE
~~TQS-142~~	saveRoomToNonExistingFacility	JG	DONE
~~TQS-112~~	createRoomToNonExistingFacility	JG	DONE

Fig.1b) Associated tests to the user story mentioned in the previous image



Test Coverage

Add Tests | Execute

Analysis & Scope

Scope: Latest **Final Status** — OK

Status	Key	Summary	Test Status
DONE	TQS-110	createRoomWithoutName	PASSED
DONE	TQS-111	createRoomWithSuccess	PASSED
DONE	TQS-112	createRoomToNonExistingFacility	PASSED
DONE	TQS-113	createRoomWithExistingName	PASSED
DONE	TQS-114	createRoomWithInvalidCapacity	PASSED
DONE	TQS-115	createRoomInFullFacility	PASSED
DONE	TQS-125	testSaveRoomWithSuccess	PASSED
DONE	TQS-140	saveRoomInFullCapacityFacility	PASSED
DONE	TQS-141	saveRoomWithMissingFields	PASSED
DONE	TQS-142	saveRoomToNonExistingFacility	PASSED

Prev 1 2 Next

Fig1c) XRay results of the associated tests

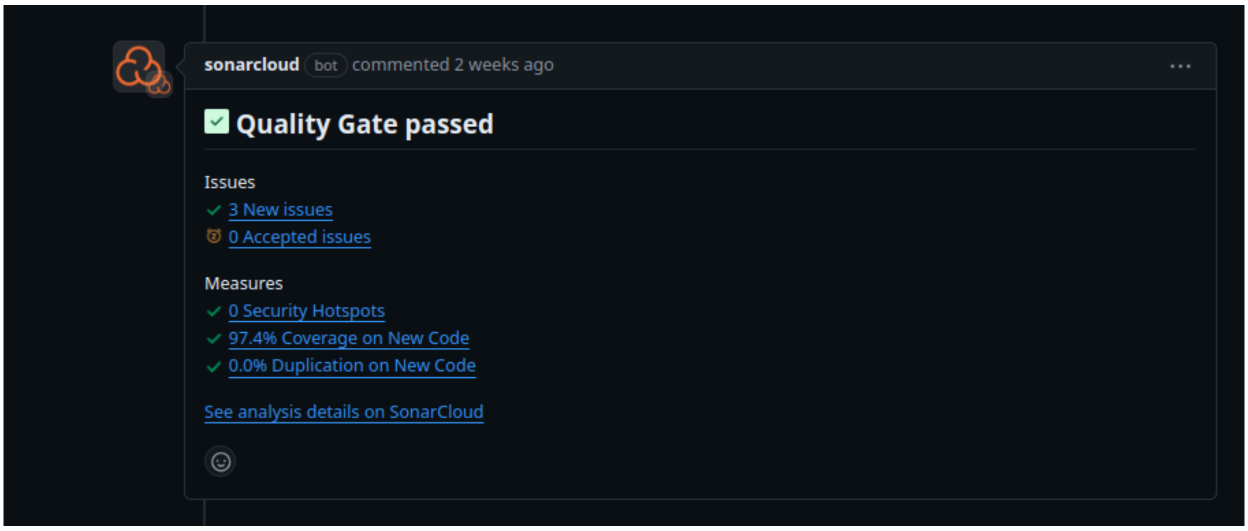
2. Code Quality management

2.1 Guidelines for contributors (coding style)

The team follows the coding convention as established by the AOS project. This includes guidelines on code formatting, naming conventions, and documentation standards.

2.2 Code quality metrics

The team used strict rules for code contribution, assuring the necessary code quality. These metrics were: a minimum of eighty per cent of code covered by tests; no security issues detected by the sonarcloud analysis; all the created tests must be passing and have been created before the code; and last but not least, their pull requests must be reviewed by at least one of the other team members.


sonarcloud bot commented 2 weeks ago

Quality Gate passed

Issues
✓ 3 New issues
0 Accepted issues

Measures
✓ 0 Security Hotspots
✓ 97.4% Coverage on New Code
✓ 0.0% Duplication on New Code

See analysis details on SonarCloud

Fig2. Example of a Pull Request with the Quality Gate passed

3. Continuous delivery pipeline (CI/CD)

3.1 Development workflow

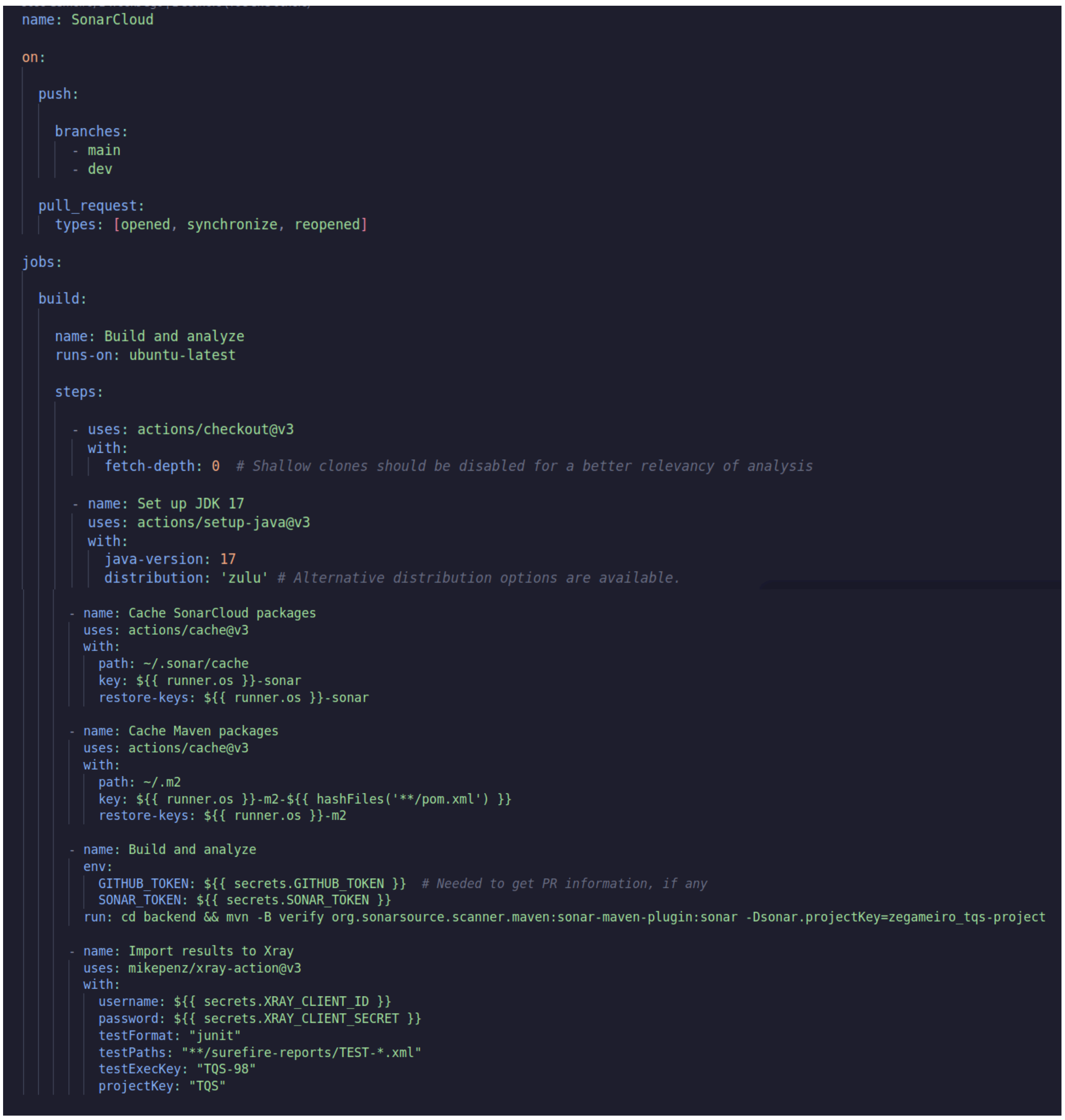
The team follows the GitFlow workflow for version control using GitHub. Feature branches are created for each user story, and pull requests are submitted for code review before merging into the respective epic branch and finally into the *develop* branch. Once this branch hits a stable and desirable configuration it gets pulled requested into the *main*.
The code reviews happen as part of the pull request process to ensure code quality and adherence to coding standards.

3.2 CI/CD pipeline and tools

In our project, we have implemented a robust Continuous Integration (CI) and Continuous Delivery (CD) pipeline to ensure the continuous integration of increments and the seamless deployment of software.
Continuous Integration is a key practice in our development process, allowing us to detect and address issues early by frequently integrating code changes into the development branch. Our CI pipeline is automated using GitHub Actions.

We have defined two primary GitHub Actions workflows:

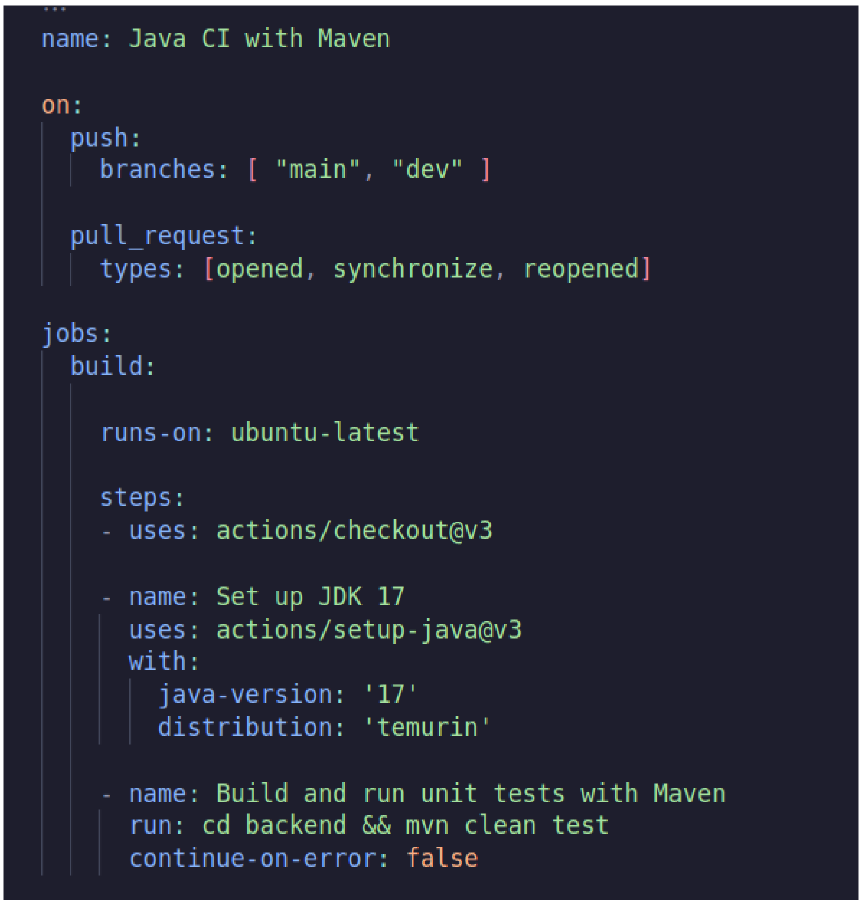
1. **SonarCloud Analysis and XRay Test Results Import:** This workflow focuses on code quality and test coverage analysis. It is triggered by push events to the main and dev branches, as well as by pull requests that are opened, synchronised, or reopened.



```yaml
name: SonarCloud

on:

  push:

    branches:
      - main
      - dev

  pull_request:
    types: [opened, synchronize, reopened]

jobs:

  build:

    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: Build and analyze
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  # Needed to get PR information, if any
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: cd backend && mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=zegameiro_tqs-project

      - name: Import results to Xray
        uses: mikepenz/xray-action@v3
        with:
          username: ${{ secrets.XRAY_CLIENT_ID }}
          password: ${{ secrets.XRAY_CLIENT_SECRET }}
          testFormat: "junit"
          testPaths: "**/surefire-reports/TEST-*.xml"
          testExecKey: "TQS-98"
          projectKey: "TQS"
```

Fig.3 Github action to run a coverage analysis and the existing tests and send those results respectively to the project in SonarCloud and to the XRay

2. **Java CI with Maven:** The second workflow focuses on building the project and running unit tests. It is also triggered by push events to the main and dev branches, as well as by relevant pull request events.



```yaml
...
name: Java CI with Maven

on:
  push:
    branches: [ "main", "dev" ]

  pull_request:
    types: [opened, synchronize, reopened]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up JDK 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'

    - name: Build and run unit tests with Maven
      run: cd backend && mvn clean test
      continue-on-error: false
```

Fig.4 Github action to run the existing tests

4. Software Testing

4.1 Overall strategy for testing

The team practised Test-Driven Development (TDD), a methodology where tests were written before the code implementation began. This approach ensured incremental code development with a strong emphasis on meeting specific requirements, all while avoiding the creation of tests that were deliberately designed to pass. This disciplined method helped maintain code quality and reliability from the outset. Additionally, the team adopted Behaviour-Driven Development (BDD) to enhance user story verification. BDD emphasised writing tests in a natural language that described the desired behaviour of the application, thus ensuring that the software not only met functional requirements but also aligned with user expectations and workflows. Lastly, the team implemented a comprehensive suite of Integration Tests (IT). These tests were crucial for verifying that different modules and components of the application worked together seamlessly, ensuring the integrity and consistency of actions across the entire system.

4.2 Functional testing/acceptance

Functional tests were automated using Selenium WebDriver, which simulated user interactions with the application as if a real user were performing them. This automation helped ensure that the user interface behaved correctly under various conditions. Test scenarios were meticulously defined in Cucumber feature files, which followed Behavior-Driven Development (BDD) principles. These feature files were written in plain language, making them accessible to both technical and non-technical stakeholders. By doing so, the team ensured that the application met the intended user requirements and delivered a smooth user experience.

4.3 Unit tests

Unit tests formed a core part of the TDD process. Written in Java using the JUnit framework, these tests focused on verifying the functionality of individual components and classes in isolation from the rest of the system. This isolation allowed for thorough testing of each component's logic and behaviour, ensuring that each part of the codebase functioned correctly on its own. The rigorous unit testing process helped catch defects early in the development cycle, reducing the likelihood of bugs in later stages.

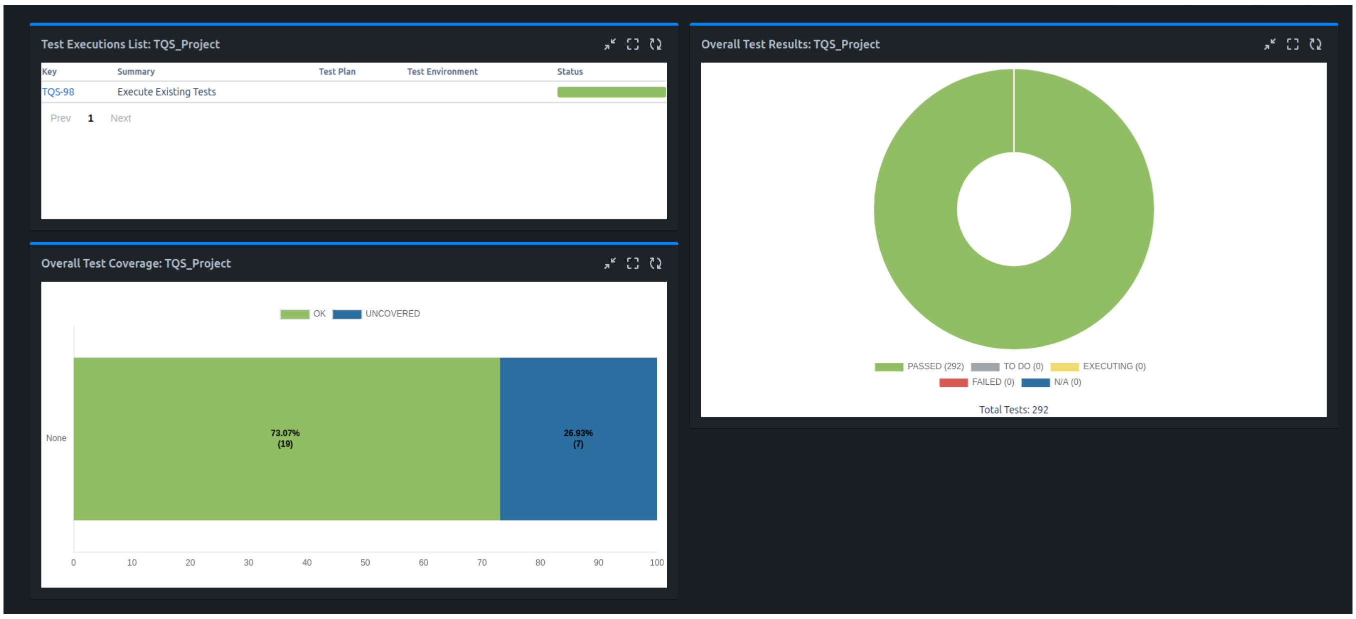
4.4 System and integration testing

System and integration testing encompassed a broader scope of the application. Integration tests were crafted using Selenium to perform end-to-end testing of critical user flows, validating that the entire system operated as expected when various components interacted. This type of testing helped uncover issues related to system integration, data flow, and overall system behaviour. Additionally, API testing was conducted using REST Assured, which validated the functionality, performance, and reliability of backend services. These tests ensured that APIs handled requests and responses correctly, maintaining robust communication between the client and server parts of the application.

5.Conclusion

We encountered challenges that prevented us from completing all the planned functionalities within the given timeframe. Time constraints impacted our ability to fully implement and integrate some of the features we had initially scoped for the project. However, the functionalities we did manage to complete are well-tested.

Below are two images showcasing our progress and code quality metrics:

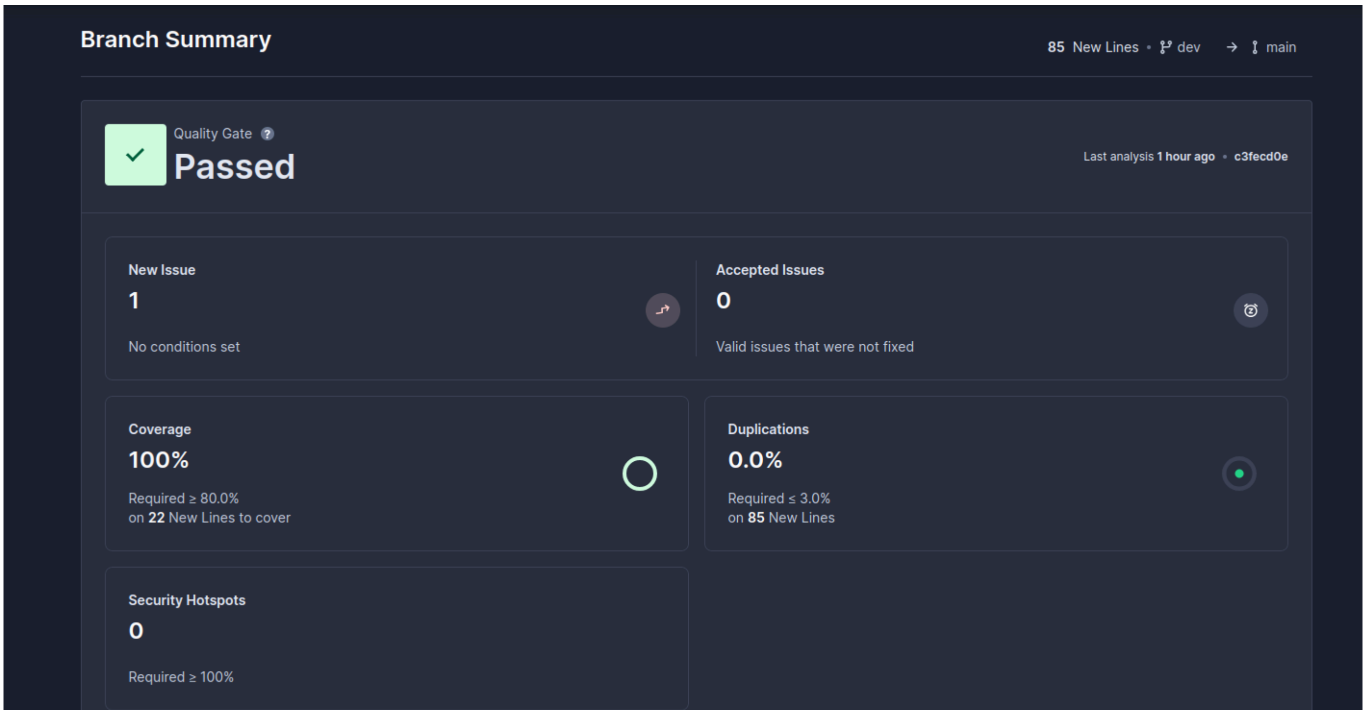


Test Executions List: TQS_Project

Key	Summary	Test Plan	Test Environment	Status
TQS-98	Execute Existing Tests			🟩

Prev 1 Next

Fig.5a) Jira dashboard with information about the test coverage and test results



Branch Summary

85 New Lines • dev → main

Quality Gate
Passed

Last analysis 1 hour ago • c3fecd0e

New Issue
1
No conditions set

Accepted Issues
0
Valid issues that were not fixed

Coverage
100%
Required ≥ 80.0%
on 22 New Lines to cover

Duplications
0.0%
Required ≤ 3.0%
on 85 New Lines

Security Hotspots
0
Required ≥ 100%

Fig.5b) SonarCloud dashboard with the final coverage analysis of our code